

Automatic Synthesis of Systems with Data

Léo Exibard

Affidavit

Je soussigné, Léo Exibard, déclare par la présente que le travail présenté dans ce manuscrit est mon propre travail, réalisé sous la direction scientifique d'Emmanuel Filiot et de Pierre-Alain Reynier dans le respect des principes d'honnêteté, d'intégrité et de responsabilité inhérents à la mission de recherche. Les travaux de recherche et la rédaction de ce manuscrit ont été réalisés dans le respect à la fois de la charte nationale de déontologie des métiers de la recherche et de la charte d'Aix-Marseille Université relative à la lutte contre le plagiat.

Ce travail n'a pas été précédemment soumis en France ou à l'étranger dans une version identique ou similaire à un organisme examinateur.

Fait à Marseille, le 9 juillet 2021.



Cette œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International.

This work is released into the public domain using the Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) Creative Commons License.

Digest

We often interact with machines that react in real time to our actions (robots, websites etc). They are modelled as reactive systems, that continuously interact with their environment. The goal of reactive synthesis is to automatically generate a system from the specification of its behaviour so as to replace the error-prone low-level development phase by a high-level specification design.

In the classical setting, the set of signals available to the machine is assumed to be finite. However, this assumption is not realistic to model systems which process data from a possibly infinite set (e.g. a client id, a sensor value, etc.). The goal of this thesis is to extend reactive synthesis to the case of data words. We study a model that is well-suited for this more general setting, and examine the feasibility of its synthesis problem(s). We also explore the case of non-reactive systems, where the machine does not have to react immediately to its inputs.

Keywords: Register Automata · Register Transducers · Reactive Synthesis · Church Problem · Data Words · Data Domains · Asynchronous Transducers · Computability · Continuity

Propos

Nous interagissons régulièrement avec des machines qui réagissent en temps réel à nos actions (robots, sites web etc). Celles-ci sont modélisées par des systèmes réactifs, caractérisés par une interaction constante avec leur environnement. L'objectif de la synthèse réactive est de générer automatiquement un tel système à partir de la description de son comportement afin de remplacer la phase de développement bas-niveau, sujette aux erreurs, par l'élaboration d'une spécification haut-niveau.

Classiquement, on suppose que les signaux d'entrée de la machine sont en nombre fini. Un tel cadre échoue à modéliser les systèmes qui traitent des données issues d'un ensemble infini (un identifiant unique, la valeur d'un capteur, etc). Cette thèse se propose d'étendre la synthèse réactive au cas des mots de données. Nous étudions un modèle adapté à ce cadre plus général, et examinons la faisabilité des problèmes de synthèse associés. Nous explorons également les systèmes non réactifs, où l'on n'impose pas à la machine de réagir en temps réel.

Mots Clés : Automates à registres · Transducteurs à registres · Synthèse réactive · Problème de Church · Mots de données · Domaines de données · Transducteurs asynchrones · Calculabilité · Continuité

Abstract

A reactive system is a system that continuously interacts with its environment. The environment provides an input signal, to which the system reacts with an output signal, and so on *ad infinitum*. In reactive synthesis, the goal is to automatically generate an implementation from a specification of the reactive and non-terminating input/output behaviours of a system. In the classical setting, the set of signals is assumed to be finite. However, this assumption is not realistic to model systems which process sequences of signals accompanied with data from a possibly infinite set (e.g. a client id, a sensor value, etc.), which need to be stored in memory and compared against each other.

The goal of this thesis is to lift the theory of reactive system synthesis over words on a finite alphabet to data words. The data domain consists in an infinite set whose structure is given by predicates and constants enriched with labels from a finite alphabet. In this context, specifications and implementations are respectively given as automata and transducers extended with a finite set of registers that they use to store data values. To determine the transition to take, they compare the input data with the content of the registers using the predicates of the domain.

In a first part, we consider both the register-bounded and unbounded synthesis problem; the former additionally asks for a bound on the number of registers of the implementation, along with the specification. We do so for different instances, depending on whether the specification is a nondeterministic, universal (a.k.a. co-non-deterministic) or deterministic automaton, for various domains. While the register-bounded synthesis problem is undecidable for non-deterministic specifications, we provide a generic approach consisting in a reduction to the finite alphabet case, that is done through automata-theoretic constructions. This allows to reprove decidability of register-bounded synthesis for universal specifications over $(\mathbb{N}, =)$, and to obtain new ones, such as the case of a dense order, or the ability of data guessing, all with a 2-ExpTime complexity. We then move to the unbounded synthesis problem, which is undecidable for specifications given by non-deterministic and universal automata, but decidable and ExpTime -complete for deterministic ones over $(\mathbb{N}, =)$ and $(\mathbb{Q}, <)$. We also exhibit a decidable subclass in the case of $(\mathbb{N}, <)$, namely one-sided specifications.

In a second part, we lift the reactivity assumption, considering the richer class of implementations that are allowed to wait for additional input before reacting, again over data words. Specifications are modelled as non-deterministic asynchronous transducers, that output a (possibly empty) word when they read an input data. Already in the finite alphabet case, their synthesis problem is undecidable.

A way to circumvent the difficulty is to focus on functional specifications, for which any input sequence admits at most one acceptable output. Targeting programs computed by input-deterministic transducers is again undecidable, so we shift the focus to deciding whether a specification is computable, in the sense of the classical extension of Turing-computability to infinite inputs. We relate this notion with that of continuity for the Cantor distance, which yields a decidable characterisation of computability for functional specifications given by asynchronous register transducers over $(\mathbb{N}, =)$ and for the superseding class of oligomorphic data domains, that also encompasses $(\mathbb{Q}, <)$. The study concludes with the case of $(\mathbb{N}, <)$, that is again decidable. Overall, we get PSPACE-completeness for the problems of deciding computability and refined notions, as well as functionality.

Résumé

Les systèmes réactifs sont caractérisés par une interaction constante avec leur environnement : celui-ci fournit un signal d'entrée, auquel le système répond par un signal de sortie, et ainsi de suite à l'infini. L'objectif de la synthèse réactive est de générer automatiquement l'implémentation d'un tel système à partir de la spécification de son comportement. Classiquement, l'ensemble des signaux est supposé fini. Cependant, ce cadre échoue à modéliser des systèmes qui traitent des signaux accompagnés de données issues d'un ensemble potentiellement infini (un identifiant unique, la valeur d'un capteur, etc.), qui doivent être stockées et comparées entre elles.

L'objectif de cette thèse est d'étendre la théorie de la synthèse réactive sur les mots à alphabet fini au cas des mots de données. Le domaine de données consiste en un ensemble infini, dont la structure est définie par des prédicats et des constantes, enrichi par un ensemble fini de signaux. Les spécifications et les implémentations sont alors respectivement représentées par des automates et des transducteurs à registres, qu'ils utilisent pour stocker les données. Pour déterminer la transition à prendre, ils comparent la donnée d'entrée au contenu de leurs registres à l'aide des prédicats du domaine.

Dans une première partie, nous considérons les problèmes de la synthèse à registres bornés et non-bornée. Dans le premier cas, l'algorithme prend en entrée une borne sur le nombre de registres de l'implémentation, en plus de la spécification à implémenter. Nous considérons plusieurs instances, selon que la spécification est un automate non-déterministe, universel (ou co-non-déterministe), ou encore déterministe, pour plusieurs domaines de données. Tandis que le problème de la synthèse à registres bornés est indécidable pour les spécifications non-déterministes, nous élaborons une approche générique qui permet de le réduire au cas d'un alphabet fini. Celle-ci permet de redémontrer la décidabilité de la synthèse à registres bornés à partir d'automates universels sur $(\mathbb{N}, =)$ et d'étendre le résultat à $(\mathbb{Q}, <)$, y compris en autorisant l'automate à deviner des données, tout cela en 2-ExpTime.

Quant à la synthèse non-bornée, elle est indécidable pour les spécifications données par des automates non-déterministes ou universels, mais décidable et ExpTime-complète pour les automates déterministes sur $(\mathbb{N}, =)$ et $(\mathbb{Q}, <)$. Nous exhibons également une sous-classe décidable dans le cas de $(\mathbb{N}, <)$, à savoir les spécifications unilatérales.

Dans une seconde partie, nous examinons comment étendre au cas non-réactif, où l'implémentation est autorisée à attendre d'obtenir plus d'information avant de sélectionner son signal de sortie, toujours dans le cadre des mots de données. Les spécifications sont modélisées par des transducteurs non-déterministes asynchrones, qui produisent un mot

(possiblement vide) à chaque fois qu'ils lisent une entrée. Déjà dans le cas fini, un tel problème est indécidable pour cette classe de spécifications.

Une manière de contourner la difficulté est de traiter le cas des spécifications fonctionnelles, pour lesquelles chaque suite infinie d'entrées admet au plus une suite de sorties. Pour les implémentations données par des transducteurs déterministes sur l'entrée, le problème est indécidable, aussi nous intéressons-nous au problème de la calculabilité au sens de Turing, classiquement étendue au cas des mots infinis. Nous lions cette notion à celle de continuité pour la distance de Cantor, ce qui nous fournit une caractérisation de la calculabilité qui est décidable pour les fonctions définies par des transducteurs non-déterministes asynchrones sur $(\mathbb{N}, =)$ et pour la classe des domaines oligomorphes, qui englobe $(\mathbb{N}, =)$ et $(\mathbb{Q}, <)$. L'étude se conclut par le cas de $(\mathbb{N}, <)$, également décidable. Pour ces trois domaines, les problèmes de calculabilité et ses déclinaisons, ainsi que la fonctionnalité, sont décidables en espace polynomial ($PSPACE$).

Contents

Affidavit	ii
Digest / Propos	iii
Abstract	iv
Résumé	vi
Contents	viii
1. Introduction en français	1
1.1. Synthèse de programmes	1
1.2. Les systèmes réactifs	2
1.3. Extension aux alphabets infinis	9
1.4. Extension au cas asynchrone	18
1.5. Plan	21
1.6. Comment lire cette thèse ?	22
2. Introduction in English	24
2.1. Program Synthesis	24
2.2. Reactive Systems	25
2.3. Reactive Synthesis and the Church Problem	28
2.4. Extension to the Infinite Alphabet Case	31
2.5. Further Extension to the Asynchronous Case	38
2.6. Outline	41
2.7. How to Read this Thesis?	41
I. REACTIVE SYNTHESIS	44
3. Reactive Synthesis over Finite Alphabets	45
3.1. Relations and Functions	46
3.2. Uniformisation and Program Synthesis	46
3.3. The Reactive Synthesis Problem	48
3.4. The Church Synthesis Problem	52
3.5. Games for Reactive Synthesis	65
4. Register Automata over a Data Domain	71
4.1. Data Words	73
4.2. Tests and Valuations	75
4.3. Register Automata	76
4.4. General Properties	83
4.5. Non-Deterministic Register Automata with Equality	92
4.6. Non-Deterministic Register Automata with a Dense Order	106
4.7. Register Automata with a Discrete Order	108
4.8. Universal Register Automata	118
4.9. Register Automata and Arithmetic	121
4.10. Extensions of the Model	122

5. Synthesis Problems over Data Words	130
5.1. Reactive Synthesis over Data Words	130
5.2. Data Automatic Specifications	131
5.3. Register Transducers	132
5.4. The Church Synthesis Problem	136
6. Register-Bounded Synthesis of Register Transducers	137
6.1. A Generic Approach	138
6.2. Universal Register Automata Specifications	147
6.3. Non-Deterministic Register Automata Specifications	152
7. Unbounded Synthesis of Register Transducers	159
7.1. Non-Deterministic Register Automata Specifications	160
7.2. Universal Register Automata Specifications	160
7.3. Deterministic Register Automata Specifications	162
8. Partial Versus Total Domain	195
8.1. Domains of Specification Automata	195
8.2. Uniformisation Problem	197
8.3. Good-Enough Synthesis	197
9. Conclusion	199
 II. COMPUTABILITY OF FUNCTIONS DEFINED BY TRANSDUCERS OVER INFINITE ALPHABETS	 202
10. Non-Deterministic Asynchronous Register Transducers	205
10.1. The Model	205
10.2. Register Transducers and Register Automata	209
10.3. Functions Recognised by Transducers	212
10.4. Closure under Composition	214
11. ω-Computability and Continuity	216
11.1. Computability	216
11.2. Continuity	219
11.3. ω -Computability versus Continuity	222
12. Exploration of Various Data Domains	226
12.1. Data Domains with Equality	226
12.2. Oligomorphic Data Domains	240
12.3. Discretely Ordered Data Domains	270
13. Conclusion	306
14. Thesis Conclusion	309
14.1. Abstraction of the Behaviour of Register Automata	309
14.2. Generalisation to Other Formalisms	310
14.3. Minimisation and Learning	311

APPENDIX	313
A. Graphical Conventions	314
A.1. Relations and Functions, Input and Output	314
A.2. Reactive Synthesis	314
A.3. ω -Automata	314
A.4. Register Automata	315
A.5. Synchronous Sequential Register Transducers	315
A.6. Non-Deterministic Asynchronous Register Transducers	315
B. Details of Chapter 3	316
B.1. Section 3.4.1 (Logics for Specifications)	316
B.2. Section 3.4.3 (Automata)	317
C. Details of Part II	320
Bibliography	322
Index of Notations	333

List of Figures

1.1. Un automate déterministe qui détermine si son entrée contient un nombre impair de a	2
1.2. Un automate non-déterministe qui vérifie que l'antépénultième lettre est un a	3
1.3. Un transducteur séquentiel qui permute son entrée selon qu'il a lu un nombre pair de a	4
1.4. Un ω -transducteur qui modélise une machine à café	5
1.5. Un automate universel de co-Büchi qui vérifie que toute occurrence de commander_cafe est ultérieurement suivie de verser_cafe	9
1.6. Un automate à registres déterministe qui vérifie que la première donnée apparaît à nouveau ultérieurement	10
1.7. Un transducteur à registres séquentiel et synchrone modélisant une machine à café	11
1.8. Un automate à registres universel de co-Büchi qui vérifie que chaque requête commander_cafe de chaque client id finit par être satisfaite	11
2.1. A deterministic finite-state automaton that checks whether its input contains an odd number of a	25
2.2. A non-deterministic finite-state automaton that checks whether the second letter before the end is an a	26
2.3. A sequential transducer that shifts its input depending whether it contains an even number of a	26
2.4. An ω -transducer modelling a coffee machine	27
2.5. A universal co-Büchi automaton checking that any order_coffee is eventually followed by pour_coffee	30
2.6. A deterministic register automaton checking that the first data value appears again	32
2.7. A synchronous sequential register transducer modelling a coffee machine	32
2.8. A universal co-Büchi register automaton checking that every order_coffee of any client id is eventually satisfied	33
3.1. Uniformisation of a relation by a function	46
3.2. The program synthesis problem	47
3.3. Reactive systems	50
3.4. A non-deterministic Büchi automaton recognising ω -words with finitely many a	59
3.5. A deterministic Büchi automaton recognising ω -words with infinitely many a	59
3.6. An ω -automaton recognising $L_{IO} = (IO)^\omega$	59
3.7. A universal co-Büchi automaton checking that every request is eventually granted	60
3.8. A universal co-Büchi automaton checking that input and output alternate, and that every request is eventually granted	62
3.9. Two finite-state machines whose ω -behaviour implements the request-grant specification in different ways	63
3.10. Two synchronous sequential transducers implementing the request-grant specification in different ways	64

3.11. Adam and Eve	66
4.1. A series of register automata, recognising the data languages of Example 4.1	81
4.2. Comparison of the different variations of register automata over data domains with equality	83
4.3. An unambiguous register automaton checking that the last data value appears before	98
4.4. A register automaton that is syntactically non-empty, but accepts no data words.	98
4.5. A deterministic register automaton recognising blocks of increasing bounded data values	108
4.6. Graphical depiction of a constraint sequence	110
4.7. Extracting an infinitely decreasing one-way chain from a two-way one . .	114
4.8. Extracting arbitrarily deep ceiled one-way chains from two-way ones . .	115
4.9. Two register automata with guessing	123
4.10. A register automaton with non-deterministic reassignment that silently increments its register	124
4.11. A fresh-register automaton that recognises words with pairwise distinct data values	128
4.12. A universal register automaton that recognises globally fresh data values .	128
5.1. A universal co-Büchi register automaton checking that every request of any client id is eventually granted	132
5.2. A synchronous sequential register transducer computing the identity function	134
5.3. Two synchronous sequential register transducers that satisfy the request-grant specification over data words	135
6.1. A data automatic specification that is not realisable	153
7.1. A specification automaton whose automaton game fails to abstract the reactive synthesis problem	169
7.2. A specification automaton over a dense order that is realisable, but not by any register transducer	177
7.3. A specification automaton in which Eve cannot uniformly concretise all behaviours of Adam	178
7.4. Simulating increment (the gadget for decrement is similar) and zero-tests.	179
7.6. An $xy^{(m)}$ -connecting chain.	189
7.7. Two possible suffix constraints, that impose to leave sufficient room between r_0, r_1 and r_2	189
7.8. Both x and y were already present.	191
7.9. x is a new value at the top	191
7.10. x is inserted between two registers	191
10.1. A register transducer T_{rename} modelling Example 10.1	208
10.2. A variant $T_{\text{rename}2}$ with guessing modelling Example 10.1	208
10.3. A functional non-deterministic register transducer $T_{\text{rename}3}$, that models Example 10.2	213
10.4. A variant $T_{\text{rename}4}$ modelling Example 10.2 with non-deterministic reassignment	213

11.1. A register transducer with non-deterministic reassignment which puts the last data value of each block at its beginning	217
11.2. A non-deterministic register transducer whose output depends on whether the first input data value appears again	218
12.1. A pattern characterising non-emptiness of the domain of functions definable by a non-deterministic ω -transducer	230
12.2. A pattern characterising joint coaccessibility of two states in a non-deterministic ω -transducer	231
12.3. A pattern characterising non-continuity of functions definable by a NRT over $(\mathbb{D}, =)$	235
12.4. Decomposition of the runs of a NRT in order to pump	237
12.5. A pattern characterising non-uniform continuity of functions definable by a NRT over $(\mathbb{D}, =)$	240
12.6. Two register automata that are empty over $(\mathbb{N}, <)$, but not over $(\mathbb{Q}, <)$. . .	271
12.7. A register automaton that is not empty over $(\mathbb{Z}, <)$ but does not have any loop	271
12.8. A schema of the relative order between values	276
12.9. How to send an interval to a strictly bigger one while preserving $\mathbb{N}$	278
12.10. A non-deterministic register transducer whose output depends on whether \$ appears in the input	298

List of Tables

1.1.	Décidabilité de la synthèse à registres bornés et non-bornée sur $(\mathbb{D}, =)$	16
1.2.	Décidabilité de la synthèse à registres bornés et non-bornée sur $(\mathbb{Q}, <)$	16
2.1.	Decidability status of register-bounded and unbounded synthesis over $(\mathbb{D}, =)$	36
2.2.	Decidability status of register-bounded and unbounded synthesis over $(\mathbb{Q}, <)$	36

Introduction

1.

This introduction is in French. The English version can be found in Chapter 2.

1.1. Synthèse de programmes

La programmation consiste en l'automatisation de tâches. S'agissant elle-même d'une tâche, il est naturel de se demander s'il est possible de l'automatiser ; tel est l'objectif de la synthèse de programmes. Considérons par exemple le tri d'une liste de nombres : plutôt que de spécifier *comment* le programme devrait procéder (construire la liste en insérant chaque valeur une à une, comme lorsque l'on trie des cartes à jouer), il serait préférable de spécifier *ce* qu'il devrait faire (prendre en entrée une liste de nombres et produire une liste triée qui contient les mêmes éléments). Autrement dit : étant donné la spécification du comportement attendu du programme (*quoi*), un algorithme de synthèse devrait produire un programme qui présente ce comportement (*comment*) si un tel programme existe, et Non si ce qui est demandé est impossible. Un tel programme est appelé une *implémentation* de la spécification donnée en entrée ; nous disons qu'il *satisfait* la spécification.

Disons les choses formellement. Dans sa forme la plus générale, un problème de synthèse a deux paramètres : un ensemble In d'entrées et un ensemble Out de sorties pour les programmes que l'on souhaite synthétiser. Il lie une classe $\mathcal{S}$ de spécifications à une classe $\mathcal{I}$ d'implémentations. Une spécification $S \in \mathcal{S}$ est une relation binaire $S \subseteq \text{In} \times \text{Out}$ qui est interprétée de la manière suivante : pour chaque $x \in \text{In}$ et chaque $y \in \text{Out}$, $(x, y) \in S$ si et seulement si y est une sortie acceptable pour l'entrée x . Une implémentation est une fonction $I : \text{In} \rightarrow \text{Out}$ qui, pour chaque entrée $x \in \text{In}$, choisit une sortie acceptable $y \in \text{Out}$. Ainsi, I satisfait S lorsque pour tout $x \in \text{In}$, $(x, I(x)) \in S$. Le problème de synthèse pour $\mathcal{S}$ et $\mathcal{I}$ est alors défini de la manière suivante : étant donné une (représentation finie de) $S \in \mathcal{S}$, existe-t-il $I \in \mathcal{I}$ telle que I satisfait S ? Et, le cas échéant, l'algorithme de synthèse doit produire un programme qui exécute I .

Il s'agit évidemment d'un objectif ambitieux, et nous savons depuis bien longtemps qu'il est hors de portée pour les langages de programmation généralistes. En effet, résoudre le problème de la synthèse revient alors à trouver un algorithme qui, étant donné la spécification d'un problème (de décision ou d'optimisation), lui trouve une solution procédurale si elle existe, et produit un témoin d'indécidabilité sinon. D'une certaine manière, cette impossibilité est rassurante, car elle montre que l'informatique a de beaux jours devant elle. Elle a pu avoir dans un premier temps un effet dissuasif sur les recherches effectuées dans le domaine de la synthèse. Pour autant, de la même manière que l'indécidabilité du problème de l'arrêt [1] a déclenché des investigations dans le domaine de l'analyse de programmes, ce résultat d'impossibilité a ouvert la voie à la recherche de restrictions du problème qui seraient décidables. * Il y a essentiellement deux manières de récupérer la décidabilité. Premièrement, il est possible de « guider » la synthèse en fournissant de

Nous décrivons le *tri par insertion*. Techniquement, il n'est pas optimal, mais présente l'avantage de correspondre à la pratique quotidienne.

Nous attirons l'attention du lecteur ou de la lectrice sur le fait que la synthèse est une notion d'ordre supérieur, au sens où un algorithme de synthèse est un algorithme auquel l'on demande d'écrire des programmes. C'est pour clarifier cet état de fait que nous distinguons les termes d'« algorithme » et de « programme ».

Dans le corps du document, nous privilégions le terme « réaliser » (*to realise*) à celui de « satisfaire », du fait de ses accointances avec la logique formelle.

À nouveau, soyons prudents ici quant à l'ordre dans lequel on se situe : les ensembles In et Out concernent le programme que l'on veut générer ; l'algorithme de synthèse, quant à lui, prend en entrée une spécification de $\mathcal{S}$ et produit en sortie une implémentation de $\mathcal{I}$.

En d'autres termes, la quête d'un algorithme de synthèse pour les langages généralistes se ramène à celle d'un algorithme qui serait un informaticien parfait, capable de déterminer si un problème donné est faisable et, le cas échéant, d'en produire une solution. Formellement, on peut démontrer l'impossibilité de cette entreprise en la réduisant au problème de l'arrêt pour les machines de Turing, ou bien au Entscheidungsproblem : dès lors que l'on peut spécifier des réquisits arithmétiques, résoudre le problème de la synthèse implique en particulier de décider si une formule arithmétique donnée est vraie, ce que l'on sait impossible depuis Turing et Church.

[1]: TURING (1936), « On Computable Numbers, with an Application to the Entscheidungsproblem »

* À cet égard, l'histoire de toute l'informatique jusqu'à nos jours n'a été que l'histoire de luttes de classes de complexité.

l'information additionnelle à l'algorithme de synthèse. C'est l'idée qui sous-tend la synthèse guidée par la syntaxe (*syntax-guided synthesis* [2]), qui prend en entrée un squelette du programme à générer, et l'étoffe afin d'obtenir un programme complet qui satisfait la spécification. La précision du squelette donné en entrée permet d'ajuster la difficulté du problème. L'approche de la synthèse bornée (*bounded synthesis* [3]) repose sur une idée similaire : en fixant une borne sur la taille du programme cible, nous bornons l'espace des programmes à explorer, ce qui garantit que la procédure termine. L'autre manière d'obtenir la décidabilité est de restreindre la classe des spécifications ou des implémentations.

[2]: ALUR et al. (2013), « Syntax-guided synthesis »

[3]: FINKBEINER et SCHEWE (2013), « Bounded synthesis »

Les deux approches ne sont pas mutuellement exclusives.

1.2. Les systèmes réactifs

1.2.1. Les automates

À ce titre, les programmes représentables par des machines à états finis, ou automates, constituent une classe de choix. Comme leur nom l'indique, de telles machines sont dotées d'un ensemble fini d'états. Elle traitent leur entrée lettre par lettre de gauche à droite, en utilisant leurs états pour stocker l'information. Elles sont notamment utilisées pour modéliser les systèmes embarqués ou les circuits électroniques. Dans leur forme la plus pure, ils consistent en des accepteurs pour un langage qui, étant donné une séquence de lettres, ou *mot*, répondent Oui ou Non selon que le mot appartient au langage ou non. Pour prendre un exemple simple, considérons le langage des mots qui contiennent un nombre impair de *a*. Pour déterminer si un mot donné appartient au langage, un programme n'a pas besoin de compter le nombre de *a*, seulement de garder en mémoire la parité de ce nombre. La FIGURE 2.1 représente un automate qui reconnaît ce langage. Dans cet exemple, lorsqu'il est

Le lecteur connaisseur, qui sait déjà ce qu'est un automate, peut se rendre à la Section 1.2.2

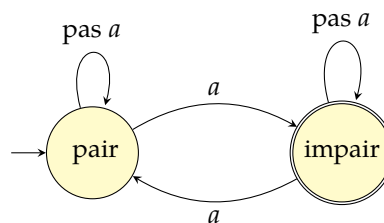


FIGURE 1.1. : Un automate avec deux états mémoire qui détermine si son entrée contient un nombre impair de *a*. Les états sont représentés par des cercles, et les transitions par des flèches. Par exemple, quand l'automate est dans l'état « pair » et lit un *a*, il transitionne vers l'état « impair ». Comme indiqué par la flèche entrante, il commence dans l'état « pair ». La machine répond Oui si et seulement si son exécution termine dans un état final, distingué par un double cercle (ici, seul l'état « impair » est acceptant). Pour illustrer notre propos, nous invitons le lecteur ou la lectrice à méditer sur le comportement de l'automate lorsqu'il lit le mot *baa*. Tout d'abord, il commence dans son état initial « pair », puisque avant de lire le mot, il a lu 0 occurrences de la lettre *a*, et 0 est pair. Lorsqu'il lit la lettre *b*, il suit la boucle sur l'état « pair » et reste dans cet état (puisque *b* n'est pas *a*). Ensuite, il lit *a* et va dans l'état « impair ». Enfin, il lit *a* de nouveau, et retourne dans l'état « impair ». Il répond alors Non, puisque « pair » n'est pas un état acceptant. Il est aisé de vérifier que lors d'une exécution quelconque, l'automate est dans l'état « pair » précisément lorsqu'il a vu un nombre pair de *a* (et inversement pour « impair »).

dans un état donné et lit une lettre, l'automate a une seule transition possible ; on dit alors qu'il est « déterministe ». En général, il est pratique de doter ces machines de la possibilité de « deviner » la transition à prendre parmi plusieurs possibles ; de telles machines sont appelées « non-déterministes ». Examinons par exemple le langage des mots dont l'antépénultième lettre est un *a*. Puisque l'automate lit son entrée de

Notez que dans la définition d'un automate non-déterministe, il n'est pas *requis* qu'il ait plusieurs transitions possibles parmi lesquelles choisir, au sens où la notion d'automate non-déterministe généralise celle d'automate déterministe. C'est pourquoi, bien que cela semble contre-intuitif, un automate déterministe est, en particulier, non-déterministe.

gauche à droite, il ne peut pas savoir à l'avance combien de lettres il lui reste à lire avant la fin. La FIGURE 1.2 représente un automate non-déterministe qui reconnaît ce langage. Dans la pratique, un programme

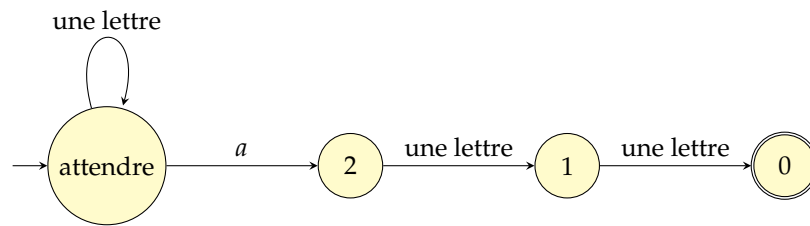


FIGURE 1.2 : Un automate non-déterministe qui vérifie que l'antépénultième lettre est un *a*. Il commence dans l'état « attendre ». Lorsqu'il lit une lettre qui n'est pas un *a*, il reste dans cet état. Lorsqu'il lit un *a*, il doit faire un choix : ou bien rester dans l'état « attendre » en prenant la boucle, ou bien transitionner vers l'état 2. Pour faire un tel choix, il devine si le *a* qu'il lit est deux lettres avant la fin du mot. Si c'est le cas, il va dans l'état 2, et vérifie qu'il a deviné correctement en lisant les deux lettres suivantes. Si le mot se termine à ce moment-là, il répond Oui, puisqu'il est alors dans l'état 0, qui est acceptant. S'il y a encore des lettres à lire, l'exécution échoue puisqu'il n'y a pas de transition à prendre depuis l'état 0. Par définition, un automate non-déterministe répond Oui si et seulement si *au moins* une de ses exécutions termine dans un état acceptant. Par exemple, sur l'entrée *ababb*, lorsqu'il lit le premier *a*, l'automate peut soit aller dans l'état 2, soit rester dans l'état « attendre ». S'il opte pour la première possibilité, lorsqu'il a lu *aba*, il est dans l'état 0 et a encore deux lettres à lire, et l'exécution échoue. Dans le second cas, il doit à nouveau faire ce choix en lisant le second *a*. S'il reste dans l'état « attendre », il y reste jusqu'à la fin de l'exécution puisqu'il n'y a plus de *a* à lire, et l'exécution n'est pas acceptante puisqu'elle termine dans un état non-acceptant. Cependant, s'il transitionne vers 2, l'exécution termine dans l'état 0 et l'automate accepte. Ainsi, il existe une exécution acceptante, ce qui signifie que *ababb* est accepté par l'automate, qui répond Oui.

n'est pas capable de « deviner » la bonne exécution, sauf en exécutant tous les calculs en parallèle et en vérifiant qu'il en existe au moins un qui est acceptant. Par conséquent, le non-déterminisme devrait être interprété comme une manière compacte de représenter un ensemble de comportements possibles. Il est bien connu que les automates non-déterministes admettent toujours un équivalent déterministe, qui peut cependant être exponentiellement plus grand (en termes de nombre d'états).

Pour représenter des spécifications, nous aurons également recours aux automates *universels*, ou « co-non-déterministes ». Plutôt que de demander qu'*au moins* une exécution est acceptante, nous demandons à ce qu'elles le soient *toutes*. Si nous revenons à l'automate de la FIGURE 1.2, vu comme un automate universel et en intervertissant les états acceptants et rejetants, il accepte précisément les mots dont l'antépénultième lettre *n'est pas* un *a*. Plus généralement, les sémantiques non-déterministes et universelles sont *duals* : en interprétant un automate non-déterministe comme un automate universel, et en intervertissant les états acceptants et rejetants, nous obtenons un automate qui reconnaît le complément du langage.

1.2.2. Les transducteurs

Souvent, nous attendons d'un programme qu'il fournisse plus d'information que le seul fait d'accepter ou de rejeter un mot. Par exemple, si nous revenons à l'automate de la FIGURE 1.1, nous pourrions lui demander de signaler tout au long du calcul si le nombre de *a* qu'il a vus est impair. Sur l'entrée *aba*, il produirait alors la séquence *impair · impair · pair* (si l'on suppose qu'il lit d'abord une lettre avant de produire une sortie). Dans ce cas particulier, il suffit de demander à l'automate d'afficher son état courant le long de son exécution. Cependant, en général, il est plus pratique de distinguer les états internes de la machine et ses interactions

avec son environnement. Pour ce faire, nous enrichissons plutôt les transitions avec des lettres de sortie : à chaque fois qu'elle lit une lettre d'entrée, la machine transitionne vers un état, et, ce faisant, produit une lettre de sortie. Cela permet de représenter des programmes qui effectuent des transformations de leur entrée, par exemple remplacer a par b , b par c et c par a lorsqu'un nombre pair de a a été lu, et inversement $a \rightarrow c \rightarrow b \rightarrow a$ lorsque ce nombre est impair (cf FIGURE 1.3).

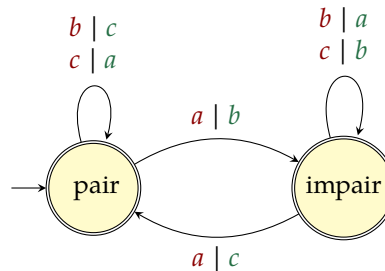


FIGURE 1.3. : Un transducteur séquentiel qui permute son entrée, dans un sens ou dans l'autre selon qu'il a lu un nombre pair de a . Pour simplifier, nous supposons qu'il lit seulement des mots dont les lettres sont issues de l'alphabet $\{a, b, c\}$. Les entrées sont colorées en rouge, les sorties en vert, et une transition « pair $\xrightarrow{a|b}$ impair » se lit « lorsque l'on est dans l'état "pair" et que l'on lit un a , on produit b et transitionne vers l'état "impair" ». Par exemple, sur l'entrée $abbac$, le transducteur produit la sortie $baaca$. Et, sur l'entrée $acab$ (acronyme bien connu de « all cats are beautiful »), la machine produit $bbcc$. À l'instar des automates, les transducteurs bénéficient d'un mécanisme d'acceptance : on peut demander à ce que la machine n'accepte que les mots contenant un nombre impair de a en considérant l'état « impair » comme le seul état acceptant.

ω -Transducteurs

Dans notre cadre d'analyse, nous sommes principalement intéressés par l'interaction entre le programme et son environnement, c'est-à-dire la relation entre ses entrées et ses sorties, plus que par sa terminaison. Ainsi, nous considérons le cas d'un système en interaction constante avec son environnement. Ce dernier fournit un signal d'entrée, modélisé comme l'élément (lettre) d'un ensemble fini (alphabet). Ensuite, le système réagit par une lettre de sortie, et ceci indéfiniment. Leur interaction donne lieu à une suite infinie de signaux d'entrée et de sortie entrelacés. Nous supposons que l'interaction ne termine jamais, puisque l'étude est focalisée sur le contrôle du système, plutôt que sur son aspect calculatoire. Un tel système est appelé « réactif ». Précisons en passant qu'il y a un procédé d'abstraction à l'œuvre ici, intrinsèque à la démarche de modélisation : nous avons besoin d'objets qui soient manipulables algorithmiquement, que nous construisons à partir du système réellement existant. Ainsi, nous travaillons toujours sous l'hypothèse que le modèle mathématique est en adéquation avec le système considéré.

Un exemple paradigmatique de système réactif est celui d'un serveur en interaction avec ses clients : ces derniers adressent des requêtes au serveur, qu'il doit satisfaire en temps voulu. Ici, nous présentons une variation sur le même thème, qui présente essentiellement les mêmes caractéristiques, à savoir le cas d'une machine à café, modélisée comme le système, qui interagit avec ses utilisateurs qui constituent collectivement son environnement. Les utilisateurs fournissent les signaux d'entrée via les boutons de la machine, qui réagit avec des signaux de sortie consistant à effectuer des actions (démarrer, verser le café, afficher des informations sur l'écran, s'éteindre, etc). Typiquement, une interaction consiste en un

utilisateur qui allume la machine, commande un café, le reçoit et éteint la machine. Si personne n'utilise la machine jusqu'à la fin des temps, elle reste inactive indéfiniment. Une telle interaction peut être modélisée par le mot infini suivant :

`appuyer_sur_on · demarrer · commander_cafe · verser_cafe · appuyer_sur_off · eteindre · (inactif · inactif)ω`

où l'exposant ω dans $(\text{inactif} \cdot \text{inactif})^\omega$ signifie que la séquence `inactif · inactif` est répétée à l'infini. L'alphabet d'entrée est défini comme $\{\text{appuyer\_sur\_on}, \text{commander\_cafe}, \text{appuyer\_sur\_off}, \text{inactif}\}$ et l'alphabet de sortie est $\{\text{demarrer}, \text{verser\_cafe}, \text{eteindre}, \text{inactif}\}$. Ici, la notion d'alphabet est entendue comme celle d'un ensemble fini : les éléments des alphabets précités sont des mots, mais pourraient tout aussi bien être remplacés par des symboles (A pour `appuyer_sur_on`, V pour `verser_cafe`, etc).

Par convention, les entrées sont colorées en rouge, et les sorties en vert. Le symbole $\cdot$ dénote la concaténation.

La machine est guidée par un contrôleur qui réagit en temps réel aux signaux d'entrée. S'il est encodé dans une puce électronique, il a un nombre d'états fini, et peut donc être modélisé par un transducteur déterministe qui opère sur des mots infinis ; nous les appelons ω -transducteurs. Nous pourrions alors modéliser notre machine à café par l' ω -transducteur de la FIGURE 1.4. En général, nous supposons que de tels programmes acceptent toutes les séquences d'entrées possibles, aussi tous leurs états sont-ils implicitement acceptants. Insistons sur le fait que l'interaction doit être

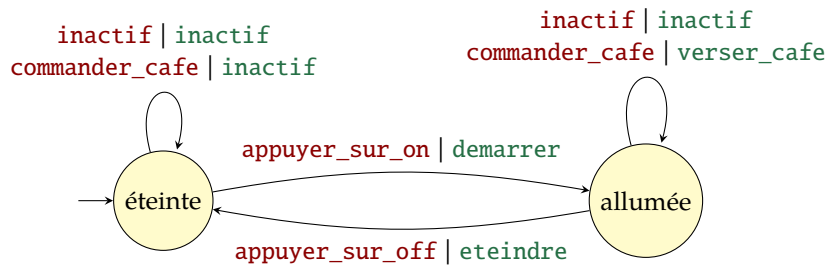


FIGURE 1.4. : Un ω -transducteur qui modélise une machine à café. Elle est initialement éteinte, et peut être allumée en appuyant sur « on ». Lorsqu'elle est éteinte, les autres commandes n'ont aucun effet. Lorsqu'elle est allumée, il est possible de commander un café, auquel cas la machine s'exécute diligemment.

imaginée sur l'ensemble du cycle de vie de la machine. Par exemple, lorsqu'elle s'éteint, l'interaction n'est pas terminée, puisqu'elle peut être rallumée par la suite. Le cycle de vie global est lui-même fini, puisque la machine cessera un jour de fonctionner en dépit de tous les efforts pour la maintenir en vie, ne serait-ce qu'au moment de l'effondrement du système solaire qui adviendra dans un temps long mais fini. Cependant, focaliser l'étude sur l'interaction permet d'exprimer plus aisément des réquisits qui concernent une fin prématurée de l'exécution, par exemple lorsque la machine s'éteint sans crier gare. C'est le cas par exemple de la séquence fautive `appuyer_sur_on · demarrer · commander_cafe · eteindre · (inactif · inactif)ω`, où la machine s'éteint avant d'avoir servi un café commandé. Ainsi, une spécification typique pourrait requérir que chaque occurrence de `commander_cafe` est immédiatement suivie de `verser_cafe`. Une telle exigence pourrait être trop élevée. Premièrement, on ne peut pas attendre d'une machine qu'elle serve un

café en étant éteinte[†], aussi cette condition devrait-elle être limitée aux périodes ouvertes par une occurrence de **démarrer**, qui ne contiennent pas **eteindre**. Cependant, une telle spécification est trivialement satisfaite par une machine qui ne s'allume jamais, aussi faudrait-il également spécifier de quelle manière la machine démarre. Par ailleurs, il est possible que la machine prenne du temps pour servir un café, par exemple si elle doit effectuer des opérations internes à cette fin, comme moulinier le café, chauffer l'eau, etc. Par conséquent, elle n'est peut-être pas capable de réagir immédiatement, auquel cas nous pouvons relâcher la spécification et demander à ce que le café soit servi dans les k unités de temps suivantes pour un certain k fixé, voire simplement que la machine finisse par servir un café (sans borne de temps).

1.2.3. La synthèse réactive et le problème de Church

En définitive, une spécification met en relation des suites infinies de signaux d'entrée avec les suites infinies de signaux de sortie qui constituent des comportements acceptables pour le système. Puisque nous nous intéressons aux systèmes réactifs qui sont en interaction constante avec leur environnement, il est souvent plus pratique de voir la spécification de manière entrelacée, c'est-à-dire comme les mots qui consistent en l'entrelacement d'une suite infinie d'entrées et d'une suite correspondante de sorties acceptable. Par exemple, si l'on demande que **verser_cafe** adienne au bout d'un moment, et ce après chaque occurrence de **commander_cafe**, cela correspond à la spécification qui contient tous les mots satisfaisant cette propriété.

Le problème de la synthèse réactive consiste alors, étant donné une spécification (représentée de manière finie), à générer un programme réactif qui satisfait la spécification si un tel programme existe, et de fournir un témoin d'infaisabilité dans le cas contraire. Ce problème fut posé par Church en 1957 qui demanda ce qui est désormais connu comme le problème de Church [5] :

Étant donné une exigence à laquelle un circuit doit satisfaire, nous pouvons supposer qu'un tel réquisit est exprimé dans un formalisme adapté qui est une extension de l'arithmétique limitée [c'est-à-dire la théorie du second ordre des entiers naturels, restreinte à l'addition]. Le problème de la synthèse consiste alors à trouver une fonction calculable qui représente un circuit satisfaisant à cette exigence (ou, alternativement, de déterminer qu'un tel circuit n'existe pas). ([4], notre traduction)

Les systèmes réactifs sont notoirement difficiles à concevoir correctement, et l'objectif de la synthèse est de générer automatiquement des systèmes qui sont *corrects par construction*. Dans le cadre défini par Church, la classe des implémentations consiste en les programmes réactifs qui peuvent être exécutés par des circuits booléens, c'est-à-dire utilisant une quantité finie de mémoire. Autrement dit, des ω -transducteurs.

[5]: THOMAS (2009), « Facets of Synthesis : Revisiting Church's Problem »

[4]: CHURCH (1957), « Applications of recursive arithmetic to the problem of circuit synthesis »

[†] Pas plus qu'on ne peut demander du lait à un bouc.

La logique monadique du second ordre (MSO)

Concernant le formalisme de spécification, Church a principalement considéré la logique monadique du second ordre (MSO) sur les mots infinis avec le prédicat $+1$ (ou, de manière équivalente, $\leq$), et des prédicats pour les lettres de l'alphabet d'entrée et de sortie. Cette logique est capable d'exprimer les différentes spécifications que nous avons ébauchées précédemment. Par exemple, l'on peut demander à ce que `verser_cafe` advienne après `commander_cafe` (sans borne de temps) de la manière suivante :

$$\forall x, \text{commander\_cafe}(x) \Rightarrow \exists y, y \geq x \wedge \text{verser\_cafe}(y)$$

La synthèse vue comme un jeu

Dans un rapport technique non publié [6], McNaughton remarqua que le problème de la synthèse peut être vu comme un jeu à deux joueurs à durée infini [7], que nous appelons le jeu de Church. Un joueur modélise le système, l'autre, son environnement. L'objectif du système est de satisfaire la spécification ; l'environnement est antagoniste et cherche à ce qu'elle soit enfreinte[‡]. Dans cette perspective, une stratégie gagnante pour le système correspond à une implémentation de la spécification, tandis qu'une stratégie gagnante pour l'environnement témoigne de l'infaisabilité de la spécification. En 1969, Büchi et Landweber ont résolu le problème pour les spécifications MSO à l'aide de méthodes issues de la théorie des jeux [8]. L'idée est de convertir la formule logique qui exprime la spécification en une machine à états finis qui reconnaît les interactions valides (plus précisément un ω -automate, c'est-à-dire un automate à états finis qui reconnaît des mots infinis), autrement dit les entrelacements d'entrées avec leurs sorties acceptables. Cette machine constitue alors un « observateur » pour le jeu, et définit sa condition de victoire. Reste alors à résoudre le jeu pour en déterminer le vainqueur, ce qui peut être fait par un calcul de points fixes — ou attracteurs — dans le cas des jeux de parité (voir par exemple [9]). Par la suite, Rabin a élaboré une solution qui repose sur les automates d'arbres [10] : une stratégie peut être modélisée par un arbre infini, et il est possible de construire un automate qui reconnaît exactement les stratégies gagnantes dans le cas des jeux ω -réguliers (i.e. dont la condition de gain est reconnaissable par une formule MSO ou, équivalentement, un automate). Dans cette thèse, nous adoptons la présentation « ludique », qui permet de tirer parti de résultats connus sur les jeux de parité, et qui fournit un formalisme élégant pour décrire les implémentations. Nous suivons globalement l'approche proposée par Thomas dans [5] concernant le problème de Church.

La logique temporelle linéaire (LTL)

Lorsque la spécification est exprimée par une formule MSO, le problème de Church hérite de la borne inférieure de complexité du problème de la satisfaisabilité de ces formules, qui est non-élémentaire [11]. Cette

Dans le cadre de Church, les lettres sont en réalité encodées en binaire, mais il est plus pratique de se doter de prédicats spécifiques.

[6]: McNAUGHTON (1965), *Finite-state infinite games*, cité dans [5].

[7]: McNAUGHTON (1993), « Infinite Games Played on Finite Graphs »

[8]: J.R. BÜCHI et L.H. LANDWEBER (1969), « Solving sequential conditions finite-state strategies »

[9]: BLOEM, CHATTERJEE et JOBSTMANN (2018), « Graph Games and Reactive Synthesis »

[10]: RABIN (1972), *Automata on Infinite Objects and Church's Problem*

[5]: THOMAS (2009), « Facets of Synthesis : Revisiting Church's Problem »

[11]: MEYER (1975), « Weak monadic second order theory of successor is not elementary-recursive »

Un algorithme est élémentaire s'il est dans $k\text{-ExpTime}$ pour un certain $k \in \mathbb{N}$, c'est-à-dire s'il s'exécute en un temps

au plus $2^{\left(\begin{matrix} 2^n \\ k \text{ fois} \end{matrix} \right)}$. À l'inverse, la fonction TOWER, définie, pour $n \in \mathbb{N}$, par $2^{\left(\begin{matrix} 2^n \\ n \text{ fois} \end{matrix} \right)}$, n'est pas élémentaire.

[‡] La modélisation informatique aboutit ainsi à la situation cocasse d'une machine à café en lutte contre ses utilisateurs.

complexité inouïe a longtemps empêché l'application concrète des algorithmes de synthèse. L'introduction de la Logique Temporelle Linéaire (LTL) [12] constitua une première tentative pour l'apprivoiser, dans le contexte voisin de la vérification formelle. Ce problème est le jumeau (plus sympathique) du problème de la synthèse : étant donné une spécification *et* un système, ce dernier satisfait-il la spécification ? A contrario, la synthèse consiste à *construire* le système à partir de la spécification. Pnueli montra qu'un tel problème est PSPACE-complet, puis que le problème de la synthèse est 2-EXPTIME-complet [13]. Ce fossé entre la complexité des deux problèmes est l'une des raisons pour lesquelles la vérification formelle est aujourd'hui appliquée plus largement que la synthèse [14]. Cependant, au cours de la décennie écoulée, l'apparition de méthodes efficaces pour la synthèse a déclenché un renouveau d'intérêt pour le domaine [3, 15, 16], désormais très actif [9, 17–20].

Pour donner une idée du fonctionnement de LTL, signalons que la spécification « *verser_cafe* apparaît après chaque occurrence de *commander_cafe* » (sans borne de temps) peut être exprimée comme $G(\text{commander\_cafe} \Rightarrow F\text{verser\_cafe})$, où G (pour « *globally* ») signifie que la propriété est vraie tout au long de l'exécution, et F (pour « *finally* ») qu'elle sera vraie à un certain moment dans le futur. Ainsi, la formule ci-dessus se lit « Tout au long de l'exécution, il doit être vrai que si l'on rencontre *commander_cafe*, alors on rencontre *verser_cafe* par la suite ».

1.2.4. Les ω -automates

Il est également possible de représenter directement les spécifications comme des ω -automates. En effet, toute spécifications MSO et, a fortiori, LTL, peut être convertie en un ω -automate qui reconnaît les interactions acceptables correspondantes. Dans le cas des mots infinis, on ne peut plus définir l'acceptance par le fait qu'une exécution termine dans un état acceptant, puisque les exécutions sont infinies et ne terminent donc pas. À la place, nous pouvons demander à ce que l'exécution visite un état acceptant infiniment souvent (condition dite « de Büchi »), ou, dualement, que tout état rejetant n'est visité qu'un nombre fini de fois (« co-Büchi »). Des conditions plus générales existent, telles la condition de parité ou de Muller, mais les deux conditions ci-dessus suffisent pour notre propos.

Il est souvent plus aisé de concevoir un automate non-déterministe qui reconnaît les violations de la spécification puis de le compléter (en un automate universel), plutôt que d'élaborer directement un automate pour la spécification. Par exemple, l'automate non-déterministe de la FIGURE 1.5 vérifie qu'il existe une occurrence de *commander_cafe* qui n'est jamais suivie de *verser_cafe*. En le dualisant, nous obtenons un automate universel qui vérifie que toute occurrence de *commander_cafe* est ultérieurement suivie d'une occurrence de *verser_cafe*.

Le Chapitre 3 présente la synthèse réactive et les définitions associées, ainsi que les résultats connus du domaine dont nous aurons besoin pour notre étude.

[12]: PNUELI (1977), « The Temporal Logic of Programs »

Une algorithm est dans 2-EXPTIME s'il s'exécute en un temps polynomial en 2^{2^n} , où n est la taille de l'entrée.

[13]: PNUELI et ROSNER (1989), « On the Synthesis of a Reactive Module »

[14]: FINKBEINER (2016), « Synthesis of Reactive Systems »

[3]: FINKBEINER et SCHEWE (2013), « Bounded synthesis »

[15]: EHLERS (2012), « Symbolic bounded synthesis »

[16]: FILIOT, JIN et RASKIN (2011), « Antichains and compositional algorithms for LTL synthesis »

[9]: BLOEM, CHATTERJEE et JOBSTMANN (2018), « Graph Games and Reactive Synthesis »

[17]: JACOBS et al. (2017), « The first reactive synthesis competition (SYNTCOMP 2014) »

[18]: BONAKDARPOUR et FINKBEINER (2020), « Controller Synthesis for Hyperproperties »

[19]: FINKBEINER, GEIER et PASSING (2021), « Specification Decomposition for Reactive Synthesis »

[20]: ALMAGOR et KUPFERMAN (2020), « Good-Enough Synthesis »

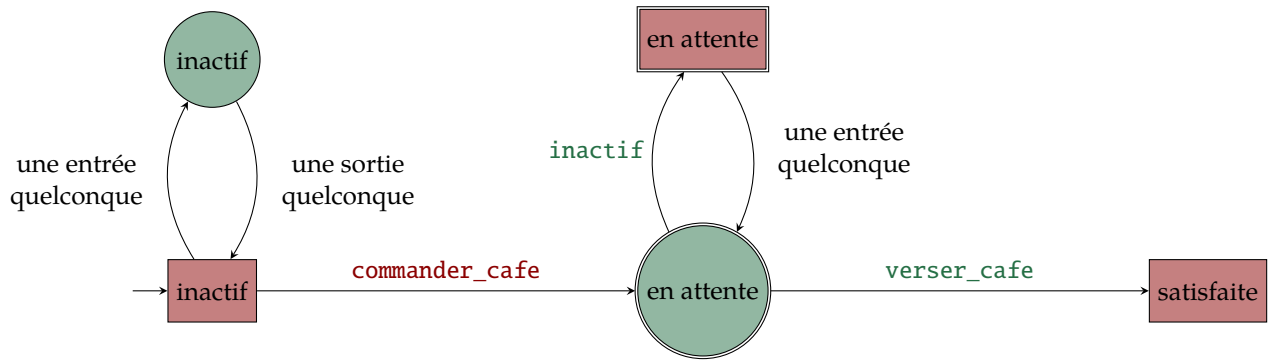


FIGURE 1.5. : Un automate non-déterministe avec une condition de Büchi qui vérifie qu’une certaine occurrence de `commander_cafe` n’est jamais suivie de `verser_cafe` : lorsqu’il est dans l’état « inactif », il devine à un moment donné que l’occurrence de `commander_cafe` qu’il lit n’est jamais suivie de `verser_cafe` et transitionne vers l’état « en attente ». L’exécution accepte si et seulement si elle boucle infiniment dans cet état, ce qui est le cas lorsqu’il n’y a plus aucune occurrence de `verser_cafe`. En le dualisant, on obtient un automate qui vérifie que `commander_cafe` est ultérieurement suivi de `verser_cafe` : « en attente » est désormais le seul état rejetant, et il est visité infiniment souvent par au moins une exécution si et seulement si une demande de café n’est jamais satisfaite.

1.3. « Vers l’infini et au-delà ! » : le passage aux alphabets infinis

1.3.1. Les mots de données

Le renouveau d’intérêt qu’a connu la synthèse réactive a ouvert la voie à l’extension de techniques existantes à des cadres plus généraux. Ainsi, si l’on revient à notre exemple de la machine à café (ou, de manière équivalente, à un serveur chargé de satisfaire des requêtes), il n’est pas toujours possible d’amalgamer indistinctement les requêtes de tous les utilisateurs : chaque utilisateur pourrait avoir une attente spécifique (avoir son nom sur la tasse, avoir son dosage personnalisé, etc), ce qui ne peut pas être modélisé par un alphabet fini. Dans cette thèse, nous proposons de généraliser l’étude aux *mots de données*, dont les lettres prennent leurs valeurs dans un ensemble infini $\mathbb{D}$ de valeurs, que l’on appelle un *domaine de données* (par conséquent, on ne parle plus de « lettres », mais de « données »). Pour faciliter la modélisation, il est pratique d’étiqueter ces données avec des lettres issues d’un alphabet fini Σ , même s’il est techniquement possible de les encoder dans le domaine de données (on parle alors de *données étiquetées*). Par exemple, nous pouvons modéliser le fait qu’un utilisateur donné a commandé un café par la paire (`commander_cafe`, `id`), où `id` est une donnée qui correspond à un identifiant unique (par exemple, un entier naturel). Nous pouvons doter les domaines de données de différentes structures, en fonction de ce que l’on souhaite modéliser. La structure est donnée par un ensemble de prédicats et de constantes, qui sont interprétés dans le domaine associé. La structure la plus élémentaire consiste en un domaine de données doté du seul prédicat d’égalité (« = »), qui permet par exemple de spécifier que l’utilisateur n° i reçoit la tasse n° i . Nous pouvons aussi considérer le cas d’un ensemble linéairement ordonné, pour modéliser le fait que certains utilisateurs sont plus égaux que d’autres et ont par conséquent la priorité sur les autres. La généralité du concept de domaine de données permet également d’envisager le cas où l’on peut effectuer des opérations sur les données, ce qui est le cas si la machine à café reçoit de la monnaie et doit vérifier que la somme due a été payée, auquel cas le modèle doit

pouvoir effectuer des additions, en plus des tests d'égalité. Cependant, nous naviguons désormais en eaux troubles, et le lecteur aguerri pressent probablement que ce type de structure est trop expressif pour préserver la décidabilité (voir la Section 4.9).

1.3.2. Les automates à registres

Il s'agit désormais d'étendre les modèles de systèmes réactifs au cas des mots de données, afin qu'ils soient capables de manipuler des données. Les *automates à registres* constituent l'une des principales extensions des automates aux langages de données [21, 22]. Un automate à registres consiste, comme son nom l'indique, en un automate (tel que défini précédemment), équipé d'un ensemble fini de *registres*, qu'il utilise pour stocker certaines données lues, et pour les comparer entre elles à l'aide des prédicats du domaine. Précisons qu'un registre ne peut contenir qu'une seule donnée à la fois, c'est-à-dire que lorsqu'une donnée est stockée dans un registre, son contenu antérieur est « écrasé ». Ce modèle

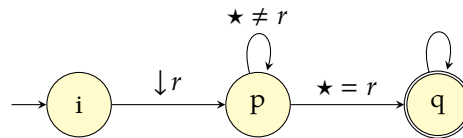


FIGURE 1.6 : Un automate à registres déterministes avec un seul registre r , opérant sur le domaine de données $(\mathbb{N}, =)$. Il vérifie que la première donnée apparaît à nouveau ultérieurement dans le mot d'entrée. Il commence donc par stocker la première donnée dans r ($\downarrow r$), puis attend dans l'état p de rencontrer à nouveau cette donnée (le test $\star = r$ est vérifié lorsque la donnée lue en entrée est égale au contenu de r). Si cela advient, il transitionne vers l'état q , dans lequel il boucle inconditionnellement et accepte.

fut initialement introduit pour les domaines de données avec le seul prédicat d'égalité, puis généralisé aux ordres linéaires [23, 24]. Nous proposons de l'étendre à des domaines arbitraires, dans l'esprit des G-automates [25, Section 3] et de [26, Chapitre 1].

De même que les automates finis, les automates à registres peuvent être dotés de deux sémantiques duales, selon que l'on les interprète comme non-déterministes ou universels. Un automate à registres non-déterministe accepte son entrée si au moins l'une de ses exécutions est acceptante, c'est-à-dire s'il est capable de deviner une exécution correcte. À l'inverse, un automate universel accepte si toutes ses exécutions sont acceptantes. Contrairement au cas des alphabets finis, les deux modèles ne sont pas équivalents, et reconnaissent des classes duales de langages [21, Proposition 5]. Il est également possible de combiner les deux sémantiques via la notion d'*alternance* [27]. Cependant, le modèle résultant est trop expressif pour l'objectif visé, car il est impossible de décider si un automate alternant donné n'accepte aucun mot (problème du vide) [28, Theorem 4.2], pas plus qu'il n'est possible de décider s'il accepte tous les mots (problème de l'universalité), puisque ce modèle est trivialement clos par complément (par dualité). Ceci douche tout espoir d'obtenir la décidabilité pour le problème plus difficile de la synthèse pour les spécifications exprimées par des automates alternants.

[21]: KAMINSKI et FRANCEZ (1994), « Finite-Memory Automata »

[22]: SEGOUFIN (2006), « Automata and Logics for Words and Trees over an Infinite Alphabet »

[23]: BENEDIKT, LEY et PUPPIS (2010), « What You Must Remember When Processing Data Words »

[24]: FIGUEIRA, HOFMAN et LASOTA (2016), « Relating timed and register automata »

[25]: BOJAŃCZYK, KLIN et LASOTA (2014), « Automata theory in nominal sets »

[26]: BOJAŃCZYK (2019), *Atom Book*

[21]: KAMINSKI et FRANCEZ (1994), « Finite-Memory Automata »

[27]: CHANDRA, KOZEN et STOCKMEYER (1981), « Alternation »

[28]: DEMRI et LAZIC (2009), « LTL with the freeze quantifier and register automata »

Voir aussi [29]: FIGUEIRA (2010), « Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données) », *thèse de doctorat*.

Le problème de l'universalité n'a pas de rapport direct avec la sémantique dite universelle (ou co-non-déterministe).

1.3.3. Les transducteurs à registres

L'approche par automates des problèmes de vérification et de synthèse peut être étendue aux mots de données à l'aide de modèles d'automates et de transducteurs qui opèrent sur les données. Dans cette perspective, nous pouvons étendre les transducteurs aux *transducteurs à registres*, qui permettent de modéliser un système réactif manipulant des données. De manière analogue à la généralisation des automates aux automates à registres, un transducteur à registres consiste en un transducteur équipé d'un ensemble fini de registres[§]. Lorsqu'il lit une donnée étiquetée $(\sigma, d) \in \Sigma \times \mathbb{D}$ en entrée, il compare d au contenu de ses registres à l'aide des prédicats du domaine et, en fonction du résultat du test, décide de manière déterministe de stocker d dans certains de ses registres (ou aucun) et produit le contenu d'un de ses registres, accompagné d'une étiquette $\gamma \in \Gamma$ (voir l'exemple de la FIGURE 1.7). Les exécutions d'un

Rappelons que Σ est un alphabet fini d'étiquettes, et $\mathbb{D}$ un ensemble infini de données.

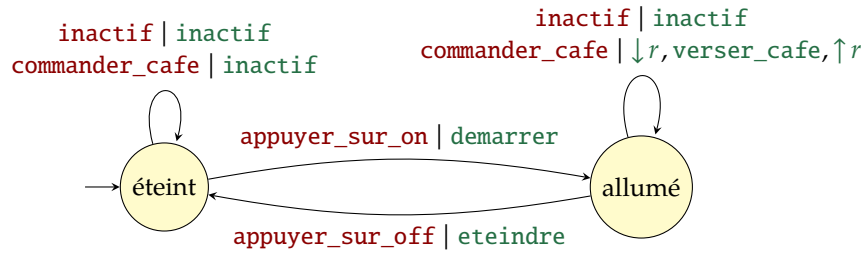


FIGURE 1.7. : Un transducteur à registres séquentiel et synchrone modélisant une machine à café qui sert chaque utilisateur ayant commandé de manière spécifique (e.g., en indiquant son nom).

tel modèle correspondent alors à des mots de données alternant entre une donnée étiquetée d'entrée et une donnée étiquetée de sortie. De même que les spécifications dites « automatiques » sont décrites par des ω -automates qui reconnaissent des suite infinies alternant entre signaux d'entrées et de sorties, l'on peut utiliser les automates à registres pour représenter des spécifications par des langages de données (voir FIGURE 1.8).

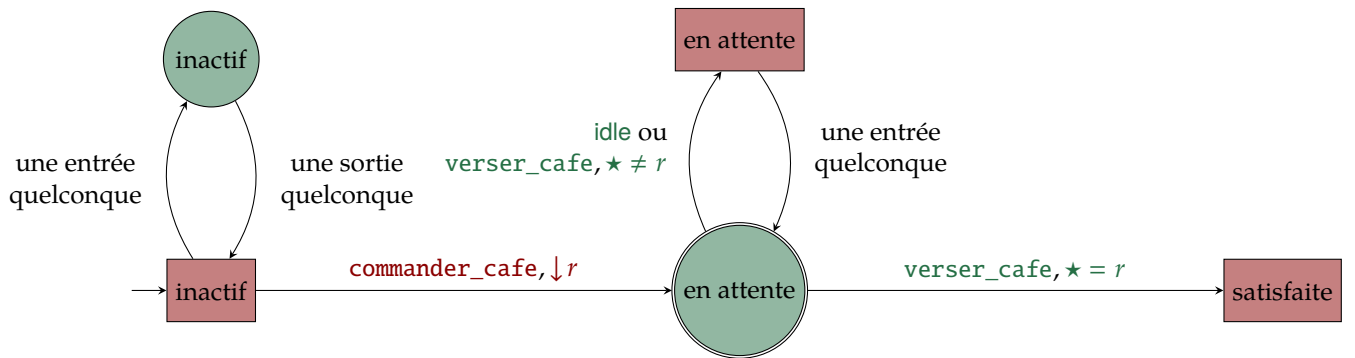


FIGURE 1.8. : Un automate à registres universel de co-Büchi qui vérifie que chaque requête `commander_cafe` de chaque client id finit par être satisfaite par (`verser_cafe, id`).

[§] Étonnant, n'est-ce pas ?

1.3.4. Automates et logiques sur les mots de données

Contrairement au cas des alphabets finis, nous ne connaissons pas à ce jour de correspondance entre automates et logique qui puisse être exploitée pour résoudre le problème de la synthèse pour les spécifications exprimées dans un formalisme logique. Nous ne pouvons donc pas adapter l'approche du problème de Church dans le cas des spécifications MSO, qui consiste à convertir la spécification en un automate, et à résoudre un jeu sur cet automate [30]. Dans [28], les auteurs font le lien entre la logique LTL étendue avec le quantificateur « freeze » [31], qui permet de stocker une valeur apparaissant dans la formule, et les automates à registres. Ils démontrent que les langages reconnus par les formules de cette logique peuvent être reconnus par des automates à registres alternants et bi-directionnels (c'est-à-dire que leur tête de lecture peut se déplacer non seulement de gauche à droite, mais aussi de droite à gauche), dont le nombre de registres est égal au nombre de variables qui peuvent être « congelées » (*frozen*) [28, Theorem 4.1]. Cependant, le vide d'un tel modèle est indécidable, déjà avec un seul registre lorsque l'on considère les mots infinis [28, Theorem 5.2]. Par la suite, Demri, D'Souza et Gascon ont étudié la Logique des Valeurs Récurrentes (*Logic of Repeating Values*, LRV) qui limite l'usage du quantificateur « freeze » afin d'obtenir une logique qui soit à la fois décidable et close par négation [32]. Cependant, les jeux dont la condition de victoire est donnée par une formule LRV sont indécidables, comme établi dans [33, Theorem 4.1]. Le cas des spécifications unilatérales (*single-sided*) est décidable, et a inspiré notre étude du problème de la synthèse pour les spécifications données par des automates à registres déterministes, en particulier l'adaptation de la notion d'unilatéralité à ce modèle. Au vu de ces résultats d'indécidabilité, et faute d'une logique pour les spécifications qui soit suffisamment expressive et permette à la fois au système et à l'environnement de manipuler des données tout en restant décidable, nous proposons d'étudier le cas des spécifications données directement par des automates à registres, que nous appelons *spécifications automatiques avec données*.

[30]: THOMAS (2008), « Church's Problem and a Tour through Automata Theory »

[28]: DEMRI et LAZIC (2009), « LTL with the freeze quantifier and register automata »

[31]: FRENCH (2003), « Quantified Propositional Temporal Logic with Repeating States »

[32]: DEMRI, D'SOUZA et GASCON (2012), « Temporal Logics of Repeating Values »

On dit qu'une classe de jeux est indécidable si le problème de décider le vainqueur d'un jeu donné est indécidable.

[33]: FIGUEIRA, MAJUMDAR et PRAVEEN (2020), « Playing with Repetitions in Data Words Using Energy Games »

1.3.5. Contrôle du système et traitement des données

Doter les automates de registres permet de séparer deux aspects du comportement du système, à savoir le contrôle des états du système et la manière dont il traite les données, ce qui montre la pertinence du modèle y compris dans le cas des alphabets finis. En effet, les algorithmes de synthèse existants ont des difficultés à passer à l'échelle lorsque la taille de l'alphabet augmente. Revenons à notre exemple favori, et supposons que la machine est capable de servir différents types de brevages. Il y en a un nombre fini, pour autant ce qui importe n'est pas tant la nature précise de tel ou tel breverage, que le fait que si un utilisateur demande le breverage x , il reçoit le breverage x . Ce caractère « paramétrique » ne peut pas être modélisé par un alphabet fini sans structure additionnelle. Nous pouvons l'illustrer de la manière suivante : supposons que les utilisateurs choisissent parmi un ensemble B , et que k d'entre eux commandent en même temps. Dans ce cas, la machine doit garder en mémoire l'ensemble des brevages commandés. N'importe quel sous-ensemble de taille k est possible, ce qui signifie que la mémoire requise est de taille au moins $\binom{|B|}{k}$.

$\binom{n}{k}$ compte le nombre de choix possibles de k valeurs parmi n valeurs, et croît exponentiellement en fonction de n .

À l'inverse, nous pouvons représenter ce cas de figure par un automate à k registres, qui stocke la commande de chaque client dans le registre correspondant, et vérifie par la suite que la commande a été correctement servie. On retrouve ce phénomène du côté de l'implémentation : sans registres, un transducteur a besoin d'une mémoire $\binom{|B|}{k}$, tandis qu'un transducteur à registres a seulement besoin de k registres (et de k états mémoire).

Cette direction de recherche a été spécifiquement explorée dans le cas des spécifications exprimées dans des fragments de la logique du premier ordre sur les mots de données, sous le nom de « synthèse paramétrée » (*parameterised synthesis*). Ce cadre repose sur l'hypothèse que le système interagit avec un nombre fixé, mais arbitraire, d'éléments distincts [34].

[34]: BÉRARD et al. (2020), « Parameterized Synthesis for Fragments of First-Order Logic Over Data Words »

1.3.6. Contributions

Les contributions suivantes sont détaillées dans la première partie du manuscrit. Le Chapitre 4 présente le modèle des automates à registres, qui est au cœur de notre étude, en ce qu'il constitue le formalisme de spécification. Y sont décrites les propriétés utiles pour notre étude, tirées de la littérature existante et généralisées aux domaines de données arbitraires lorsque c'est possible. Nous proposons ensuite une déclinaison du problème de la synthèse de Church aux spécifications sur les mots de données, qui cible des implémentations représentées par des transducteurs à registres, en tant qu'homologue des transducteurs dans le cas des mots de données (Chapitre 5).

La synthèse à registres bornés

Dans notre première contribution (Chapitre 6), nous considérons la synthèse à registres bornés (*register-bounded synthesis* [35]), qui consiste à cibler une implémentation définie par un transducteur à registres dont le nombre de registres est donné en entrée. Nous commençons par les spécifications données par des automates à registres universels. Comme expliqué précédemment, la sémantique universelle est particulièrement adaptée pour exprimer des spécifications : elle est naturellement close par intersection, et permet de spécifier une violation de la spécification à l'aide d'un automate non-déterministe, puis de compléter en un automate universel. Par exemple, demander à ce qu'un utilisateur finisse par recevoir le café qu'il a demandé (ou, dans le cadre client/serveur, que toute requête finisse par être satisfaite), peut être exprimé par un automate universel qui vérifie qu'aucune commande ne se retrouve insatisfaite (FIGURE 1.8). Le fait qu'une commande reste insatisfaite s'exprime aisément par un automate non-déterministe, puisqu'il suffit de deviner ladite commande ; la spécification s'obtient alors en dualisant l'automate pour obtenir la négation. Notons d'ailleurs que dans le cas d'un alphabet infini, une telle spécification n'est pas exprimable par un automate non-déterministe, seulement par un universel.

[35]: KHALIMOV et KUPFERMAN (2019), « Register-Bounded Synthesis »

Nous montrons que la synthèse à registres bornés est décidable pour les spécifications données par des automates à registres universels sur les domaines de données $(\mathbb{D}, =)$ (Théorème 6.14) et $(\mathbb{Q}, <)$ (Théorème 6.16). Plus précisément, ces problèmes sont dans 2-EXPTIME. Le premier résultat

est connu [36, Corollary 3]; nous l'avons prouvé indépendamment dans [37, Theorem 12]. La modularité de notre preuve permet d'étendre le résultat au cas de $(\mathbb{Q}, <)$ avec peu de travail supplémentaire, tout en autorisant les automates de spécification à stocker dans leurs registres des données choisies arbitrairement, et de lancer une exécution pour chaque choix possible (*guessing*). Il s'agit du mécanisme dual à la « réassignation non-déterministe » (*non-deterministic reassignment*) introduite dans [38]. Le cas de $(\mathbb{N}, <)$ est plus complexe, car la structure de $(\mathbb{N}, <)$ induit des comportements que l'on ne peut abstraire de manière régulière (c'est-à-dire avec des ω -automates). En effet, les exécutions des automates à registres opérant sur ce domaine correspondent aux langages ωB -réguliers [39, Theorem 17], qui étendent les langages ω -réguliers. La synthèse bornée pour ce domaine fait présentement l'objet de recherches, qui ne sont pas décrites dans ce manuscrit car elles exigent un temps de maturation supplémentaire.

Nos algorithmes reposent sur une méthode générique qui consiste en une réduction à la synthèse réactive sur les alphabets infinis. Elle permet d'obtenir la décidabilité lorsque le formalisme de spécification présente certaines propriétés de clôture, et permet certaines opérations sur les langages. Le théorème de transfert (Théorème 6.8) en compose la clé de voûte : il établit qu'une spécification de mots de données est réalisable si et seulement si son homologue sur les alphabets finis l'est, cette dernière étant définie à partir de la notion de *suite d'actions*, qui représente les exécutions de transducteurs à registres vues de manière syntaxique. La notion de mots de données *compatibles* permet alors de faire le pont avec la sémantique.

Les spécifications données par des automates non-déterministes ne se laissent pas si aisément domestiquer : déjà avec un seul registre pour l'implémentation, le problème est indécidable (6.19). Pour cette raison, nous étudions le cas des spécifications *sans tests*, où les transitions des automates de spécification ne peuvent pas dépendre de tests effectués sur l'entrée. Ces machines restent capables de dupliquer, effacer et déplacer des valeurs de données. La méthode générique décrite ci-dessus permet d'établir que la synthèse à registres bornés est décidable et dans 2-ExpTime pour cette classe de spécifications. La preuve repose sur la notion d'origine, telle qu'introduite dans [40].

Une version préliminaire de ce travail est parue dans [37], puis étendue dans [41]. Le Chapitre 6 traite de plus de $(\mathbb{Q}, <)$ et de l'extension au cas du *guessing*. Les preuves ont également été reformulées dans le cadre plus général de la synthèse à partir de templates (*template-guided synthesis*).

Les Tables 1.1 (pour $(\mathbb{D}, =)$) et 1.2 (pour $(\mathbb{Q}, <)$) récapitulent les résultats obtenus, de même que ceux de la contribution suivante.

La synthèse non-bornée

Le Chapitre 7 est dédié au problème de la synthèse dans toute sa généralité, que nous appelons « non-bornée » pour marquer la distinction avec la synthèse à registres bornés. L'implémentation cible peut être n'importe quel transducteur à registres, c'est-à-dire qu'il n'y a pas de bornes sur son nombre de registres. Nous montrons que le problème est indécidable pour les spécifications exprimées par des automates non-déterministes

[36]: KHALIMOV, MADERBACHER et BLOEM (2018), « Bounded Synthesis of Register Transducers »

[37]: EXIBARD, FILIOT et REYNIER (2019), « Synthesis of Data Word Transducers »

[38]: KAMINSKI et ZEITLIN (2010), « Finite-Memory Automata with Non-Deterministic Reassignment »

[39]: SEGOUFIN et TORUNCZYK (2011), « Automata based verification over linearly ordered data domains »

[40]: BOJANCZYK (2014), « Transducers with Origin Information »

[37]: LÉO EXIBARD, Emmanuel FILIOT et Pierre-Alain REYNIER. « Synthesis of Data Word Transducers ». In : *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Sous la dir. de Wan FOKKINK et Rob van GLABBEK. T. 140. LI-PIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019

[41]: LÉO EXIBARD, Emmanuel FILIOT et Pierre-Alain REYNIER. « Synthesis of Data Word Transducers ». In : *Logical Methods in Computer Science* 17.1 (2021)

et universels, déjà dans le cas de $(\mathbb{D}, =)$ (Théorèmes 6.20 et 7.1), ce qui motive rétrospectivement l'étude de la synthèse à registres bornés. Le cas des automates universels était posé comme un problème ouvert dans [36]. Pour récupérer la décidabilité, nous examinons le cas des spécifications données par des automates à registres déterministes, où la notion de déterminisme est entendue une fois l'entrée *et* la sortie données (par opposition au déterminisme des transducteurs dits séquentiels, qui se comportent de manière déterministe à partir de leur seule entrée). Nous montrons que si la spécification est donnée par un automate *localement concrétisable* (c'est-à-dire que depuis n'importe quelle configuration, toute transition peut être prise, autrement dit il est toujours possible d'instancier les tests avec une donnée, indépendamment de la configuration concrète), alors il suffit de résoudre un jeu sur l'automate, comme dans le cas des alphabets finis. Ce résultat permet d'établir la décidabilité du problème de la synthèse réactive et du problème de la synthèse de Church dans le cas de $(\mathbb{D}, =)$ (Théorème 7.15) et de $(\mathbb{Q}, <)$ (Théorèmes 7.16 et 7.17). Dans le cas de $(\mathbb{D}, =)$, nous montrons de plus que les deux problèmes (synthèse réactive et synthèse de Church) sont équivalents, un résultat analogue au cas des alphabets finis. Autrement dit, si une spécification admet une implémentation, elle admet une implémentation représentable par un transducteur à registres. Notons que ce n'est pas le cas dans $(\mathbb{Q}, <)$ (Propriété 7.18).

De ce fait, nous étudions également le cas unilatéral, analogue à la restriction *single-sided* introduite pour les jeux LRV qui permet de récupérer la décidabilité dans [33]. Ainsi, les transitions de sortie des automates de spécification sont restreintes aux valeurs de données déjà présentes dans les registres. De manière équivalente, il s'agit de restreindre à un ensemble fini de données, puisque choisir un registre dont le contenu doit être produit en sortie est équivalent à choisir une lettre qui désigne ledit registre. En d'autres termes, de telles spécifications ont un comportement similaire à des transducteurs (non séquentiels), et résoudre le problème de la synthèse se ramène à choisir des transitions le long d'une exécution. Formulé dans le cadre de la théorie algorithmique des jeux, cela revient à demander à ce que le joueur « Système » choisisse ses actions dans un ensemble booléen. Pour cette sous-classe, nous obtenons à nouveau l'équivalence entre synthèse réactive et synthèse de Church, c'est-à-dire qu'il suffit de cibler des implémentations représentables par des transducteurs à registres, y compris dans le cas de $(\mathbb{Q}, <)$. Enfin, nous montrons que cette restriction marque la frontière entre décidabilité et indécidabilité dans le cas des entiers naturels avec l'ordre $(\mathbb{N}, <)$ (Théorème 7.28 versus 7.19). La décidabilité dans le cas des spécifications unilatérales sur $(\mathbb{N}, <)$ est obtenue en montrant qu'il suffit de considérer une approximation ω -régulière du jeu sur l'automate (rappelons que les comportements des automates à registres sur $(\mathbb{N}, <)$ ne sont en général pas ω -réguliers). L'indécidabilité provient du fait que l'alternance et l'antagonisme entre les deux joueurs permet de simuler des machines à compteurs : un joueur propose les valeurs successives du compteur, tandis que l'autre vérifie qu'il ne triche pas.

Ces travaux ont également été présentés dans [37] et [41]. Dans ce manuscrit, l'argumentaire est présenté de manière plus modulaire, à l'aide de la notion d'automate de spécification « prêt pour les jeux » (*game-ready*, à ne pas confondre avec « bon pour les jeux », *good-for-games*).

[36]: KHALIMOV, MADERBACHER et BLOEM (2018), « Bounded Synthesis of Register Transducers »

Rappelons la différence entre synthèse réactive et synthèse de Church : la première cible des implémentations données par des programmes synchrones, qui peuvent disposer d'une mémoire arbitraire, tandis que la seconde cible des transducteurs, c'est-à-dire des machines à états *finis*.

[33]: FIGUEIRA, MAJUMDAR et PRAVEEN (2020), « Playing with Repetitions in Data Words Using Energy Games »

On parle de *transition de sortie* pour désigner les transitions qui lisent des lettres de sortie. Rappelons qu'un couple (entrée, sortie) est représenté par l'entrelacement de l'entrée et de la sortie.

[37]: EXIBARD, FILIOT et REYNIER (2019), « Synthesis of Data Word Transducers »

[41]: EXIBARD, FILIOT et REYNIER (2021), « Synthesis of Data Word Transducers »

Par ailleurs, nous étendons l'étude au cas de $(\mathbb{Q}, <)$.

Le cas de $(\mathbb{N}, <)$ a été traité séparément dans [42], car il requiert l'introduction de techniques de preuves supplémentaires.

[42]: EXIBARD, FILIOT et KHALIMOV (2021), « Church Synthesis on Register Automata over Linearly Ordered Data Domains »

TABLE 1.1 : Décidabilité et complexité des problèmes étudiés dans le cas de $(\mathbb{D}, =, \mathbb{C})$. $\{D, N, U\}RA$ est une abréviation pour « automates à registres déterministes » (respectivement, « non-déterministes », « universels »); la mention « tf » signifie « sans tests ». La « synthèse bornée » désigne la synthèse à registres bornés. Comme remarqué dans le Corollaire 7.13, la synthèse à registres bornés pour les automates à registres déterministes est dans $ExpTime$ dès que le nombre de registres cible dépasse le nombre de registres de la spécification, car le problème se ramène alors à la synthèse non-bornée.

	DRA	NRA	URA	NRA ^{tf}
Synthèse bornée	$2ExpTime$ (Thm. 6.14)	Indécidable ($k \geq 1$) (Thm. 6.19)	$2ExpTime$ ([36] et Thm. 6.14)	$2ExpTime$ (Thm. 6.25)
Synthèse non-bornée	$ExpTime$ -complet (Thm. 7.15)	Indécidable (Thm. 6.20)	Indécidable (Thm. 7.1)	Ouvert

TABLE 1.2 : Décidabilité et complexité des problèmes étudiés dans le cas de $(\mathbb{Q}, <, 0)$. Le paysage est très similaire au cas de $(\mathbb{D}, =, \mathbb{C})$, il s'agit surtout de pointer vers les théorèmes correspondants. Nous omettons les spécifications « sans tests », car elles ne dépendent pas du domaine de données choisi.

	DRA	NRA	URA
Synthèse bornée	$2ExpTime$ (Thm. 6.16)	Indécidable ($k \geq 1$) (Thm. 6.19)	$2ExpTime$ (Thm. 6.16)
Synthèse non-bornée	$ExpTime$ -complet (Thm. 7.16)	Indécidable (Thm. 6.20)	Indécidable (Thm. 7.1)

1.3.7. Travaux connexes

Comme mentionné précédemment, la synthèse à registres bornés a d'abord été étudiée dans [36], qui établit la décidabilité dans le cas des automates à registres universels. Nous avons prouvé le résultat principal [36, Corollary 3] de manière indépendante. L'étude du problème est approfondies dans [35], qui contient des preuves plus simples et une analyse de complexité plus fine, ainsi que l'étude d'un cadre où l'environnement est modélisé comme un transducteur avec un nombre borné de registres. Les auteurs démontrent que ce cas est décidable, bien que le jeu correspondant ne soit pas déterminé.

[36]: KHALIMOV, MADERBACHER et BLOEM (2018), « Bounded Synthesis of Register Transducers »

[35]: KHALIMOV et KUPFERMAN (2019), « Register-Bounded Synthesis »

Dans [43], Faran et Kupferman étudient le problème de la synthèse pour les automates avec arithmétique (*word automata with arithmetic*). Ces automates sont dotés d'un nombre fini de variables sur lesquelles ils peuvent (comme le nom l'indique) effectuer des opérations arithmétiques et des tests. La notion de variable est analogue à notre notion de registres; la différence majeure est que la valuation des variables est fixée pour l'entière de l'exécution, tandis que les registres peuvent être mis à jour en remplaçant leur contenu par la donnée lue en entrée. Les autrices établissent la décidabilité du problème de la synthèse pour les automates avec arithmétique non-déterministes qui sont « sémantiquement déterministes » (*semantically deterministic*), c'est-à-dire dont le non-déterminisme peut-être résolu une fois la valuation des variables connue. Plus précisément, elles fournissent un algorithme polynomial en la taille de l'automate et exponentiel en le nombre de variables [43, Theorem 5]. Elles démontrent également que les automates sémantiquement déterministes se situent strictement entre les automates déterministes et non-déterministes (avec arithmétique) [43, Theorem 2 et Theorem 3].

Un jeu est dit *déterminé* si l'un des deux joueurs a une stratégie gagnante. C'est le cas par exemple des échecs ou du go, en assimilant les parties nulles à une victoire pour l'un des deux joueurs (de telles stratégies ne sont cependant pas calculables avec les connaissances et la puissance de calcul disponibles à l'heure actuelle). Ce n'est en revanche généralement pas le cas des jeux à information imparfaite comme le poker ou les jeux de dés, ni des jeux simultanés comme « pierre-feuille-ciseaux ».

[43]: FARAN et KUPFERMAN (2020), « On Synthesis of Specifications with Arithmetic »

Le problème de la synthèse dans le cas des alphabets infinis est également considéré dans [44]. Les données représentent alors des identifiants uniques, et les spécifications sont exprimées dans un formalisme équivalent aux automates à registres universels avec le prédicat d'égalité. Le formalisme d'implémentation est, quant à lui, beaucoup plus expressif, puisqu'il autorise une mémoire infinie : les données sont stockées dans une file (*queue*). Les auteurs démontrent que le problème de la synthèse est indécidable, y compris pour les spécifications exprimées par des automates semi 1-faibles (*semi one-weak*). Ils fournissent un algorithme correct mais incomplet, et en démontrent l'utilité à travers une application à un robot serveur dans un restaurant.

La classe des automates non-déterministes sans tests, qui permet d'obtenir la décidabilité de la synthèse à registres bornés pour une classe d'automates non-déterministes, correspond à la version uni-directionnelle et non-déterministe du modèle de transducteurs défini par [45]. Le problème de la synthèse n'est pas considéré dans cet article.

Dans [46], les auteurs présentent une approche de « perfectionnement d'abstraction guidé par contre-exemples » (*Counter-Example Guided Abstraction Refinement*, CEGAR) pour les spécifications exprimées dans la logique temporelle des flux (*Temporal Stream Logic*, TSL), qui permet de définir des spécifications via des fonctions non-interprétées. L'algorithme essaie de générer une implémentation en autorisant un usage incohérent des fonctions. S'il échoue, il calcule un contre-exemple. Si ce dernier est erroné (c'est-à-dire incohérent du point de vue de l'usage des fonctions), il raffine son abstraction et itère le processus. À l'inverse, si le contre-exemple est cohérent, l'algorithme s'arrête et renvoie ce contre-exemple, qui témoigne de l'infaisabilité de la spécification. Cela fournit un semi-algorithme pour la synthèse à partir de spécifications TSL ; les auteurs montrent par ailleurs que ce problème est indécidable [46, Theorem 2]. L'article se conclut par une application à la synthèse d'un lecteur audio fonctionnant sous Android.

La synthèse guidée par la syntaxe (*syntax-guided synthesis*) pour des programmes manipulant des données a été étudiée dans [47]. L'algorithme de synthèse prend en entrée un squelette du programme à synthétiser, qu'il doit compléter. Comme précédemment, les fonctions sont non-interprétées, c'est-à-dire que le programme doit être correct indépendamment de l'interprétation choisie. Les auteurs démontrent que le problème est 2-ExpTIME-complet [47, Theorem 9] pour la classe des programmes non-interprétés *cohérents* (une restriction connue pour permettre de rétablir la décidabilité dans le cas de la vérification formelle), tandis que le problème général est indécidable [47, Theorem 1].

Enfin, nous avons déjà mentionné que les jeux avec données ont été étudiés dans [33]. La condition de victoire est donnée par une formule de la logique des valeurs récurrentes (*Logic of Repeating Values*, LRV), qui consiste en une restriction de la logique LTL avec le quantificateur « freeze ». Les auteurs démontrent que de tels jeux sont indécidables, mais que, lorsque la condition de gain est unilatérale (*single-sided*), décider le vainqueur est dans 4-ExpTIME, lorsque les répétitions distantes ne peuvent être testées que vis-à-vis du passé, et non du futur. Les deux restrictions sont simultanément nécessaires pour conjurer l'indécidabilité. Dans une perspective de synthèse, ce résultat peut être reformulé

[44]: EHLERS, SESHIA et KRESS-GAZIT (2014), « Synthesis with Identifiers »

[45]: DURAND-GASELIN et HABERMEHL (2016), « Regular Transformations of Data Words Through Origin Information »

[46]: FINKBEINER et al. (2019), « Temporal Stream Logic : Synthesis Beyond the Bools »

[47]: KROGMEIER et al. (2020), « Decidable Synthesis of Programs with Uninterpreted Functions »

[33]: FIGUEIRA, MAJUMDAR et PRAVEEN (2020), « Playing with Repetitions in Data Words Using Energy Games »

La notion de répétition distante (*remote repetition*) est similaire à l'opérateur F de LTL : il est demandé qu'une répétition advienne au bout d'un moment, mais sans borne de temps, à la différence de X^j , qui demande à ce qu'un événement advienne dans exactement (ou au plus) j unités de temps, pour un j fixé.

comme la décidabilité de la synthèse réactive pour les spécifications LRV unilatérales. La synthèse de Church reste une question ouverte, puisque la structure de la mémoire des stratégies obtenues n'est pas examinée.

1.4. « Tout vient à point à qui sait attendre » : abandonner le réquisit de synchronicité

1.4.1. Implémentations ω -calculables

Dans certains cas de figure, nous ne pouvons pas faire l'hypothèse que le système réagit de manière synchrone à ses signaux d'entrée, car cela serait trop restrictif pour les implémentations. Revenons une fois de plus à notre chère machine à café, et considérons une machine qui peut préparer et servir deux cafés à la fois. Ainsi, lorsqu'une machine reçoit une commande, il est potentiellement préférable d'attendre quelques unités de temps (disons t) l'arrivée d'une seconde commande avant de commencer l'opération. En conséquence, les spécifications et les implémentations sont modélisées par des machines asynchrones. Plus précisément, la spécification est désormais donnée par un transducteur non-déterministe asynchrone, qui est autorisé à produire un mot (possiblement vide) à chaque lettre lue, et les implémentations consistent en des transducteurs *séquentiels* asynchrones, c'est-à-dire des transducteurs qui se comportent de manière déterministe au regard de leur mot d'entrée. Déjà dans le cas d'un alphabet fini, le problème de la synthèse devient indécidable pour un tel attelage de spécifications et d'implémentations [48, Theorem 17]. Dans le cas des alphabets infinis, nous avons établi dans notre étude de la synthèse réactive qu'un tel problème est indécidable dans le cas synchrone (Theorem 6.20), il l'est donc a fortiori dans le cas asynchrone.

[48]: CARAYOL et LÖDING (2015), « Uniformization in Automata Theory »

En lieu et place des transducteurs, nous étendons la classe des implémentations à toutes les fonctions ω -calculables, c'est-à-dire des fonctions f calculables par une machine de Turing déterministe qui prend en entrée un mot infini $x \in \text{dom}(f)$, et doit produire des préfixes de plus en plus longs de la sortie $f(x)$ lorsqu'elle lit des préfixes de plus en plus longs de l'entrée x . Ainsi, une telle machine produit asymptotiquement $f(x)$. Dans l'exemple ci-dessus, la spécification est ω -calculable, puisqu'il suffit d'attendre t unités de temps pour décider s'il faut commencer à préparer un seul café ou deux à la fois. Cependant, examinons la situation où la machine doit plutôt décider automatiquement si elle doit s'éteindre ou non, et n'est autorisée à le faire que si aucune commande n'advient dans le futur. Une telle spécification n'est pas ω -calculable, puisque la décision de s'éteindre ou non dépend de l'entière du suffixe infini des actions restantes. Enfin, pour distinguer les implémentations ω -calculables de celles qui ont une mémoire finie, examinons le cas où une machine accumule des commandes, et ne les satisfait qu'une fois que l'on appuie sur le bouton « envoyer ». Cette spécification est ω -calculable par une machine qui accumule les commandes en mémoire, mais pas par un transducteur séquentiel, puisqu'une telle machine a nécessairement une quantité bornée de mémoire.

1.4.2. Restriction aux spécifications fonctionnelles

Cependant, déjà dans le cas d'un alphabet fini, nous ne savons pas s'il est possible de décider si une spécification donnée par un transducteur non-déterministe *synchrone* est réalisable par une fonction ω -calculable. Notons que lorsque l'on évacue la question du domaine, c'est-à-dire que l'on cible des implémentations qui acceptent n'importe quelle entrée, il a été établi que le problème est décidable [49, Corollary 15]. Dans le cas asynchrone, il est aisé d'adapter la preuve d'indécidabilité de [48, Theorem 17] pour démontrer l'impossibilité de décider la réalisabilité de spécifications données par des transducteurs non-déterministes asynchrones par des fonctions ω -calculables.

[49]: HOLTSMANN, KAISER et THOMAS (2012), « Degrees of Lookahead in Regular Infinite Games »

En toute généralité, une spécification est une relation entre des entrées et des sorties. Lorsqu'il s'agit d'une fonction (c'est-à-dire lorsque toute entrée est en relation avec *au plus* une sortie), nous disons que la spécification est *fonctionnelle*. Au vu des résultats négatifs mentionnés ci-dessus à propos de la synthèse de fonctions ω -calculables à partir de spécifications non-fonctionnelles, nous concentrons notre attention sur le cas des spécifications fonctionnelles et traitons la question suivante : étant donnée la spécification d'une fonction de mots de données infinis, cette fonction est-elle « implémentable », où nous entendons par implémentable le fait d'être ω -calculable par une machine de Turing. De plus, dans le cas où la fonction est implémentable, nous aimerions disposer d'un algorithme qui construit une implémentation. Cela soulève une question d'importance : est-il possible de décider si une spécification donnée est fonctionnelle ? Nous étudions ces questions dans le cas des spécifications données par des transducteurs à registres non-déterministes asynchrones, que nous appellerons simplement transducteurs à registres par la suite. L'asynchronicité accroît considérablement le pouvoir expressif, mais induit dans le même temps des difficultés techniques supplémentaires.

1.4.3. Contributions

De même que dans le cas des automates à registres, les transducteurs à registres sont paramétrés par le domaine de données utilisé pour construire leurs mots d'entrée, et par les prédicats utilisés pour tester les données. Nous exhibons une vaste classe de domaines de données pour laquelle nous établissons la décidabilité des problèmes étudiés (rappelés plus bas), à savoir les domaines de données *oligomorphes* [26, Section 3.2]. En quelques mots, un domaine de données $(\mathbb{D}, \mathfrak{R}, \mathfrak{C})$ est *oligomorphe* lorsque pour tout entier naturel $n \in \mathbb{N}$, $\mathbb{D}^n$ peut être partitionné en un nombre *fini* de classes d'équivalences, où deux n -uplets sont équivalents s'ils sont l'image l'un de l'autre par un automorphisme du domaine (c'est-à-dire une bijection qui préserve les prédicats $\mathfrak{R}$ et les constantes $\mathfrak{C}$).

[26]: BOJAŃCZYK (2019), *Atom Book*

Rappelons qu'un domaine de données consiste en un ensemble dénombrable de données, doté d'un ensemble fini de prédicats et de constantes interprétées sur le domaine.

Par exemple, tout ensemble avec le seul prédicat d'égalité est oligomorphe, à l'image de $(\mathbb{N}, =)$. En effet, un n -uplet est caractérisé par l'égalité ou la différence de ses composantes. Ainsi, $(1, 2)$ est équivalent à $(3, 5)$, car $1 \neq 2$ et $3 \neq 5$, et l'on peut par conséquent construire un morphisme μ qui envoie l'un sur l'autre, en permutant d'une part 1 et 3 et d'autre part 2 et 5 : $\mu(1) = 3$, $\mu(3) = 1$, $\mu(2) = 5$, $\mu(5) = 2$ et $\mu(n) = n$ pour tout $n \neq 1, 2, 3, 5$. De même, $(1, 2)$ est équivalent à $(2, 1)$; à l'inverse, il est

distinct de $(1, 1)$ ou de $(0, 0)$, qui sont caractérisés par l'égalité de leurs deux composantes. Un autre exemple d'importance est celui de $(\mathbb{Q}, <)$, qui est également oligomorphe. Les automorphismes de $(\mathbb{Q}, <)$ sont précisément les bijections croissantes, et les n -uplets sont caractérisés par l'ordre relatif de leurs composantes. Ainsi, dans $(\mathbb{Q}, <)$, on a toujours que $(1, 2)$ est équivalent à $(3, 5)$ (puisque $1 < 2$ et $3 < 5$), mais il n'est plus équivalent à $(2, 1)$ (puisque $1 < 2$ mais $2 \not< 1$). Crucialement, $(\mathbb{N}, <)$ n'est *pas* oligomorphe. En effet, la seule bijection de $\mathbb{N}$ qui préserve $<$ est l'identité, et tous les tuples sont distincts. De même, $(\mathbb{Z}, <)$ n'est pas oligomorphe, car ses automorphismes sont les translations $n \mapsto n + k$ pour un $k \in \mathbb{Z}$ fixé. Les tuples sont alors caractérisés par les distances entre leurs composantes. Nos contributions sont les suivantes :

1. Nous commençons par établir l'équivalence entre ω -calculabilité et continuité (au sens usuel du terme) pour la distance de Cantor, pour les fonctions sur les mots infinis de données. Rappelons que la distance de Cantor entre deux mots est inversement proportionnelle au plus long préfixe commun de ces mots. Formellement, $d(u, v) = 0$ si $u = v$ et $d(u, v) = \frac{1}{|u \wedge v|}$, où $u \wedge v$ est le plus long mot w tel que $w \leq u$ et $w \leq v$ et $|w|$ désigne la longueur de w . Cette équivalence est vérifiée à condition que le problème de la prochaine lettre soit décidable. Ce problème consiste à déterminer (si elle existe) la prochaine valeur de donnée qui peut être produite indépendamment de la suite du calcul, c'est-à-dire en ne connaissant qu'un préfixe fini fixé du mot d'entrée. Nous établissons également des équivalences analogues pour des notions plus fortes d' ω -calculabilité et de continuité, à savoir la continuité de Cauchy, la continuité uniforme et la m -continuité, qui admettent chacune un équivalent naturel en termes d' ω -calculabilité. Dans chaque cas, la construction est effective, c'est-à-dire qu'il est algorithmiquement possible de construire une machine de Turing qui ω -calcule la fonction.
2. Nous définissons une condition générique de calculabilité pour les domaines de données oligomorphes, à savoir la décidabilité du problème de satisfaisabilité de la logique du premier ordre sur le domaine en question [26]. De tels domaines sont appelés *décidables*. Nous montrons alors que le problème de la prochaine lettre est décidable (et la prochaine lettre calculable) pour les fonctions définies par des transducteurs à registres non-déterministes sur les domaines de données oligomorphes et décidables, ainsi que sur $(\mathbb{N}, <)$. Par conséquent (Théorèmes 12.39 et 12.66), nous obtenons que de telles fonctions sont ω -calculables si et seulement elles sont continues, et de même pour les notions dérivées (continuité de Cauchy, continuité uniforme et m -continuité). Cette caractérisation de l' ω -calculabilité nous permet d'établir le résultat principal de notre étude, à savoir la décidabilité de ces notions (cf *infra*).
3. Comme expliqué précédemment, un transducteur à registres ne définit pas nécessairement une fonction, mais une relation, du fait du non-déterminisme. La fonctionnalité est une notion sémantique, et non syntaxique. Pour autant, nous démontrons que le problème de la fonctionnalité est décidable pour les domaines oligomorphes et décidables (Théorème 12.40; il est de plus PSPACE-complet pour les domaines *polynomialement décidables*), ainsi que pour $(\mathbb{N}, <)$ (et à nouveau, PSPACE-complet, Théorème 12.64). Un tel problème constitue un pré-requis pour notre étude, puisque nous supposons

Dans le corps du document, nous définissons cette distance par $d(u, v) = 0$ si $u = v$ et $d(u, v) = 2^{-|u \wedge v|}$; les deux définitions induisent la même topologie et la même notion de continuité.

La notation $w \leq u$ signifie « w est un préfixe de u », c'est-à-dire $u = w \cdot x$ pour un certain mot x .

[26]: BOJAŃCZYK (2019), *Atom Book*

Un domaine oligomorphe est *polynomialement décidable* si la satisfaisabilité de la logique du premier ordre sur sa structure est décidable en utilisant un espace polynomial (et donc un temps au plus exponentiel).

que les spécifications étudiées sont fonctionnelles.

4. Enfin, nous démontrons (Théorème 12.41) que la continuité de fonctions définies par un transducteur à registres non-déterministe sur un domaine oligomorphe décidable (respectivement, polynomialement décidable) est décidable (respectivement, PSPACE-complète). Nous obtenons également la complétude pour PSPACE dans le cas de $(\mathbb{N}, <)$, qui n'est pas oligomorphe, y compris en présence de *guessing*, bien que cela soulève des difficultés supplémentaires (Théorème 12.72). Ces résultats restent vrais pour la notion plus forte de continuité uniforme (Théorème 12.68). Puisque la continuité est équivalente à l' ω -calculabilité pour ces fonctions, il en résulte que l' ω -calculabilité est décidable pour les fonctions définies par des transducteurs non-déterministes à registres opérant sur des domaines de données oligomorphes décidables, et PSPACE-complet lorsqu'ils sont de plus polynomialement décidables. De tels résultats sont également vrais pour la continuité uniforme, et pour $(\mathbb{N}, <)$. Ils constituent notre contribution principale dans cette partie de la thèse, et forment une réponse positive à notre motivation initiale, à savoir la synthèse de programmes.

1.4.4. Travaux connexes

Nous avons mentionné précédemment les travaux liés au problème de la synthèse. La notion de continuité pour la distance de Cantor n'est pas nouvelle, et sa décidabilité était déjà connue pour les fonctions rationnelles sur les alphabets *finis* [50, Proposition 4]. L'approche de Prieur consiste à réduire la continuité à la fonctionnalité en définissant, à partir d'un transducteur T , un transducteur qui reconnaît sa clôture topologique. Ce dernier est fonctionnel si et seulement si le transducteur initial est continu [50, Lemma 5.1]. Nous avons réussi à étendre cette approche à presque tous les cas considérés, à l'exception des transducteurs sur $(\mathbb{N}, <)$ en la présence de *guessing*. Pour cette raison, nous avons adopté une stratégie de preuve différente. Le lien entre continuité et ω -calculabilité pour les fonctions de mots infinis sur des alphabets *finis* a récemment été étudié dans [51] pour les transducteurs uni-directionnels et bi-directionnels. Nos résultats généralisent le cas des transducteurs uni-directionnels de [51, Theorems 6 and 12] au cas des mots de données.

Une fonction est *rationnelle* si elle est représentable par un transducteur non-déterministe uni-directionnel.

[50]: PRIEUR (2002), « How to decide continuity of rational functions on infinite words »

[51]: DAVE et al. (2020), « Synthesis of Computable Regular Functions of Infinite Words »

1.5. Plan

Les contributions présentées dans ce manuscrit sont doubles. La première partie est dédiée aux problèmes de synthèse (synthèse de Church et synthèse réactive). Le premier chapitre présente l'état des connaissances actuelles sur le cas des alphabets finis. Le Chapitre 4 est consacré aux automates à registres. Le Chapitre 5 définit la notion de spécifications automatiques avec données, ainsi que les transducteurs à registres séquentiels synchrones et présente notre proposition de généralisation du problème de Church au cas des mots de données, de même que le jeu de Church correspondant. Le Chapitre 6 établit les résultats qui concernent la synthèse à registres bornés pour les spécifications définies par des automates à registres universels (Section 6.2) et non-déterministes

(Section 6.3) sur les domaines de données $(\mathbb{D}, =)$ et $(\mathbb{Q}, <)$, ainsi que dans le cas des automates non-déterministes sans tests (Section 6.3.2). Le chapitre suivant est dédié à la synthèse non-bornée. Il s'ouvre par la présentation des résultats d'indécidabilité qui motivent l'étude de la synthèse à registres bornés (Théorèmes 6.20 et 7.1), puis continue avec les résultats de décidabilité qui concernent les automates à registres déterministes (Section 7.3). La partie se conclut par une discussion sur la question du domaine de l'implémentation (total ou partiel), qui marque la frontière entre décidabilité et indécidabilité dans le cas des alphabets infinis (Chapitre 8).

La seconde partie est consacrée à l'étude de l' ω -calculabilité et de la continuité pour les fonctions représentées par des transducteurs à registres non-déterministes asynchrones. Le modèle est présenté au Chapitre 10, qui établit également la clôture par composition pour cette classe de fonctions. Dans le Chapitre 11, nous présentons la notion d' ω -calculabilité, et montrons que pour les fonctions que nous étudions elle est équivalente à la continuité pour la distance de Cantor, de même pour les notions dérivées. Le reste de la partie est dédié à l'étude de différents domaines de données, où nous déterminons la décidabilité et la complexité de la décision des notions de fonctionnalité, ω -calculabilité et continuité, ainsi que leurs notions dérivées (Chapitre 12). La Section 12.1 se concentre sur le cas des domaines avec le seul prédicat d'égalité. La section suivante généralise l'étude à $(\mathbb{Q}, <)$ et aux domaines de données oligomorphes. Enfin, la Section 12.3 traite du cas de $(\mathbb{N}, <)$. La partie se conclut par une discussion sur les possibilités d'extensions de nos résultats.

Enfin, le document se termine par une conclusion générale (Chapitre 14) qui fait le lien entre les deux parties et ouvre des perspectives pour les recherches à venir.

1.6. Comment lire cette thèse ?

Conventions de numérotation Les environnements qui correspondent à des hypothèses, c'est-à-dire les définitions, les notations et les conventions, sont numérotés ensemble. De même concernant ceux qui correspondent à des conclusions, à savoir les théorèmes, les propositions, les corollaires, les lemmes, les propriétés et les observations. En outre, la numérotation se fait par chapitres.

Lien avec les résultats publiés Une partie des résultats présentés dans ce manuscrit ont été publiés dans des conférences ou des journaux. Lorsque c'est le cas, les théorèmes font le lien avec l'article correspondant ; la citation apparaît alors en gras.

Par contraposition, cette phrase signifie également que les théorèmes qui ne mentionnent pas de publication existante sont inédits.

Kaobook Cette thèse a été rédigée à l'aide de la classe $\LaTeX$ kaobook. Elle est dotée d'une marge étendue qui contient des notes, des références bibliographiques [52], les légendes des figures, et parfois même les figures elles-mêmes. Les notes de marge ne sont pas numérotées afin de ne pas perturber la lecture, mais sont alignées avec le contenu avec lequel elles sont en rapport. L'idée qui sous-tend cette classe est que ce document est probablement lu sur un ordinateur ou une machine analogue, sur

[52]: MAROTTA (2020), *Example and documentation of the kaobook class*

lequel il est malaisé d’aller au bas de la page pour trouver l’information requise (pour les notes de bas de page[¶]), ou à la fin du document (pour les références bibliographiques).

Définitions et Notations Une autre caractéristique de ce travail est l’usage du paquet `knowledge`, qui lie chaque occurrence d’une notion au lieu où elle est définie. Cette caractéristique peut sembler contradictoire avec la précédente, qui insiste sur la localité de l’information. La contradiction peut être dépassée en utilisant un lecteur de PDF récent, qui affiche une prévisualisation du lien ciblé, et par conséquent de la définition recherchée. La thèse comporte également un index des notations à la fin du document.

Conventions graphiques Une image vaut parfois mille mots[‡], et de nombreuses notions sont illustrées par des exemples. C’est également le cas de certaines idées de preuve. Pour des raisons de lisibilité, ce document est doté de conventions graphiques présentées au fil de la lecture, et groupées dans le Chapitre A en annexe. En particulier, les couleurs sont souvent utilisées dans les figures pour transmettre de l’information.

Feuilles de route La lecture est une affaire de goût (et de temps disponible), aussi ce document a-t-il été conçu pour permettre une lecture modulaire, en fonction des connaissances et des attentes du lecteur. En plus de la lecture linéaire classique, voici quelques parcours possibles :

En quête d’égalité Les investigations présentées ici ont démarré par l’étude du cas des domaines de données avec le seul prédicat d’égalité. Les automates à registres se comportent plutôt bien sur de tels domaines, ce qui donne lieu à des constructions et à des preuves plus simples. Le lecteur qui s’intéresse seulement à ce cas peut se dispenser des développements liés aux structures plus riches (les ordres denses et discrets, ainsi que les domaines oligomorphes). Les sauts correspondants sont mentionnés dans la marge au fil du document.

Tout sur les registres Le Chapitre 4 est conçu comme une introduction aux automates à registres. Il a été écrit avec en tête ses applications à la synthèse, mais présente la plupart des propriétés standard du modèle, et comporte des pointeurs vers la littérature sur le sujet.

Les deux faces de la synthèse réactive Les Chapitres 6 et 7 peuvent être lus indépendamment. Ils ont chacun pour pré-requis les Chapitres 3 à 5.

Rien ne sert de courir... La Partie II est consacrée aux transducteurs à registres asynchrones, et peut, pour l’essentiel, être lue de manière indépendante. Elle suppose néanmoins une connaissance élémentaire des automates à registres, qui est disponible dans le Chapitre 4.

Le document comporte également une section dédiée aux références bibliographiques, située à la fin. Les liens présents dans le corps du texte renvoient à cette section.

La plupart des figures ont été dessinées à l’aide du merveilleux paquet `TikZ`.

Les couleurs ne sont pas utilisées dans le corps du texte, sauf pour les liens, afin que le document ne ressemble pas à Elmer.

Le lecteur ou la lectrice est bien sûr invitée à élaborer sa propre feuille de route.

[¶] Quelques notes de bas de page ont survécu, pour le matériau *auxiliairement* auxiliaire comme les jeux de mots.

[‡] Littéralement, dans le cas des articles de conférence.

2.1. Program Synthesis

Programming consists in automating tasks. Being itself a task, one can wonder whether it can be automated, and this is the goal of program synthesis. For instance, consider the problem of sorting a list of numbers. Instead of specifying *how* the program should proceed (e.g. construct the sorted list by iteratively inserting each value at its right place, as is done when sorting playing cards), one would prefer to be content with specifying *what* it should do (take an input list and output a sorted list that contains the same elements). Thus, given a specification of the expected behaviour of the program (*what*), a program synthesis algorithm should output a program that exhibits this behaviour (*how*) if it exists, or No otherwise. Such a program is called an implementation of the input specification, and we say that it fulfils the specification.

To put things formally, in its most general form, a synthesis problem has two parameters, a set of inputs In and a set of outputs Out , and relates two classes $\mathcal{S}$ and $\mathcal{I}$ of specifications and implementations respectively. A specification $S \in \mathcal{S}$ is a relation $S \subseteq \text{In} \times \text{Out}$ and an implementation $I \in \mathcal{I}$ is a function $I : \text{In} \rightarrow \text{Out}$. The $(\mathcal{S}, \mathcal{I})$ -synthesis problem asks, given a (finite representation of a) specification $S \in \mathcal{S}$, whether there exists $I \in \mathcal{I}$ such that for all $u \in \text{In}$, $(u, I(u)) \in S$. If such an I exists, then the procedure must return a program computing I .

This is of course an ambitious objective, and it has long been known that it is not reachable for general-purpose programming languages. This does not come as a surprise, since solving the program synthesis problem in full generality amounts to finding a program which, given the specification of an algorithmic problem, finds an algorithmic solution to it if it exists. In a way, this impossibility is reassuring as it means that computer science will continue to thrive. It may have deterred efforts in this direction for a time, but in the same way the undecidability of the halting problem [1] triggered research efforts in program analysis, the impossibility of general synthesis set off investigations in decidable restrictions of the problem. * There are broadly two ways to recover decidability. First, one can guide synthesis by providing additional information to the synthesis algorithm. This is the rationale behind syntax-guided synthesis [2], that takes as input a template of the target program, and fills the gaps so as to satisfy the specified behaviour. The granularity of the template allows to act on the hardness of the problem. The approach of bounded synthesis [3] is based on a similar idea: by setting a bound on the size of the target program, we bound the exploration space, so the procedure is guaranteed to terminate. Second, one can restrict the classes of specifications and/or of implementations.

We present the insertion sort, which is technically not optimal, as it is easier to describe and closer to everyday human practice.

Be careful as there is some higher-order stuff going on here: synthesis is about designing algorithms that write programs.

In our development, we use the term ‘realises’ instead of ‘fulfils’ due to its link with formal logic.

In other words, this consists in finding a perfect computer scientist that is able to decide whether a given problem can be solved or not, and, if yes, provide a solution. Formally, this impossibility is proven by reducing from the halting problem or the Entscheidungsproblem: as soon as one can specify arithmetic requirements, being able to solve the program synthesis problem implies in particular to be able to decide whether a given arithmetic formula holds.

[1]: Turing (1936), ‘On Computable Numbers, with an Application to the Entscheidungsproblem’

[2]: Alur et al. (2013), ‘Syntax-guided synthesis’

[3]: Finkbeiner and Schewe (2013), ‘Bounded synthesis’

Those approaches are of course not mutually exclusive.

* In that sense, the history of hitherto existing computer science is the history of complexity classes struggle, between hardness results and ways to overcome them.

2.2. Reactive Systems

2.2.1. Automata

A class of choice is that of programs computed by finite-state machines, also known as automata. As the name suggests, those machines have a finite number of memory states. They process their input bit by bit from left to right, using their memory state to store information, and are typically used to model embedded systems or hardware circuits. In their simplest form, they consist in acceptors for a language, i.e. they take as input a sequence of letters (called a word) and answer Yes or No, depending on whether it belongs to the language or not. As a simple example, consider the language of words that contain an odd number of a . To decide whether a given word belongs to this language, a program does not need to count the number of a s, only to keep in memory the parity of this number. Figure 2.1 depicts an automaton that accepts this language. In this example, when it is in a given state and reads an input

The knowledgeable reader who already knows what a finite-state automaton is can jump to the next Section 2.2.2.

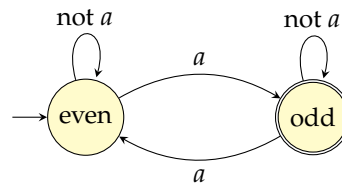


Figure 2.1: An automaton with two memory states that checks whether its input contains an odd number of a . States are depicted as circles, and transitions as arrows. For instance, when the automaton is in state ‘even’ and reads an a , it transitions to state ‘odd’. Initially, it starts in state ‘even’, as indicated by the incoming arrow. The machine answers Yes if and only if its computation ends in an accepting state, marked by a double circle (here, only the state ‘odd’ is accepting). To illustrate our point, ponder about what happens when the automaton reads input baa . First, it starts in its initial state ‘even’. When it reads the first letter b , it takes the loop over ‘even’ and stays in this state. Then, it reads a and goes to state ‘odd’. Finally, when it reads a again, it goes back to state ‘even’ and answers No, since it is in a state that is not accepting. One can check that along any computation, the automaton is in state ‘even’ whenever the input read so far contains an even number of a (analogously for state ‘odd’).

letter, the automaton has a unique possible transition; we say that it is deterministic. In general, it is convenient to equip these machines with the possibility to *guess* a transition to take among multiple ones, we call them *non-deterministic*. Consider for instance the language of words whose second letter before the end is an a . Since the program reads its input from left to right, it cannot know how many letters there are before the end. Figure 2.2 on the next page depicts a non-deterministic automaton that recognises this language. Note that an actual program cannot ‘guess’ the right computation, except by executing them all in parallel and checking that at least one is accepting. Thus, non-determinism should be seen as a way to compactly represent a set of behaviours. It is well-known that non-deterministic automata always admit an equivalent deterministic automaton, but can be exponentially more succinct.

Note that it is not required that a non-deterministic automaton *actually* has the possibility to choose between different transitions. In other words, the notion of non-deterministic automaton generalises that of a deterministic automaton. In particular, a deterministic automaton is also a non-deterministic automaton, counter-intuitive as it seems.

When representing specifications we will also be interested in universal automata, also known as co-non-deterministic. Instead of asking that at least a computation is accepting, we ask that all of them are accepting. For instance, the automaton of Figure 2.2, seen as a universal one and interverting accepting and non-accepting states, accepts exactly words whose second letter before the end is *not* an a . This is a more general phenomenon, as both semantics are dual to each other: switching from non-deterministic to universal semantics and complementing the set of accepting states allows to recognise the complement language.

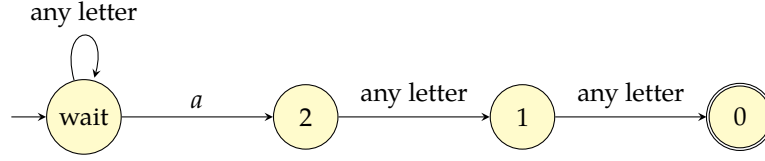


Figure 2.2.: An automaton that checks whether the second letter before the end is an a . It starts in state ‘wait’. When it reads an input letter distinct from a , it stays in this state. When it reads an a , it has to make a choice: either stay in ‘wait’, taking the self-loop, or transition to state 2. To make this choice, it guesses whether this a is two letters away from the end of the word. If this is the case, it goes to state 2, and then checks that its guess is correct by reading the next two letters. If its input ends at this moment, it answers Yes, since it is in state 0, which is accepting. If there is still some input to read, the computation fails as there is no outgoing transition from 0. By definition, a non-deterministic finite-state automaton answers Yes if and only if *there exists* a computation that ends in an accepting state. For instance, on input $ababb$, on reading the first a , the automaton can either go to 2 or stay in ‘wait’. If it does the former, after having read aba , it is in state 0 and still has two letters to read, so the computation fails. In the latter case, it is again presented with this choice when reading the second a . If it stays in ‘wait’, it will stay in this state until the end since there are no more a s to read, and will not accept. However, if it transitions to 2 at this point, it will end in state 0 and accept. Thus, there exists an accepting computation, which means that $ababb$ is accepted by the automaton, which answers Yes.

2.2.2. Transducers

Often, we want a program to provide more information than simply whether it accepts or rejects its input. For instance, if you come back to the automaton of Figure 2.1, we might ask it to report along the computation whether the number of a it has read so far is odd. E.g., on input aba , it would output $\text{odd} \cdot \text{odd} \cdot \text{even}$ (assuming it first reads a letter before outputting something). In this particular case, this can be achieved by making the automaton print its current state along the run. However, in general, it is convenient to separate the internal states of the machine from its interactions with its environment. To that end, we instead enrich the transitions with output letters: everytime it reads an input, the machine transitions to some state and additionally output a letter. This allows to represent programs that conduct transformations on their input, e.g. replace every a with b , b with c and c with a when it read an even number of a , and conversely $a \rightarrow c \rightarrow b \rightarrow a$ otherwise (see Figure 2.3).

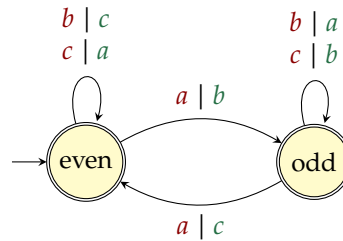


Figure 2.3.: A transducer that shifts its input in one direction or the other, depending whether it contains an even number of a . For simplicity, we consider that it only reads words over the alphabet $\{a, b, c\}$. Inputs are colored in red, outputs in green, and the transition ‘even $\xrightarrow{a|b}$ odd’ reads ‘when in state “even”, on reading a , output b and transition to state “odd”’. For instance, on input $abbac$, the transducer outputs $baaca$. We can also ask e.g. that it only accepts inputs with an odd number of a , making only the state ‘odd’ accepting.

ω -Transducers

In our setting, we are more interested in the *interaction* between the program and its environment, i.e. the relation between its input and its output, rather than on its termination: consider a setting in which a system continuously interacts with its environment. The environment provides an input signal, modelled as an element from a finite set that we

usually call a letter, and the corresponding set is called an alphabet. Then, the system reacts with an output letter, and this goes on *ad infinitum*. Their interaction yields an infinite sequence of interleaved input and output letters. We assume that the interaction does not terminate, as the focus is set on the *control* aspect on the system rather than its computational behaviour. Note that there is an abstraction operation going on here, that is intrinsic to the process of modelling: we need objects that can be algorithmically analysed, that are derived from the concrete system. Thus, we work under the assumption that the mathematical model is a safe abstraction of the system we consider. A paradigmatic example of a reactive system is that of a server that interacts with its clients: the latter address requests that the server has to grant in due time. Here, we present a variation on the theme, that essentially exhibits the same features: consider a coffee machine modelled as the system, that interacts with its users who together form the environment. Users provide inputs by pressing the buttons of the machine, to which the machine reacts with outputs that consist in conducting various actions (booting, pouring coffee, displaying information on its screen, shutting down, etc). Then, a typical interaction could consist in a user that turns the machine on, orders a coffee, receives it and turns it off. If no-one uses the machine ever again, it stays idle indefinitely. This can be modelled as the infinite word

$$\text{press\_on} \cdot \text{boot} \cdot \text{order\_coffee} \cdot \text{pour\_coffee} \cdot \text{press\_off} \cdot \text{shut\_down} \cdot (\text{idle} \cdot \text{idle})^\omega$$

where the ω exponent in $(\text{idle} \cdot \text{idle})^\omega$ means that the sequence $\text{idle} \cdot \text{idle}$ is repeated infinitely many times. The input alphabet is $\{\text{press\_on}, \text{order\_coffee}, \text{press\_off}, \text{idle}\}$ and the output alphabet is $\{\text{boot}, \text{pour\_coffee}, \text{shut\_down}, \text{idle}\}$. Here, the notion of alphabet is to be understood as a finite set: elements of those example alphabets consist in words, but they might as well be replaced with symbols (e.g. O for press_on , P for pour_coffee , etc).

By convention, inputs are coloured in red, and outputs in green. The symbol $\cdot$ denotes concatenation.

The machine is operated by a controller that responds in a timely manner to its input. If the controller is encoded on a chip, it has finitely many memory states, and can thus be modelled as a deterministic transducer that operates over infinite words; we call them ω -transducers. A typical coffee machine could be modelled as the ω -transducer of Figure 2.4. In general, we assume that those programs can accept any sequence of input, so all their states are implicitly accepting. Let us insist on the fact that

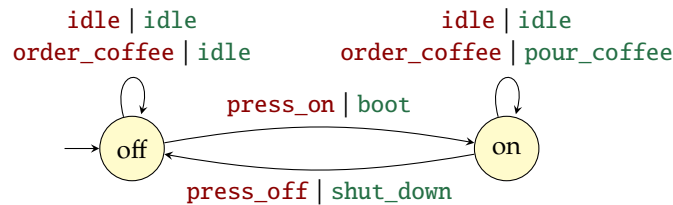


Figure 2.4: An ω -transducer modelling a coffee machine. It is initially off, and can be turned on by pressing ‘on’. When off, any other command has no effect. When on, one can order coffee, and the machine diligently pours it.

the interaction is to be conceived over the entire lifetime of the machine. For instance, when it shuts down, the interaction is not over, as it might be turned on again in the future. The overall timeline is finite, since the

machine will eventually stop working, in spite of all the efforts to keep it afloat. However, focusing on the interaction allows to specify more easily requirements regarding untimely termination, e.g. when the machine unexpectedly halts. This would be the case for instance in the faulty sequence `press_on · boot · order_coffee · shut_down · (idle · idle)ω`, where the machine shuts down before serving coffee. Thus, a typical specification could ask that everytime an occurrence of `order_coffee` is met, an occurrence of `pour_coffee` is met on the next step. This requirement might however be too strong. First, one cannot expect the machine to serve a coffee if it is off, so this requirement should be limited to periods where a `boot` has occurred with no `shut_down` afterwards. However, this specification is trivially satisfied by a machine that never boots, so one should also specify how it should boot. Besides, it might take some time for the machine to serve a coffee, e.g. if it has to conduct internal operations like grinding the coffee, warming the water etc, so it might not be able to react on the next time step, and one may instead ask that it occurs within the next k steps for some number k , or even simply that it eventually occurs.

If only with the collapse of the solar system, that will happen in a long but finite time.

2.3. Reactive Synthesis and the Church Problem

Overall, a specification relates infinite sequences of inputs to a set of infinite sequences of outputs that form acceptable behaviours. Since we are interested in reactive systems that are in constant interaction with their environment, it is often more convenient to see a specification as the set of words that consists in the interleaving of an infinite sequence of input with a corresponding acceptable behaviour. For instance, in the above example, if one asks that `pour_coffee` eventually happens after `order_coffee` is met, the corresponding specification consists in interleavings of an input sequence with any output sequence that satisfy this property.

Then, the reactive synthesis problem asks, given a finitely represented specification, to generate a reactive program that satisfies the specification if it exists, and to provide a witness that the specification cannot be fulfilled otherwise. The problem was first posed in 1957 by Church, who asked what is now known as the Church problem [5]:

Given a requirement which a circuit is to satisfy, we may suppose the requirement expressed in some suitable logistic system which is an extension of restricted arithmetic. The synthesis problem is then to find recursion equivalences representing a circuit that satisfies the given requirement (or alternatively, to determine that there is no such circuit). ([4])

Reactive systems are notoriously difficult to design correctly, and the main appealing idea of synthesis is to automatically generate systems that are *correct by construction*. In Church's setting, the class of implementations consists in reactive programs that can be executed by boolean circuits, i.e. that use a fixed finite amount of memory, which can be modelled by ω -transducers.

[5]: Thomas (2009), 'Facets of Synthesis: Revisiting Church's Problem'

[4]: Church (1957), 'Applications of recursive arithmetic to the problem of circuit synthesis'

2.3.1. Monadic Second-Order Logic

Regarding the specification formalism, Church mainly considered Monadic Second Order logic (MSO) over infinite words with the predicate $+1$ (or, equivalently, $\leq$) and predicates for letters of the alphabet. This logic is able to express the various specifications that we described above. For instance, asking that `pour_coffee` eventually occurs after `order_coffee` can be written as follows:

$$\forall x, \text{order\_coffee}(x) \Rightarrow \exists y, y \geq x \wedge \text{pour\_coffee}(y)$$

2.3.2. Synthesis as a Game

In an unpublished technical report [6], McNaughton pointed out that the problem can be seen as an infinite duration game between two players [7] that we call the Church game. One player models the system, and the other the environment. The goal of the system is to satisfy the specification and is antagonist to that of the environment, which aims at violating it[†]. Then, a winning strategy for the system corresponds to an implementation of the specification, and a winning strategy for the environment witnesses that no such implementation exists. In 1969, J.R. Büchi and L.H. Landweber solved the problem for MSO specifications with game-theoretic techniques [8]. The method relies on converting the specification to a finite-state machine (more precisely an ω -automaton, i.e. a finite-state automaton over ω -words) that accepts the interactions which satisfy the specification. Then, such a machine is used as an observer for the game, and defines its winning condition. Later on, Rabin elaborated a solution that relies on tree automata [10]. In this thesis, we adopt the game-theoretic presentation, which allows to leverage known results about parity games, and provides an elegant formalism to describe implementations. We broadly follow the approach of [5] to the Church problem.

[6]: McNaughton (1965), *Finite-state infinite games*, cited in [5].

[7]: McNaughton (1993), 'Infinite Games Played on Finite Graphs'

They cannot both have a winning strategy, since their objectives are antagonistic.

[8]: J.R. Büchi and L.H. Landweber (1969), 'Solving sequential conditions finite-state strategies'

Recall that an interaction is modelled as an infinite sequence that alternates between input and output signals.

[10]: Rabin (1972), *Automata on Infinite Objects and Church's Problem*

2.3.3. Linear Temporal Logic

For MSO specifications, Church's problem inherits the non-elementary lower bound on the satisfiability of Monadic Second-Order Logic [11]. This skyrocketing complexity has long prevented any practical applications of synthesis algorithms. A first attempt to tame it was made with the introduction of Linear Temporal Logic (LTL) [12] in the context of model-checking for formal verification. The latter problem is the (better behaved) twin of the synthesis problem which, given a specification *and* a system, asks to determine whether the system satisfies the specification (in contrast, synthesis is about *building* the system *from* the specification). The authors showed that such a problem is PSPACE-complete, and later on established that the reactive synthesis problem is 2-ExpTime-complete [13]. This complexity gap between the two problems is the reason why, as of today, formal verification has reached a wider level of practical applicability than its synthesis counterpart [14]. However, in the last decade, efficient methods and implementations have triggered a renewed

[11]: Meyer (1975), 'Weak monadic second order theory of successor is not elementary-recursive'

[12]: Pnueli (1977), 'The Temporal Logic of Programs'

[13]: Pnueli and Rosner (1989), 'On the Synthesis of a Reactive Module'

[14]: Finkbeiner (2016), 'Synthesis of Reactive Systems'

[†] Computer science modelling thus yields the unexpected situation of a coffee machine that is pitted against its users.

interest for this field [3, 15, 16], and it is now a very active area of research [9, 17–20]. To get an idea of this formalism, let us point out that asking that `pour_coffee` eventually occurs after `order_coffee` can be written as $G(\text{order\_coffee} \Rightarrow F\text{pour\_coffee})$, where G means that the property holds along the whole interaction, and F that it holds at some point in the future. Thus, this reads as ‘Along the entire run, it should be the case that if `order_coffee` is met, `pour_coffee` should be met in the future’.

2.3.4. ω -Automata

One can also represent specifications directly as ω -automata. Indeed, both MSO and LTL specifications can be turned into ω -automata that recognise acceptable sequences. Over infinite words, acceptance cannot be defined by asking that a computation ends in an accepting state, since it goes on indefinitely. Instead, we can ask that an accepting state is visited infinitely often (the so-called Büchi condition), or that all rejecting state are visited only finitely often (co-Büchi). More general conditions exist, but this is sufficient for our purpose here.

It is often easier to design a non-deterministic automaton that recognises violations of the specification and complement it as a universal one. For instance, the non-deterministic automaton of Figure 2.5 checks that some `order_coffee` is never followed by an `pour_coffee`. Dualising it yields a universal automaton that checks that any `order_coffee` is eventually followed by `pour_coffee`.

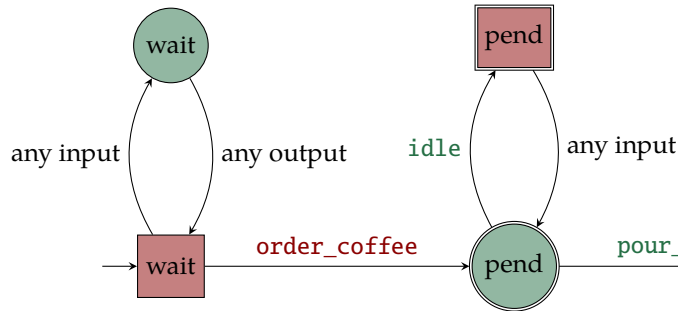


Figure 2.5.: A non-deterministic automaton that checks that some `order_coffee` is never followed by an `pour_coffee`: when in ‘wait’, it guesses at some point that this occurrence of `order_coffee` is never followed by `pour_coffee` and transitions to ‘pend’ (ing order). The computation accepts if it loops infinitely in ‘pend’, which is the only accepting state. Dualising it yields a universal automaton that checks that any `order_coffee` is eventually followed by `pour_coffee`: ‘pend’ is now the sole pair of rejecting states, and it is visited infinitely often for at least one computation if and only if some order is never satisfied.

Reactive synthesis and related definitions are presented in Chapter 3, which summarises known results that are needed for our subsequent study.

- [3]: Finkbeiner and Schewe (2013), ‘Bounded synthesis’
- [15]: Ehlers (2012), ‘Symbolic bounded synthesis’
- [16]: Filiot, Jin, and Raskin (2011), ‘Antichains and compositional algorithms for LTL synthesis’
- [9]: Bloem, Chatterjee, and Jobstmann (2018), ‘Graph Games and Reactive Synthesis’
- [17]: Jacobs et al. (2017), ‘The first reactive synthesis competition (SYNTCOMP 2014)’
- [18]: Bonakdarpour and Finkbeiner (2020), ‘Controller Synthesis for Hyperproperties’
- [19]: Finkbeiner, Geier, and Passing (2021), ‘Specification Decomposition for Reactive Synthesis’
- [20]: Almagor and Kupferman (2020), ‘Good-Enough Synthesis’

2.4. To Infinity and Beyond: the Shift to an Infinite Alphabet

2.4.1. Data Words

The renewal of interest for synthesis also opened the way to possible extensions of existing techniques to more general settings. If you come back to the coffee machine example (or, equivalently, to a server that has to grant requests), it is not always the case that the orders of all users can be amalgamated into a fixed finite number of requests. For instance, each user might have a particular request (e.g. having their name on the cup) that cannot be modelled as a finite alphabet. This is captured by the notion of data words, whose elements take their value in an infinite set of data values $\mathbb{D}$, that is called a data domain. For modelling purposes, it is often convenient to be able to label data values with letters from a finite alphabet Σ , even though it can be encoded in the data domain. For instance, the fact that a given user ordered a coffee is modeled as the pair `(order_coffee, id)`, where `id` is a data value (e.g. a natural number) that represents the user. Data domains can be equipped with various structures, given by relational predicates and constants that are interpreted over the domain. The simplest one consists in a data domain with the equality predicate, with which one can specify that user $n^\circ i$ receives cup $n^\circ i$. Then, one can consider the case of a linear ordering on users, to model the property that some users might be more equal than others, and consequently have precedence over others. This framework also allows to consider domains where operations can be conducted on data values: if the machine receives money and has to check that the right amount has been paid, it has to be able to conduct additions and comparisons for equality. However, we enter murky waters here: as the seasoned reader might feel already, this kind of structure allows to express arithmetic operations that are sufficient to get undecidability (see Section 4.9).

2.4.2. Register Automata

Models of reactive systems have to be correspondingly extended, so as to be able to manipulate data values. Among others, *register automata* are one of the main extensions of automata recognising languages of data words [21, 22]. They consist in finite-state automata that are equipped with a finite set of registers in which to store data values that are read, and to compare the current data with the content of some of the registers with regards to the predicates of the data domain. Note that a given register can only contain a single data value at a time, which means that when a data value is stored in some register, its previous content is overwritten. They were initially introduced for data domains with the equality predicate only, then generalised to linear orders [23, 24]. We further extend them to arbitrary domains, in the spirit of the G-automata of [25, Section 3], and of [26, Chapitre 1].

As for finite-state automata, those automata can be equipped with two dual semantics, whether they are considered non-deterministic or universal. A non-deterministic register automaton accepts its input if at

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

[22]: Segoufin (2006), ‘Automata and Logics for Words and Trees over an Infinite Alphabet’

[23]: Benedikt, Ley, and Puppis (2010), ‘What You Must Remember When Processing Data Words’

[24]: Figueira, Hofman, and Lasota (2016), ‘Relating timed and register automata’

[25]: Bojańczyk, Klin, and Lasota (2014), ‘Automata theory in nominal sets’

[26]: Bojańczyk (2019), *Atom Book*

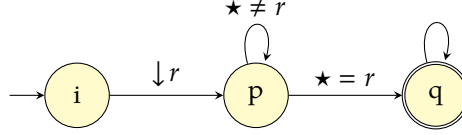


Figure 2.6.: A deterministic register automaton with one register r over data domain $(\mathbb{N}, =)$ that checks that the first data value appears again in the input. It initially stores it ($\downarrow r$), and then waits in p until it sees it again ($\star = r$ holds whenever it reads an input data value that is equal to the content of r), then transitions to q , in which it loops indefinitely.

least one of its executions is accepting, i.e. if it can guess a correct run over its input. In contrast, a universal register automaton accepts its input if all its executions are accepting. Contrary to the finite alphabet case, both models are not equivalent, and recognise dual classes of languages [21, Proposition 5]. One can also consider a semantics that combines the two, namely alternation [27]. However, it yields a model that is too expressive for our purpose, since the emptiness and universality problems are undecidable [28, Theorems 4.1 and 4.2], which dampens any hope for the harder problem of synthesis from specifications expressed as alternating register automata.

[27]: Chandra, Kozen, and Stockmeyer (1981), ‘Alternation’

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

See also [29]: Figueira (2010), ‘Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données)’, *PhD thesis*.

2.4.3. Register Transducers

The automata-theoretic approach of verification and synthesis can be extended to data words through automata and transducers that operate with data values. In this perspective, transducers can be extended to *register transducers* as a model of reactive systems over data words: a register transducer is equipped with a finite set of registers, and when reading an input labelled data $(\sigma, d) \in \Sigma \times \mathbb{D}$, it can compare d with the content of some of its registers, and depending on the result of this test, deterministically store d in some of its registers and output a label γ along with the content of one of its registers (see the example of Figure 2.7). Its executions are then data words alternating between input

Recall that Σ is a finite alphabet of labels, and $\mathbb{D}$ is an infinite data domain.

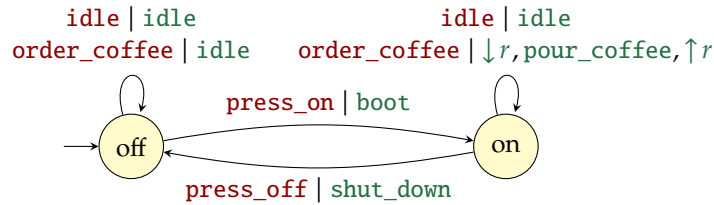


Figure 2.7.: A register transducer modelling a coffee machine that specifically serves the client that ordered.

and output labelled data. In the same way as automatic specifications are described by an ω -automaton that reads infinite sequences of alternating input and output signals, register automata can thus be used to represent specifications as data languages (cf Figure 2.8 on the following page).

2.4.4. Automata and Logics over Data Words

Contrary to the finite alphabet case, there is no known correspondence between automata and logics that can be exploited to solve the synthesis problem for specifications expressed in a logical formalism following a path similar to the Church problem for MSO specifications, that consists

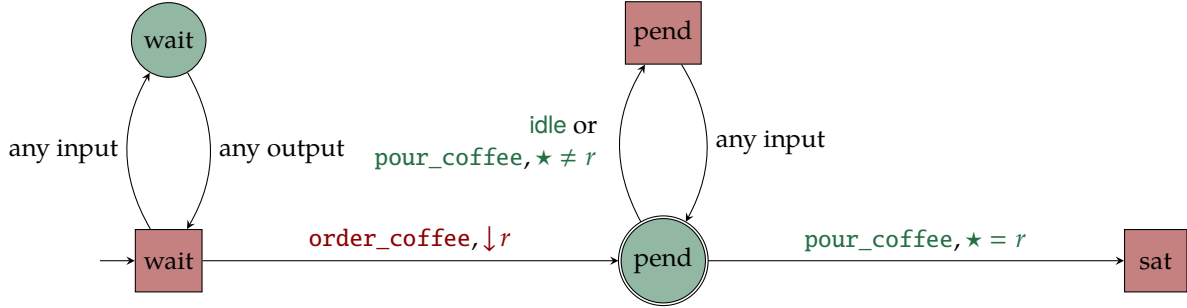


Figure 2.8.: A universal co-Büchi register automaton checking that every `order_coffee` of any client id is eventually followed by `(pour_coffee, id)`.

in representing the winning condition of the Church game with an automaton. In [28], the authors relate LTL with the freeze quantifier [31], that allows to ‘store’ a data value that appears in the formula, and register automata. They demonstrate that LTL with freeze formulas can be recognised by two-way alternating register automata, with a number of registers that is the same as the number of data variables that can be frozen [28, Theorem 4.1]. However, the emptiness problem of such a model is undecidable, already with one register when considering infinite words [28, Theorem 5.2]. Later on, Demri, D’Souza, and Gascon considered the Logic of Repeated Values, which limits the use of the freeze quantifier so as to get a logic that is both decidable and closed under negation [32]. However, it was shown in [33, Theorem 4.1] that games with objectives given as LRV formulas are undecidable. The case of single-sided specifications is decidable, and inspired our treatment of the synthesis problem for specifications expressed as deterministic register automata, triggering the introduction of the one-sided restriction. Yet, for lack of a suitable logic for expressing specifications that is expressive enough and allows both the system and the environment to manipulate data values, as a first step, we study the case of specifications expressed by register automata, that we call data automatic.

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

[31]: French (2003), ‘Quantified Propositional Temporal Logic with Repeating States’

[32]: Demri, D’Souza, and Gascon (2012), ‘Temporal Logics of Repeating Values’

We say that a class of games is undecidable if it is undecidable whether a given player has a winning strategy in them.

[33]: Figueira, Majumdar, and Praveen (2020), ‘Playing with Repetitions in Data Words Using Energy Games’

2.4.5. Control and Data Processing

Equipping finite-state automata with registers also allows to separate two aspects of the behaviour of the system, namely control and data processing, which demonstrates the relevance of the model already with a finite set of data values. Indeed, existing synthesis algorithms do not scale well when the size of the alphabet increases. For instance, if you come back to our running example, consider the setting where the machine can serve various types of beverages. There are finitely many, yet what matters is not so much the precise nature of the beverage as the fact that if a client orders x , they receive x . This parametricity cannot be captured by a finite alphabet with no further structure. This is illustrated by the following setting: assume the users make their choice in a set B , and that k clients can order at the same time. Then, the machine has to keep in memory the set of beverages that has been ordered. Any subset of size k of B is possible, which means that the smallest automaton that represents the specification needs memory $\binom{|B|}{k}$. In contrast, it can be represented with a register automaton with k registers, that stores the order of each client in the corresponding register, and later on checks that each order

has been correctly served. This is reflected on the implementation side: with no registers, the implementation again needs memory $\binom{|B|}{k}$, while a machine with registers can store the orders in memory and serve them back in order using k states and k registers.

This direction of research has been more specifically explored for specifications expressed in (fragments of) first-order logic over data words under the name of *parameterised synthesis*, that operate under the assumption that there is a fixed but arbitrary number of distinct elements that interact with the system [34].

2.4.6. Contributions

The following contributions are detailed in the first part of the manuscript. Chapter 4 contains a presentation of the model of register automata, that is at the core of our study since we use it to express specifications. It describes its useful properties, that are extracted from the literature and established for arbitrary data domains when possible. We then propose a generalisation of the Church synthesis problem to specifications over data words, that targets register transducers, as a counterpart of transducers over data words (Chapter 5).

Register-Bounded Synthesis

In our first contribution (Chapter 6), we examine the register-bounded synthesis problem [35], that consists in targeting implementation defined by register transducers with a number of registers that is given as input. We start with specifications expressed as universal register automata: the universal semantics is well-suited to express specifications, since it is naturally closed under intersection. Moreover, it is often convenient to specify a violation of the specification. For instance, the requirement that every user eventually receives their coffee (or, in a client/server setting, that any request is eventually granted) can be expressed by a universal automaton that checks that no order is never satisfied (Figure 2.8 on page 33). The fact that an order is left unsatisfied is easily expressed using a non-deterministic automaton, as it suffices to guess a faulty one; the specification is then obtained by dualising the automaton to get negation. To the contrary, in the infinite alphabet case, it cannot be specified using a non-deterministic register automaton.

We show that the register-bounded synthesis problem is decidable for specifications recognised by universal register automata over data domains $(\mathbb{D}, =)$ (Theorem 6.14) and $(\mathbb{Q}, <)$ (Theorem 6.16). The problems more precisely lie in 2-ExpTime. The first result was known from [36, Corollary 3], and independently proved by us in [37, Theorem 12]. The compositionality of our proof allows to extend it to $(\mathbb{Q}, <)$ with little additional work, and to further equip universal register automata with a guessing mechanism, as was introduced for non-deterministic register automata [38]. The case of $(\mathbb{N}, <)$ is more involved, since the structure of $(\mathbb{N}, <)$ induces behaviours that cannot be captured regularly, as witnessed by the fact that runs of those register automata correspond to ω B-regular languages [39, Theorem 17]. This is the topic of ongoing research that is not described in this manuscript as it deserves to mature a bit more.

[34]: Bérard et al. (2020), ‘Parameterized Synthesis for Fragments of First-Order Logic Over Data Words’

[35]: Khalimov and Kupferman (2019), ‘Register-Bounded Synthesis’

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

[37]: Exibard, Filiot, and Reynier (2019), ‘Synthesis of Data Word Transducers’

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

To elaborate our decision procedures, we develop a generic method that relies on a reduction to reactive synthesis over finite alphabets, and yields decidability when the class of specification enjoys some closure properties and language operations. It is based on a transfer theorem (Theorem 6.8) that expresses equi-realizability of a data word specification and its finite alphabet counterpart, whose definition is based on the notion of action sequences which essentially represent syntactical executions of register transducers.

Non-deterministic register automata specification are not so easily tamed, since the problem is already undecidable when targeting implementations with a single register (Theorem 6.19). We consider the subclass of test-free specifications, where the input transitions of the specification automaton cannot depend on tests over the input data values. Still, those machines are able to duplicate, erase and move the data values. Our generic method allows to show that the register-bounded synthesis problem for this class of specifications is decidable in 2-ExpTime , by making the link with the notion of origin, as introduced in [40].

A preliminary version of this work appeared as [37], that was later on extended to [41]. This chapter additionally features the extension to the case of $(\mathbb{Q}, <)$ and the handling of the guessing mechanism. The proofs have also been reformulated in the slightly more general context of template-guided synthesis.

The results are summarised in Tables 2.1 (for $(\mathbb{D}, =)$) and 2.2 (for $(\mathbb{Q}, <)$), which also present the results from the next contribution.

Unbounded Synthesis

The next chapter is dedicated to the study of the unbounded synthesis problem, where the target implementation can be any register transducer, i.e. there is no given bound on its number of registers. We show that the problem is undecidable for specifications expressed as non-deterministic and universal register automata over $(\mathbb{D}, =)$ (Theorems 6.20 and 7.1), which also motivates the register-bounded synthesis approach. The latter problem was posed as an open problem in [36]. To recover decidability, we consider the case of specifications expressed by deterministic register automata, where determinism is to be understood over joint inputs and outputs. We show that if the specification consists in a locally concretisable automaton, i.e. such that for any state, a transition can be taken independently of the valuation of the registers, then it essentially suffices to play on the specification as if it were a game arena, as is done in the finite alphabet case. This yields decidability of the reactive synthesis and Church problem for the case of $(\mathbb{D}, =)$ (Theorem 7.15) and of $(\mathbb{Q}, <)$ (Theorems 7.16 and 7.17). In the case of $(\mathbb{D}, =)$, we additionally show that, as in the finite alphabet case, the reactive synthesis problem is equivalent to the Church problem. In other words, if the specification admits an implementation, it admits one that is computed by a register transducer. This is not the case anymore for $(\mathbb{Q}, <)$ (Property 7.18).

Thus, we also study the one-sided restriction, that is analogous to the single-sided restriction for LRV games that allows to recover decidability in [33]. The output transitions of the specification automaton are restricted to data values that are present in the registers. This is equivalent to the

[40]: Bojanczyk (2014), ‘Transducers with Origin Information’

[37]: Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Ed. by Wan Fokkink and Rob van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019

[41]: Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *Logical Methods in Computer Science* 17.1 (2021)

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

Let us recall the distinction between reactive synthesis and Church synthesis: the former targets implementations that can be any synchronous programs, i.e. that are not restricted memory-wise, while the latter targets transducers, i.e. *finite-state* machines.

[33]: Figueira, Majumdar, and Praveen (2020), ‘Playing with Repetitions in Data Words Using Energy Games’

case where they would only depend on the labels of the data values, since choosing a register to output can be simulated by picking a letter that denotes the corresponding register. In other words, they essentially behave like transducers, so solving the synthesis problem amounts to picking output transitions along the run. In game-theoretic terms, this means asking that the system player picks her actions in a Boolean set. For this subclass, we again have that it suffices to target register transducers for specifications recognised by one-sided register automata over $(\mathbb{Q}, <)$. Finally, we show that this restriction marks the decidability boundary over $(\mathbb{N}, <)$ (Theorems 7.19 and 7.28). Decidability for one-sided specifications over $(\mathbb{N}, <)$ is obtained by showing that it suffices to consider an ω -regular approximation of the synthesis game (recall that the behaviours of register automata over $(\mathbb{N}, <)$ are not ω -regular in general). Undecidability stems from the fact that alternation and antagonism between the system and its environment can be exploited to simulate counter machines: one player suggests successive values of the counter, while the other is in charge of checking that she does not cheat.

This second line of work was also presented in [37] and [41]. In this manuscript, the argument is presented in a more modular way, using the concept of game-ready specification automaton. It also extends the study to the case of $(\mathbb{Q}, <)$.

The case of $(\mathbb{N}, <)$ has been treated in [42], as it requires the introduction of more elaborate proof techniques.

[37]: Exibard, Filiot, and Reynier (2019), ‘Synthesis of Data Word Transducers’

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

[42]: Exibard, Filiot, and Khalimov (2021), ‘Church Synthesis on Register Automata over Linearly Ordered Data Domains’

Table 2.1.: Decidability status of the problems studied over $(\mathbb{D}, =, \mathbb{C})$. $\{D, N, U\}RA$ stands for deterministic (respectively non-deterministic, universal) register automata. The mention ‘tf’ means ‘test-free’. ‘Bounded synthesis’ is a short for ‘register-bounded synthesis’. As observed in Corollary 7.13, the register-bounded synthesis problem for deterministic register automata is in ExpTime if the target number of registers is larger than or equal to the number of registers of the specification, since it then reduces to the unbounded synthesis problem.

	DRA	NRA	URA	NRA ^{tf}
Bounded Synthesis	2ExpTime (Thm. 6.14)	Undecidable ($k \geq 1$) (Thm. 6.19)	2ExpTime ([36] and Thm. 6.14)	2ExpTime (Thm. 6.25)
Unbounded Synthesis	ExpTime-complete (Thm. 7.15)	Undecidable (Thm. 6.20)	Undecidable (Thm. 7.1)	Open

Table 2.2.: Decidability status of the problems studied over $(\mathbb{Q}, <, 0)$. The landscape is essentially the same, the objective is to list the corresponding theorems. Test-free specifications are omitted as they do not depend on the data domain.

	DRA	NRA	URA
Bounded Synthesis	2ExpTime (Thm. 6.16)	Undecidable ($k \geq 1$) (Thm. 6.19)	2ExpTime (Thm. 6.16)
Unbounded Synthesis	ExpTime-complete (Thm. 7.16)	Undecidable (Thm. 6.20)	Undecidable (Thm. 7.1)

2.4.7. Related Works

As already mentioned, register-bounded synthesis of register transducers is considered in [36] where it is shown to be decidable for universal register automata. We proved the main result [36, Corollary 3] independently. The problem was further studied in [35], which features a simplified proof and a more precise complexity analysis. They also consider a setting where the environment is also modelled as a register transducer, whose

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

[35]: Khalimov and Kupferman (2019), ‘Register-Bounded Synthesis’

number of registers is bounded, and show that it is decidable, although the corresponding game is not determined.

In [43], Faran and Kupferman consider the synthesis problem for word automata with arithmetic. These automata are equipped with a finite number of variables on which (as the name suggests) it can perform arithmetic operations and tests. Variables form the analogous of registers in our context; however, their valuation is fixed for the entire run, contrary to registers which can be updated by overwriting their content with the input data value, or by guessing a data value (when equipped with the guessing ability of [38]). Their main result is the decidability of the synthesis problem for non-deterministic word automata with arithmetic that are *semantically deterministic*, i.e. whose non-determinism can be solved once the valuation of the variables is known. More precisely, it can be solved in time polynomial in the size of the automaton, and exponential in the number of variables [43, Theorem 5]. As they show, this class of automata lies strictly between deterministic and non-deterministic word automata with arithmetic [43, Theorem 2 and 3]).

The synthesis problem over infinite alphabets is also considered in [44], in which data values represent identifiers and specifications are expressed in a formalism that is equivalent to universal register automata over data domains with equality. However, the class of implementations is very expressive: it allows for unbounded memory through a queue data structure. The synthesis problem is shown to be undecidable, already for the semi one-weak restriction. A sound but incomplete algorithm is given with an application to a robot waiter in a restaurant.

The class of test-free non-deterministic register automata, that allows to get decidability for register-bounded synthesis for a class of non-deterministic machines, corresponds to the one-way, nondeterministic version of the expressive transducer model of [45], which however does not consider the synthesis problem.

In [46], the authors develop a Counter-Example Guided Abstraction Refinement (CEGAR) procedure for specifications expressed in Temporal Stream Logic (TSL), that allows to define specifications over uninterpreted functions. The algorithm tries to generate an implementation that satisfies the specification for any possible interpretation of the functions. If it fails, it computes a counter-example, examines whether it is spurious (i.e. based on an inconsistent use of the uninterpreted functions) and accordingly refines its abstraction. This yields a semi-decision procedure for TSL, whose realisability problem is shown undecidable [46, Theorem 2]. The paper describes an application to the synthesis of a music player Android app.

Syntax-guided synthesis for programs over data domains has been studied in [47]: the synthesis algorithm takes as input a skeleton of the program to synthesise, and has to fill the gaps. As above, functions are uninterpreted, so the program should be correct for all possible interpretations. They show that the problem is 2-ExpTime-complete [47, Theorem 9] for the class of uninterpreted coherent programs (a restriction that was already known to yield decidable verification), while the general problem is undecidable [47, Theorem 1].

[43]: Faran and Kupferman (2020), ‘On Synthesis of Specifications with Arithmetic’

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

[44]: Ehlers, Seshia, and Kress-Gazit (2014), ‘Synthesis with Identifiers’

[45]: Durand-Gasselin and Habermehl (2016), ‘Regular Transformations of Data Words Through Origin Information’

[46]: Finkbeiner et al. (2019), ‘Temporal Stream Logic: Synthesis Beyond the Booleans’

[47]: Krogmeier et al. (2020), ‘Decidable Synthesis of Programs with Uninterpreted Functions’

Finally, as mentioned earlier, games with data values has been considered in [33]. The objective is given as a formula of the Logic of Repeating Values, which is a restriction of LTL with the freeze quantifier. The authors show that such games are undecidable, but that their single-sided version is decidable in 4ExpTime , when the logic is restricted to test remote repetitions in the past. Both restrictions are necessary, otherwise undecidability resurfaces. In a synthesis perspective, this can be expressed as decidability of the reactive synthesis problem for single-sided specifications expressed by LRV formulas. The Church synthesis problem is left open, as the memory structure of winning strategies is not examined.

[33]: Figueira, Majumdar, and Praveen (2020), ‘Playing with Repetitions in Data Words Using Energy Games’

The notion of remote repetition is analogous to the future operator in LTL: we ask that something eventually happens, but without setting a bound on the number of steps for it to happen. This is in contrast with repetitions that happen within j steps for a fixed j .

2.5. Good Things Come to Those Who Wait: Lifting the Synchronicity Requirement

2.5.1. ω -Computable Implementations

In some settings, it is not realistic to assume that the system reacts to its inputs in a synchronous way, as this might be too restrictive for implementations. Come back once more to our running example, and consider a machine that can brew and serve two coffees at a time. In this setting, when the machine receives an order, it might be more efficient to wait for a possible second order for a few time units (say k) before starting the operation. Correspondingly, specifications and implementations are modelled by asynchronous machines. More precisely, the specification is now given as a non-deterministic asynchronous transducer, which is allowed to output a (possibly empty) word everytime it reads a letter, and implementations consist in asynchronous sequential transducers, which consists in transducers that behave deterministically with regard to their input. Already in the finite alphabet case, this setting is undecidable [48, Theorem 17]. In the infinite alphabet case, as we demonstrate in this thesis it is already undecidable for synchronous specifications (Theorem 6.20).

[48]: Carayol and Löding (2015), ‘Uniformization in Automata Theory’

We instead consider ω -computable implementations. They consist in functions that are computable by some Turing machine M that has an infinite input $x \in \text{dom}(f)$, and must produce longer and longer prefixes of the output $f(x)$ as it reads longer and longer prefixes of the input x . Therefore, such a machine produces the output $f(x)$ in the limit. In the above example, the specification is ω -computable, as it suffices to wait for k times unit for deciding whether to start brewing a single coffee or two at a time. However, consider the case where the machine instead has to decide whether to enter sleep mode or not, and can only do so if no orders are made in the future (with no bound). Such a specification is not ω -computable, as this information depends on the infinite suffix of actions. Finally, to distinguish ω -computable implementations from finite-memory ones, consider the case where the machine accumulates orders, and only delivers them when some button ‘deliver’ is pushed. This specification is ω -computable by a machine that simply stores the orders in memory, but not by any deterministic transducer, since it necessarily has a bounded amount of memory.

We use the terminology of ω -computability to avoid the confusion the usual notion of computability, which we also use in our study.

2.5.2. Restricting to Functional Specifications

However, already in the finite alphabet setting, the problem of deciding if a specification given as some non-deterministic *synchronous* transducer is realisable by some ω -computable function is open. The particular case of realisability by ω -computable functions of universal domain (the set of all ω -words) is known to be decidable [49, Corollary 15]. In the asynchronous setting, the undecidability proof of [48, Theorem 17] can be easily adapted to show the undecidability of realisability of specifications given by non-deterministic (asynchronous) transducers by ω -computable functions.

[49]: Holtmann, Kaiser, and Thomas (2012), ‘Degrees of Lookahead in Regular Infinite Games’

In general, a specification is a relation from inputs to outputs. If this relation is a function, we call it functional. Due to the negative results just mentioned about the synthesis of ω -computable functions from non-functional specifications, we instead here focus on the case of functional specifications and address the following general question: given the specification of a function of data ω -words, is this function ‘implementable’, where we define implementable as being ω -computable by some Turing machine. Moreover, if it is implementable, then we want a procedure to automatically generate an algorithm that computes it. This raises another important question: how to decide whether a specification is functional? We investigate these questions for asynchronous register transducers, here called register transducers. This asynchrony allows for much more expressive power, but is a source of technical challenge.

2.5.3. Contributions

As for register automata, register transducers are parameterised by the data domain from which the data ω -words are built, along with the set of predicates which can be used to test those data values. We distinguish a large class of data domain for which we obtain decidability results for the later problem, namely the class of oligomorphic data domains [26, Section 3.2]. Briefly, they consist in a countable set $\mathbb{D}$ equipped with a finite set of predicates which satisfies that for all n , $\mathbb{D}^n$ can be partitioned into finitely many equivalence classes by identifying tuples which are equal up to automorphisms (predicate-preserving bijections). For example, any set equipped with equality is oligomorphic, such as $(\mathbb{N}, =)$. Importantly, $(\mathbb{Q}, <)$ is oligomorphic while $(\mathbb{N}, <)$ is not. However $(\mathbb{N}, <)$ is an interesting data set in and of itself. We also investigate non-deterministic register transducers over such data domain, using the fact that it is a substructure of $(\mathbb{Q}, <)$ which is oligomorphic. Our detailed contributions are the following:

[26]: Bojańczyk (2019), *Atom Book*

1. We first establish a general correspondence between ω -computability and the classical mathematical notion of continuity (for the Cantor distance) for functions of data ω -words (Theorems 11.2 and 11.3). This correspondence holds under a general assumption, namely the decidability of what we called the *next-letter problem*, which consists in finding the next data value which can be safely output knowing only a finite prefix of the input data ω -word, if it exists. We also show similar correspondences for more constrained ω -computability and continuity notions, namely Cauchy, uniform and m -uniform computability and continuity. In these correspondences,

the construction of a Turing machine ω -computing the function is effective.

2. We consider a general computability assumption for oligomorphic data domains, namely that they have decidable first-order satisfiability problem [26]. We call such data domains *decidable*. We then show that functions defined by non-deterministic register transducers over decidable oligomorphic data domains and over $(\mathbb{N}, <)$, have decidable next-letter problem. As a consequence (Theorems 12.39 and 12.66), we obtain that a function of data ω -words over a decidable oligomorphic data domains or over $(\mathbb{N}, <)$ that is definable by a non-deterministic register transducer is ω -computable if and only if it is continuous (and likewise for all ω -computability and continuity notions we introduce). This is a useful mathematical characterisation of ω -computability, which we use to obtain our main result.
3. As explained before, a register transducer may not define a function in general but a relation, due to non-determinism. Functionality is a semantical, and not syntactical, notion. We nevertheless show that checking whether a non-deterministic register transducer defines a function is decidable for decidable oligomorphic data domains. This problem is called the *functionality problem* and is a prerequisite to our study of ω -computability, as we assume specifications to be functional. We establish PSPACE-completeness of the functionality problem for register transducers over $(\mathbb{N}, <)$ (Theorem 12.64) and for oligomorphic data domains (Theorem 12.40) under some additional assumptions on the data domain that we call polynomial decidability. In short, it is required that the data domain has PSPACE-decidable first-order satisfiability problem.
4. Finally, we show (Theorem 12.41) that continuity of functions defined by register transducers over decidable (resp. polynomially decidable) oligomorphic data domains is decidable (resp. PSPACE-C). We also obtain PSPACE-completeness in the non-oligomorphic case $(\mathbb{N}, <)$ (Theorem 12.72). These results again hold for the stronger notion of uniform continuity (see also Theorem 12.68). As a result of the correspondence between ω -computability and continuity, we also obtain that ω -computability and uniform computability are decidable for functions defined by non-deterministic register transducers over decidable oligomorphic data domains, and PSPACE-complete for polynomially decidable oligomorphic data domains as well as $(\mathbb{N}, <)$. This is our main result and it answers positively our initial synthesis motivation.

[26]: Bojańczyk (2019), *Atom Book*

2.5.4. Related Works

We have already mentioned works related to the synthesis problem. We now give references to results on ω -computability and continuity. The notion of continuity with regards to the Cantor distance is not new, and for rational functions over finite alphabets, it was already known to be decidable [50, Proposition 4]. The approach of Prieur is to reduce continuity to functionality by defining, from a transducer T a transducer realising its topological closure. The latter is functional if and only if the former is continuous [50, Lemma 5.1]. We were able to extend this approach to almost all the cases we considered, except for transducers

A function is *rational* if it is computed by a non-deterministic one-way transducer.

[50]: Prieur (2002), 'How to decide continuity of rational functions on infinite words'

over $(\mathbb{N}, <)$ in the presence of guessing, so we chose a different proof strategy. The connection between continuity and ω -computability for functions of ω -words over a finite alphabet has recently been investigated in [51] for one-way and two-way non-deterministic transducers. Our results lift the case of one-way transducers from [51, Theorems 6 and 12] to data ω -words.

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

2.6. Outline

The contributions detailed in this manuscript are twofold. The first part is dedicated to the reactive and Church synthesis problems. Its first chapter exposes the landscape of what is known for the finite alphabet case. Chapter 4 focuses on register automata. Chapter 5 defines data automatic specifications, register transducers and the Church problem over data words, as well as the corresponding Church game. Chapter 6 tackles the register-bounded synthesis problem for universal (Section 6.2) and non-deterministic (Section 6.3) specifications over data domains $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$, as well as the test-free restriction (Section 6.3.2). The next chapter is dedicated to unbounded synthesis. It first presents the undecidability results that motivate the introduction of register-bounded synthesis (Theorems 6.20 and 7.1), followed by positive results in the case of a specification given by a deterministic automaton (Section 7.3). The part concludes with a discussion on the question of the domain of the implementation (total versus partial), that marks the decidability border in the infinite alphabet case (Chapter 8).

The second part is dedicated to the study of ω -computability and continuity for functions represented by non-deterministic asynchronous register transducers. The model is introduced in Chapter 10, which also establishes closure under composition for this class of functions. In Chapter 11 we expose the notion of ω -computability, and show that for transducers it is equivalent with continuity for the Cantor distance, and similarly for refined notions. The rest of the part is dedicated to the study of various data domains, where we settle the decidability status of functionality, ω -computability, continuity and their refined notions (Chapter 12). Section 12.1 is focused on data domains with the equality predicate. The next section generalises the study to $(\mathbb{Q}, <)$ and oligomorphic data domains. Finally, Section 12.3 tackles the case of $(\mathbb{N}, <)$, which is more involved as register transducers over this data domain do not exhibit a regular behaviour. The part concludes with a discussion on further extensions.

This part is followed by a general conclusion that links the two parts and draws perspectives for future research.

2.7. How to Read this Thesis?

Numbering Conventions Environment that correspond to hypotheses, i.e. definitions, notations and conventions are numbered together. Idem for those regarding conclusions, namely theorems, propositions, corollaries, lemmas, properties and observations. Besides, the numbering is according to chapters.

Link with Publications Part of the work that is presented here has been published in conference or journal papers. When this is the case, theorems are linked with the corresponding paper; the citation is highlighted in bold.

Kaobook This thesis has been written using the kaobook class. It features a wide margin, which contains notes, bibliographic references [52], captions of figures and sometimes the figures themselves. Side notes are not numbered so as not to impede reading, but they are aligned with the content they refer to. The rationale behind this class is that this document is probably read on a computer or a similar device, on which looking for the information at the bottom of the page (for footnotes[‡]) or at the end of the document (for references) is tedious.

Definitions and Notations Another feature of this work is the use of the knowledge package, which links any occurrence of a notion to the place where it is defined. This might appear contradictory with the previous one, which focuses on locality of information. The contradiction can however be overcome through the use of modern PDF readers, which display a preview of the targeted link, and hence of the definition of the notion. The thesis also contains an index of notations at 334.

Graphical Conventions An image is sometimes worth a thousand words[§], and many notions are illustrated through examples. Some proof ideas are also conveyed through figures. For readability, this document features a set of graphical conventions, which are introduced along the way and listed in Chapter A in the appendix. In particular, colours are used in figures as a way to convey information.

Roadmaps Reading order is a matter of taste (and of available time), so some efforts have been devoted to allowing a modular reading of the document, depending on the background and expectations of the reader. Besides the classical linear reading, here are various possible roadmaps:

The Quest for Equality This work started with the study of data domains with the equality predicate only. Register automata over these domains are pretty well behaved, which yields simpler constructions and proofs. The reader who is only interested in this case can skip the developments about richer structures (dense and discrete orders, as well as oligomorphic domains). Jumps to that effect are mentioned along the document.

All About Registers Chapter 4 is meant to be a primer on register automata. It was written with synthesis applications in mind, but contains the standard properties of the model and pointers to the existing literature.

The Two Facets of Synthesis Chapters 6 (Register-Bounded Synthesis of Register Transducers) and 7 (Unbounded Synthesis of Register Transducers) can be read independently. Each build on Chapters 3 to 5.

By contraposition, this sentence also means that theorems with no mention to existing publications are original to this document.

Notes in the margin contain remarks, definitions of secondary concepts, technical details and all material that is deemed ancillary.

[52]: Marotta (2020), *Example and documentation of the kaobook class*

There is also a bibliographic section at the end of the document, to which the links to references in the body refer.

Links are coloured in the default version of the document, but the author can provide a version with links hidden to the reader which finds it more suitable for their use.

Most figures are drawn using the wonderful TikZ package.

Colours are not used in the text, except for links, so that the document does not look like Elmer.

The reader is of course encouraged to invent their own.

[‡] One or two footnotes survived, for *ancillary* ancillary material like puns.

[§] This can be taken literally. Authors of conference-format articles might relate.

Patience is a Virtue Part II is dedicated to asynchronous transducers, and can mostly be read independently. It only assumes basic knowledge about register automata, which can be found in Chapter 4.

Part I.

REACTIVE SYNTHESIS

Reactive Synthesis over Finite Alphabets

3.

This chapter introduces the reactive synthesis problem and presents existing results which hold in the finite alphabet case and that prove useful for our study of the generalisation to infinite alphabets in the next chapters. Reactive synthesis is first presented as a specialisation of the more general uniformisation problem, following the spirit of [5], so as to ease the exposition of the asynchronous case (cf Part II). The Church synthesis problem is then formulated in an independent way (Section 3.4) to allow for a modular reading of the document. The reader who is more familiar with the game-theoretic point of view can start with Section 3.5.3.

[5]: Thomas (2009), ‘Facets of Synthesis: Revisiting Church’s Problem’

Summary

3.1. Relations and Functions	46
3.2. Uniformisation and Program Synthesis	46
3.2.1. Uniformisation	46
3.2.2. Program Synthesis	47
3.3. The Reactive Synthesis Problem	48
3.3.1. Finite and Infinite Words	48
3.3.2. Reactive Synthesis	49
3.4. The Church Synthesis Problem	52
3.4.1. Logics for Specifications	53
First-Order Logic	53
Monadic Second-Order Logic	53
Linear Temporal Logic	55
3.4.2. Discrete Systems	56
3.4.3. Automata	57
Emptiness Problem	61
Equivalence with MSO	61
Automatic Specifications	62
3.4.4. Programs Computable by Finite-State Machines	63
3.4.5. Statement of the Church Synthesis Problem	64
3.5. Games for Reactive Synthesis	65
3.5.1. Games	65
Parity Games	67

3.5.2. Games and Automata	68
Acceptance as a Game and Alternation	68
Games over Specification Automata	69
3.5.3. The Church Game	70

3.1. Relations and Functions

Let X and Y be two sets. A *binary relation*, or *relation*, for short over X and Y is a subset $R \subseteq X \times Y$. Its *domain* is $\text{dom}(R) = \{x \in X \mid \exists y \in Y, (x, y) \in R\}$. We say that R has *total domain* if $\text{dom}(R) = X$. For a set $A \subseteq X$, the *image* of A by R is $R(A) = \{y \in Y \mid \exists a \in A, (a, y) \in R\}$. We say that $\text{Im}(R) = R(X)$ is the image of R .

When we consider k -ary relations, we explicitly state it, so that no ambiguity arises.

A relation R is *functional* if for all $x \in \text{dom}(R)$, $R(\{x\})$ is a singleton. A *partial function* from X to Y , denoted $f : X \rightarrow Y$, is a functional relation. For $x \in \text{dom}(f)$, we then denote $f(x)$ the unique element of $f(\{x\})$. We say that f is *total* if it has total domain, i.e. $\text{dom}(f) = X$. The set of total functions from X to Y is denoted Y^X . As usual, functions are by default total.

3.2. Uniformisation and Program Synthesis

3.2.1. Uniformisation

Definition 3.1 (Uniformisation) Let $R \subseteq A \times B$ be a relation, where A and B are two sets, and let $f : A \rightarrow B$ be a partial function. We say that f *uniformises* R , or that f is a *uniformiser* for R , when (see also Figure 3.1):

- (i) $\text{dom}(f) = \text{dom}(R)$
- (ii) $f \subseteq R$, i.e. for all $x \in \text{dom}(f)$, $(x, f(x)) \in R$

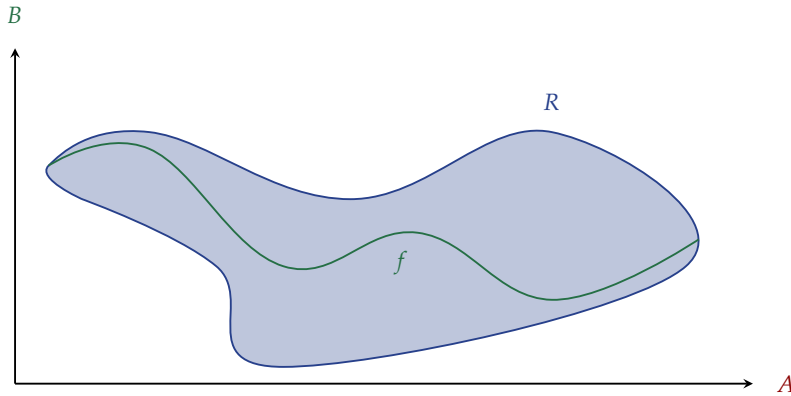


Figure 3.1.: A function $f : A \rightarrow B$ which uniformises the relation $R \subseteq A \times B$. By convention, input is coloured red, output and functions are in green, and relations in blue.

We then say that a class $\mathcal{R}$ of relations admits the *uniformisation property* if every relation $R \in \mathcal{R}$ admits a uniformiser $f \in \mathcal{R}$, i.e. a functional relation in the same class that uniformises it. More generally, the field of uniformisation theory is concerned with the following problem: given a class $\mathcal{R}$ of relations and a class $\mathcal{F}$ of functions, does any relation $R \in \mathcal{R}$ admit a uniformisation $f \in \mathcal{F}$? Here, we introduce the decision variant of

Remark 3.1 The uniformisation property is a weak form of the axiom of choice.

the problem, namely the *uniformisation problem*, which provides a general setting in which to express synthesis problems:

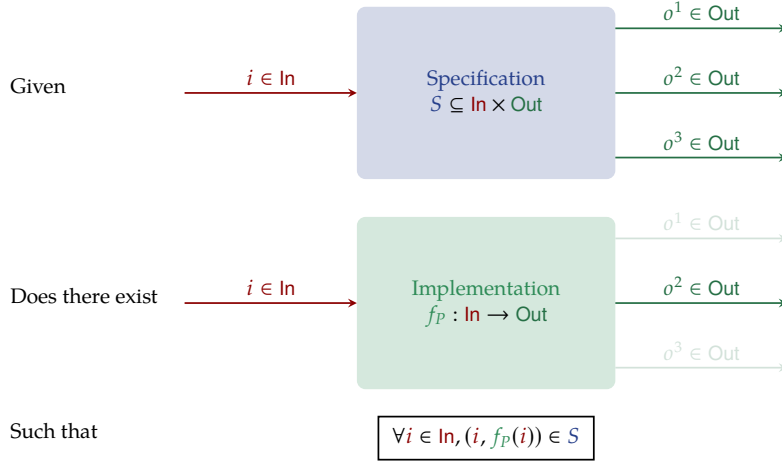
Problem 3.1: UNIFORMISATION PROBLEM

Input: A relation $R \in \mathcal{R}$
Output: A finitely represented function $f \in \mathcal{F}$ which uniformises R if it exists
 No otherwise

Remark 3.2 When $\mathcal{R}$ is a class of relations that are functional, the uniformisation problem is actually a membership problem.

3.2.2. Program Synthesis

Let In and Out be (possibly infinite) sets of *inputs* and *outputs*. *Program synthesis* asks, given a *specification* $S \subseteq \text{In} \times \text{Out}$ which relates inputs in In with *admissible outputs* in Out , to generate, if it exists, a program P which computes a function $f_P : \text{dom}(S) \rightarrow \text{Out}$ which for each input $i \in \text{dom}(S)$, selects an admissible output $f_P(i) \in \text{Out}$ such that $(i, f_P(i)) \in S$. We then say that P is an *implementation* of S (cf Figure 3.2).



A *program* is a deterministic Turing machine. It takes as *input* an $i \in \text{In}$, encoded as $\text{enc}(i) \in \{0, 1\}^*$ (Turing machines, like computers, only operate over sequences of bits). We say that it *outputs* $o \in \text{Out}$ if it halts and the content of its tape at this moment is $\text{enc}(o) \in \{0, 1\}^*$ (note that we assume that the encoding enc is injective). It *computes* the function f_P which, to each input $i \in \text{In}$ such that the machine halts, associates the corresponding output $o \in \text{Out}$.

Figure 3.2.: The program synthesis problem. S is a relation. f_P is a function computed by a program P .

Such problem can be reformulated as a uniformisation problem from $\mathcal{S}$ the class of relations over $\text{In} \times \text{Out}$ to $\mathcal{F}$ the class of computable functions from In to Out .

Example 3.1 (Scheduler) Consider a set J of *jobs*. A *precedence relation* $P \subseteq J \times J$ over J is a partial order. A sequence $t_1 \dots t_n \in J^*$ is *consistent with* P whenever for all $0 \leq k < l \leq n$, $(t_k, t_l) \in P$ or t_k and t_l are incomparable.

Now, one can define a specification S_{sch} such that for any subset $J' \subseteq J$ of jobs and any precedence relation P over J , and for any sequence T of jobs, $((J', P), T) \in S_{\text{sch}}$ whenever T is a scheduling of J' which is consistent with P .

Then, implementations of S_{sch} are exactly schedulers for the set of jobs J .

Example 3.2 (Sorting) Note that the program synthesis problem is already interesting in the case where the specification is functional.

Consider for instance the following specification $S_{\text{sort}} \subseteq \mathbb{N}^* \times \mathbb{N}^*$:

x and y are *incomparable* for a partial order relation P whenever neither $(x, y) \in P$ nor $(y, x) \in P$.

Given a set A , a sequence T is a *scheduling* of a set A if it is a permutation of A .

A list $b = [b_0, \dots, b_m]$ is a *permutation* of a list $a = [a_0, \dots, a_n]$ if they have the same length ($m = n$) and there exists a bijection $\pi : \{0, \dots, n\} \rightarrow \{0, \dots, n\}$ such for all $0 \leq i \leq m = n$, $b_i = a_{\pi(i)}$.

A list $a = [a_0, \dots, a_n]$ is *increasing* if its elements are in increasing order, i.e. for all $0 \leq i < j \leq n$, $a_i \leq a_j$.

$$S_{\text{sort}} = \left\{ \left(\begin{array}{c} [a_0, \dots, a_n], \\ [b_0, \dots, b_n] \end{array} \right) \mid \begin{array}{l} [b_0, \dots, b_n] \text{ is a permutation of} \\ [a_0, \dots, a_n] \text{ and} \\ [b_0, \dots, b_n] \text{ is increasing} \end{array} \right\}$$

Implementations of S are exactly sorting algorithms over $\mathbb{N}$.

Remark 3.3 (ω -computability) Note that for input and output sets whose elements cannot be finitely represented, and in particular for infinite words, the notion of computability needs to be extended to what we call ω -computability. Informally, a function is ω -computable if there exists a Turing machine which, on reading longer and longer prefixes of the input, outputs longer and longer prefixes of the output.

With no further restrictions to the class $\mathcal{S}$ of specification and to the class $\mathcal{I}$ of implementations, the program synthesis problem is too general to be decidable [5]. However, there are classes of specifications and implementations for which it is both interesting and decidable. One of them is the setting of reactive synthesis. The specification is an ω -regular language describing the behaviour of a non-terminating program, given for instance as an MSO formula. Target implementations are synchronous programs, i.e. programs which process their input bit by bit and produce the corresponding output in an on-line mode. We are more specifically interested in the Church synthesis problem (Section 3.4), which further restricts to programs that only use a finite amount of memory. In short, the target class of implementations is the class of synchronous transducers.

We now introduce the necessary notions to formalise reactive synthesis and the Church synthesis problem. We refer to the book [53] for a more thorough introduction to those concepts.

3.3. Words, Languages and the Reactive Synthesis Problem

In this work, we are concerned with programs which process their input bit by bit, and produce their output in a reactive way. The input is thus modelled as a word, more precisely an infinite word since reactive programs are non-terminating.

3.3.1. Finite and Infinite Words

Words over a Finite Alphabet An *alphabet* is a finite set Σ . Elements of Σ are called *letters*. A *finite word* is a finite sequence of letters $u = a_0 \dots a_{n-1} \in \Sigma^*$ for some integer $n \geq 0$. Its *length* is then $|u| = n$. The *empty word*, of length 0, is denoted ε . An *infinite word*, or ω -word, is an infinite sequence of letters $x = a_0 a_1 \dots \in \Sigma^\omega$. By convention, its length is $|x| = \infty$. We let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. For a word $w \in \Sigma^\infty$ and for $0 \leq i < |w|$, we denote its i -th letter by $w[i] = a_i$. For $0 \leq i \leq j < |w|$, we let $w[i:j] = a_i \dots a_j$. Finally, given a word $w \in \Sigma^*$ and a letter $a \in \Sigma$, we let $|w|_a = |\{i \in \{0, \dots, |w| - 1\} \mid w_i = a\}|$ be the number of *occurrences* of the letter a in w .

A *language* over alphabet Σ is a subset $L \subseteq \Sigma^*$. An ω -*language* is a subset $L \subseteq \Sigma^\omega$.

The notion of computability and ω -computability are formally defined and studied over infinite alphabets in Part II (Section 11.1). In this part, we only consider computability by synchronous programs.

[5]: Thomas (2009), ‘Facets of Synthesis: Revisiting Church’s Problem’

The notion of synchronous program is defined in Section 3.3.2.

[53]: Grädel, Thomas, and Wilke (2002), *Automata, Logics, and Infinite Games: A Guide to Current Research* [outcome of a Dagstuhl seminar, February 2001]

In this document, we often confuse letters with words of length 1.

L_{af} , its complement $L_{a\infty}$, as well as L_{aB} and L_{halt} are all ω -languages. However, as we will see, they have quite different structures: the first two are ω -regular, i.e. they can be recognised by a finite-state machine called an ω -automaton (cf Definition 3.5 and Example 3.15), while L_{aB} cannot. The last one is even more complex, and it is actually outside the Chomsky hierarchy. Indeed, it is not even recursively enumerable, since the halting problem is undecidable.

Example 3.3 The following sets are ω -languages over the alphabet $\Sigma = \{a, b\}$:

- ▶ $L_{af} = \{w \in \Sigma^\omega \mid w \text{ contains finitely many } a\}$
- ▶ $L_{a\infty} = \{w \in \Sigma^\omega \mid w \text{ contains infinitely many } a\} = L_{af}^c$
- ▶ L_{par} consists in sequences of bits separated by \$ where the last bit of each sequence indicates whether the number of 1 in the sequence is odd. Formally, $L_{\text{par}} = \{w_0b_0\$w_1b_1\$ \dots \in (\{0,1\}^*\$)^\omega \mid \text{for all } i \in \mathbb{N}, b_i = |w_i|_1 \bmod 2\}$.
- ▶ $L_{aB} = \{ba^{k_0}ba^{k_1} \dots ba^{k_i}b \dots \mid \exists B \in \mathbb{N}, \forall i \in \mathbb{N}, k_i \leq B\}$
- ▶ Let an encoding of Turing machines which, to each machine M uniquely associates a finite word $w_M \in \Sigma^*$ (such an encoding exists as Turing machines form a countable set). Then, $L_{\text{halt}} = \{w_M\$^\omega \mid M \text{ halts}\}$ is also an ω -language.

Product and Pointwise Application Given two alphabets Σ and Γ and two words $u \in \Sigma^\omega$ and $v \in \Gamma^\omega$ such that $|u| = |v|$, we define their product $u \otimes v$ as, for all $0 \leq i < |u|$, $(u \otimes v)[i] = (u[i], v[i])$.

By convention, two ω -words have the same length ∞

Functions $f : \Sigma \rightarrow A$ (for some set A) can be naturally extended *pointwise* to $(\omega$ -)words over Σ : for an ω -word $w \in \Sigma^\omega$, we let $f(w) \in A^\omega$ be the ω -word such that for all $i \in \mathbb{N}$, $f(w)[i] = f(w[i])$. The notion is defined analogously for finite words.

Concatenation and Prefix Order The *concatenation* of a finite word u with a word v is the word $w = u \cdot v$ of length $|u| + |v|$ (or ∞ if v is infinite) where for all $0 \leq i < |u|$, $w[i] = u[i]$ and for all $0 \leq j < |v|$, $w[|u| + j] = v[j]$. For two words u and v , we say that u is a *prefix* of v , written $u \leq v$, if there exists a word $w \in \Sigma^\omega$ such that $u \cdot w = v$. If $w \neq \varepsilon$, we say that u is a *strict prefix* of v , written $u < v$. Note that the prefix relation induces a partial order over words.

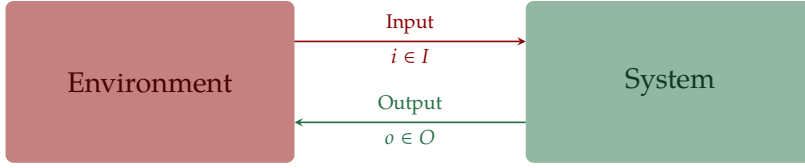
Encoding of Relations by Languages Over infinite words, relations can be encoded by languages in the following way: given a pair $(x, y) \in I^\omega \times O^\omega$ of ω -word over respective alphabets I and O , we define their *interleaving* $\langle x, y \rangle \in (IO)^\omega$ by $\langle x, y \rangle = x[0]y[0]x[1]y[1] \dots$. Given a relation $R \subseteq I^\omega \times O^\omega$, we denote $\langle R \rangle \subseteq (IO)^\omega = \{\langle x, y \rangle \mid (x, y) \in R\}$ its *interleaving*. Conversely, any infinite word $w \in (IO)^\omega$ uniquely encodes the pair $(x, y) \in I^\omega \times O^\omega$, where $x = w[0]w[2] \dots$ is the concatenation of even letters and $y = w[1]w[3] \dots$, of odd letters. We denote $\rangle w \langle = (x, y)$, and extend the notation to languages $L \subseteq (IO)^\omega$, which encode relations $\rangle L \langle \subseteq I^\omega \times O^\omega$.

3.3.2. Reactive Synthesis

Consider a setting in which a *system* continuously interacts with an *environment* (cf Figure 3.3). The environment, which can for instance model the user of the system, provides an input letter from an input alphabet I , to which the system responds with an output letter from some output alphabet O , and so on *ad infinitum*. The input set is thus $\text{In} = I^\omega$; correspondingly, the output set is $\text{Out} = O^\omega$. Note that we assume that

Most of our results also apply to the finite word setting, as one can specify that at some point, the environment ceases to provide input and only inputs meaningless data, modelled as \$. However, the reactive synthesis setting only provides coarse-grained tools to express the relation between inputs and outputs, and focuses on the behaviour of the system rather than on its computational aspect.

their interaction never terminates: the reactive setting shifts the focus from *terminating computation* to *interaction*.



Now, the behaviour of the system is stated using a specification

$$S \subseteq I^\omega \times O^\omega$$

Example 3.4 Consider a safety-critical system, namely a coffee machine. It interacts with one or more coffee drinkers, modelled as the environment. The environment provides input by pressing buttons, and the machine reacts by conducting operations such as pouring coffee, milk, sugar, etc.

A typical specification is that each button corresponds to the right command, e.g. produces an espresso when an espresso is ordered, which would correspond to a property like ‘If ask_espresso is input, then the sequence of events which compose the action of delivering an espresso should appear later in the output’. This property can be expressed in a logical formalism (cf Section 3.4.1), e.g. a temporal logic [12], or using a finite-state machine that recognises it, called an ω -automaton.

If money has to be input, one can also specify that it only delivers the brewage when the correct amount has been put. If the amount of money that one can input is not bounded, such specification requires an expressive specification language to be expressed, as properties that involve counting cannot be recognised by finite-state machines (in other words, they are not ω -regular).

Finally, one might need to express properties about the delay between the input and the corresponding coffee output. This can be done in a coarse-grained manner, by modelling time through the number of operations between the input and the output: ‘there are at most k operations between ask_espresso and deliver_espresso’. This kind of specification can be expressed in a logic like LTL [12] or MSO [54]. To get a finer-grained modelling, one can resort to expressive formalisms like timed temporal logics [55], which allow to directly reason about time, and express ‘there are at most t time units between event a and event b ’. As always, expressiveness however comes at a price, and problems that are decidable for well-behaved models like ω -automata (cf Definition 3.5) might be undecidable for more expressive ones.

Example 3.5 (Request-Grant) We now focus more precisely on the first aspect of the above example, i.e. the specification that every time a coffee is ordered, it is eventually delivered. This corresponds to the inescapable ‘request-grant’ example, traditionally introduced in the setting of a *server* receiving *requests* (req) from *clients*, that it may *grant* (grt). It is then required that every request it receives must eventually be granted. Now, the specification can be formalised as follows, where $I = \{\text{req}, \text{idle}\}$ and

Figure 3.3.: A reactive system in *interaction* with an environment. Their interaction yields the infinite word:

$$i_0 o_0 i_1 o_1 \dots \in (IO)^\omega$$

or, equivalently, a pair:

$$(i_0 i_1 \dots, o_0 o_1 \dots) \in I^\omega \times O^\omega$$

Depending on the level of details of the modelling, delivering an espresso can be directly expressed by an event deliver_espresso, or decomposed in put_cup pour_espresso display_coffee_ready, and each of those events might even be decomposed further.

[12]: Pnueli (1977), ‘The Temporal Logic of Programs’

In practice, this amount is however bounded, if only by the capacity of the money receiver.

[12]: Pnueli (1977), ‘The Temporal Logic of Programs’

See also Section 3.4.1 (Linear Temporal Logic).

[54]: Büchi (1962), ‘On a decision method in restricted second order arithmetic’ cf Section 3.4.1 (Monadic Second-Order Logic).

[55]: Bouyer et al. (2017), ‘Timed Temporal Logics’

Here, the coffee machine plays the role of the server, and the clients are coffee drinkers.

$O = \{\text{grt}, \text{idle}\}$:

$$S_{\text{rg}} = \left\{ (i, o) \in I^\omega \times O^\omega \mid \begin{array}{l} \text{for all } j \in \mathbb{N}, \text{ if } i[j] = \text{req} \\ \text{then there exists } k \geq j, o[k] = \text{grt} \end{array} \right\}$$

In our setting, it is easier to manipulate specifications encoded as ω -languages (cf Section 3.3.1). S_{rg} is encoded as:

$$L_{\text{rg}} = \left\{ w \in (IO)^\omega \mid \begin{array}{l} \text{for all } j \in \mathbb{N}, \text{ if } w[j] = \text{req} \\ \text{then there exists } k > j, w[k] = \text{grt} \end{array} \right\}$$

The target implementations are then synchronous (also known as reactive) programs. Such programs produce their outputs in an on-line mode: the k -th letter of output is produced immediately after the k -th letter of input has been read.

More precisely, consider a program P which takes as input the finite sequence of input letters which have been read so far and reacts by producing an output letter, computing a partial function $f : I^* \rightarrow O$. We define its ω -behaviour as the (non-terminating) program Ω_P which, on reading an infinite word $x = x_0x_1 \cdots \in I^\omega$, produces the infinite sequence of outputs $\Omega_f(x) = f(x_0)f(x_0x_1) \cdots \in O^\omega$, processing its input bit by bit.

The notion of *synchronous program* is then the dual notion: a non-terminating program Q is synchronous if there exists a (terminating) program P such that Q is the ω -behaviour of P .

Remark 3.4 This notion is to be related with the notion of causal function [5, 8]. A function $g : I^\omega \rightarrow O^\omega$ is *causal* if for all $k \in \mathbb{N}$, the k -th output bit only depends on the first k input bits. If g is ω -computable (by some non-terminating program), then g is causal if and only if it can be computed by a synchronous program Q .

Example 3.6 Consider again the request-grant example (Example 3.5). Without further requirements, the specification S_{rg} is satisfied by the synchronous program which ignores its input and produces a grant at every step. It can be defined directly, or as the ω -behaviour of a program which ignores its input and outputs grt, implementing the constant function $f_{\text{grt}} : i \in I^* \mapsto \text{grt}$.

If one additionally specifies that there are no spurious grants* through a specification S_{nsgr} , then the overall specification $S_{\text{rg}} \cap S_{\text{nsgr}}$ can be implemented by a synchronous program which immediately grants a request, and otherwise does not grant. It is the ω -behaviour of any program P computing $f_{\text{rg}} : \begin{cases} i_{\text{req}} \mapsto \text{grt} \\ i_{\text{idle}} \mapsto \text{idle} \end{cases}$ (where $i \in I^*$). $\square$

Formally, the *reactive synthesis problem* is then defined as the uniformisation problem from the class of relations over $I^\omega \times O^\omega$ to the class of functions computed by synchronous programs i.e. of ω -computable causal functions. Thus, it can be stated as follows:

Problem 3.2: REACTIVE SYNTHESIS PROBLEM

The ω -behaviour Ω_P of P can thus be defined as the following algorithm:

Algorithm 1: Ω_P

Input: $w \in I^\omega$
for $i = 0$ **to** $+\infty$ **do**
 $\lfloor$ output $P(w[:i])$;

In other words, $Q = \Omega_P$, where equality is *intentional*, i.e. we really ask that they are the same object, and not simply that they behave the same (the latter is known as *extensional equality*).

[5]: Thomas (2009), ‘Facets of Synthesis: Revisiting Church’s Problem’

[8]: J.R. Büchi and L.H. Landweber (1969), ‘Solving sequential conditions finite-state strategies’

Formally, g is causal if there exists a function $f : I^* \rightarrow O$ such that for all input words $x \in I^\omega$ and for all positions $k \in \mathbb{N}$, $f(x)[k] = g(x[0:k])$.

* Otherwise, what you get is better described as a coffee *fountain*.

Input: A specification $S \subseteq I^\omega \times O^\omega$
Output: A synchronous program Q which uniformises S
 (i.e. $f_Q \subseteq S$) if it exists
 No otherwise

In the field of synthesis, when R is uniformised by a function f which can be computed, we also say that f *realises* R . Subsequently, R is said to be *realisable* if such f exists. In the following, we favour this terminology as the focus is set on synthesis problems.

Remark 3.5 In general, we moreover assume that the system must be able to react to any input, i.e. we target implementations with total domain ($\text{dom}(f_Q) = I^\omega$). This assumption implies that if $\text{dom}(S) \subsetneq I^\omega$, S is not realisable. It is benign in the finite alphabet case for the classes of specification that we consider since it is always possible to complete the specification by allowing any behaviour on the complement of the domain. More precisely, given a specification $S \subseteq I^\omega \times O^\omega$, one instead considers the specification $S' = S \sqcup (\text{dom}(S)^c \times O^\omega)$, which can be expressed in the usual formalisms as they enjoy the adequate closure properties.

In this document, we write $A \sqcup B$ for the union of A and B when they are disjoint, to highlight this fact.

This is not the case anymore in our setting since the formalism we consider, namely register automata, is not closed under complement (Property 4.18). Allowing implementations to have a domain which is not total implies going beyond the decidability border (Theorem 8.4) since it is undecidable whether the domain of a specification recognised by a deterministic register automaton is total (Theorem 8.1). See Chapter 8 for a discussion on this matter.

The field of reactive synthesis then aims at studying such problem for various classes of specifications. In 1957, Church laid the foundations of the field by introducing what is now known as the Church synthesis problem, in which he targets implementations computable by finite-state machines.

3.4. The Church Synthesis Problem

In his seminal paper [4], Church considers, among others, the setting where the specification $S \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$ is given as a monadic second-order formula over the integers with the successor relation, $\text{MSO}(\mathbb{N}, +1)$.

[4]: Church (1957), ‘Applications of recursive arithmetic to the problem of circuit synthesis’

$\text{MSO}(\mathbb{N}, +1)$ is also known as SIS for ‘second-order theory with one successor’.

Regarding the target implementation, he asks that it is not only synchronous, but that it can be computed by a *circuit*, i.e. a finite-state machine. A minimal-concept definition is that there is a finite-state automaton A which, given an input $w \in I^*$, decides whether the next output bit is 1 or 0, depending on whether $w \in L(A)$ or $w \notin L(A)$. The reader who is familiar with transducer theory can conceive it in terms of synchronous sequential transducers, and we formulate it as such in Section 3.4.5. A game-theoretic statement of the problem is also available in Section 3.5.3.

Synchronous sequential transducers are an extension of Mealy machines to the setting of infinite words.

The motivation behind restricting to circuits is they can be directly implemented on a chip, without requiring the full power of a computer. Indeed, the main application of reactive synthesis is *embedded systems*.

We now introduce the notions necessary for a precise statement of the problem.

3.4.1. Logics for Specifications

First-Order Logic

As a warmup, we present first-order logic, although the focus is mainly set on monadic second-order logic (cf the next section). First-order formulas (FO for short) consist in logical formulas $\varphi(x, y, \dots)$ with *first-order variables* $x, y, \dots$ which stand for elements of the domain. We consider the first-order theory with order $\text{FO}(\mathbb{N}, <)$. Thus, *atomic formulas* are of the form $x = y$ and $x < y$.

Formulas are then built with disjunction ($\vee$) and negation ($\neg$), as well as existential ($\exists$) quantification over first-order variables.

FO formulas are generated by the following grammar:

$$\varphi ::= x = y + 1 \mid \forall x. \varphi \mid \varphi \vee \varphi \mid \neg \varphi$$

where x and y are first-order variables.

Satisfaction of a sentence φ by an ω -word, written $w \models \varphi$ is defined in the expected way (see, e.g. [53, Part VI], which describes the semantics of the more expressive MSO logic).

Note that conjunction $\varphi \wedge \psi$ can be derived with De Morgan's laws, $\exists x. \varphi := \neg(\forall x. \neg \varphi)$, and $x = y$ can be derived as $\neg(x < y) \wedge \neg(y < x)$. Then $x = y + 1$ is expressed as $y < x \wedge \neg(\exists z. y < z \wedge z < x)$.

Over $\mathbb{N}$ with the order predicate, we know that satisfiability is decidable (cf Theorem 3.2 for the more general case of MSO).

Theorem 3.1 *The satisfiability problem of $\text{FO}(\mathbb{N}, <)$ is decidable.*

First-order logic however has a very high complexity: its satisfiability problem over finite and infinite words has a non-elementary lower bound [56, Theorem 5.2].

Monadic Second-Order Logic

Monadic Second-Order Logic (MSO for short) consists in an extension of First-Order logic with set predicates. MSO formulas consist in logical formulas $\varphi(x, y, \dots, X, Y, \dots)$ with first-order variables $x, y, \dots$ which stand for elements of the domain, and *second-order variables* $X, Y, \dots$ ranging over sets of elements. We consider the second-order theory with one successor $\text{MSO}(\mathbb{N}, +1)$. Thus, *atomic formulas* are of the form $x = y$, $x = y + 1$, $x < y$ and $x \in X$. Here, we operate over words, so we chose to explicitly add unary predicates $\sigma(x)$ for all $\sigma \in \Sigma$, where $\sigma(x)$ means that the letter at position $x \in \mathbb{N}$ is σ . Note that this is not the case in Church's setting, where both the input and output alphabet are $I = O = \{0, 1\}$ and letters are encoded.

Formulas are again built with disjunction ($\vee$) and negation ($\neg$), as well as universal quantification ($\forall$) over first-order variables *and* second-order

We use parentheses in our examples for clarity, but this is only syntactic sugar.

[53]: Grädel, Thomas, and Wilke (2002), *Automata, Logics, and Infinite Games: A Guide to Current Research* [outcome of a Dagstuhl seminar, February 2001]

[56]: Stockmeyer (1974), 'The Complexity of Decision Problems in Automata Theory and Logic'

Formulas $x = y$ and $x < y$ can be derived from $x = y + 1$ (see Example 3.7).

In Church's setting, an ω -word $w \in \{0, 1\}^\omega$ is encoded by the set of positions whose bit is equal to 1. Larger alphabets can then be simulated through binary encoding. See [5]: Thomas (2009), 'Facets of Synthesis: Revisiting Church's Problem' for more details.

variables (respectively lowercase and uppercase letters). Conjunction ($\wedge$) and existential quantification ($\exists$) are derived.

Correspondingly, MSO formulas are generated by the following grammar:

$$\varphi ::= x = y + 1 \mid x \in X \mid \sigma(x) \mid \forall x.\varphi \mid \forall X.\varphi \mid \varphi \vee \varphi \mid \neg\varphi$$

where x, y are first-order variables, X is a second-order variable and $\sigma \in \Sigma$.

Satisfaction of a sentence φ by an ω -word, written $w \models \varphi$ is defined in the expected way.

Example 3.7 (Order) The formula

$$\varphi_{\leq}(x, y) := \forall X.(x \in X \wedge (\forall z.(z \in X \Rightarrow (z + 1 \in X)) \Rightarrow y \in X$$

states that any set X which contains x and that is closed under taking the successor contains y . In other words, it means that $x \leq y$.

A formula with no free variables is called a *sentence*. Then, a sentence φ defines the ω -language $L_{\varphi} = \{w \in \Sigma^{\omega} \mid w \models \varphi\}$.

Example 3.8 Consider again the ω -languages of Example 4.7.2:

- $L_{af} = \{w \in \Sigma^{\omega} \mid w \text{ contains finitely many } a\}$ can be defined by $\varphi_{af} := \exists x.\forall y.y > x \Rightarrow \neg a(y)$.
- $L_{a\infty} = \{w \in \Sigma^{\omega} \mid w \text{ contains infinitely many } a\}$ can be defined by $\varphi_{a\infty} := \neg\varphi_{af}$.
- $L_{par} = \{w_0b_0\$w_1b_1\$ \dots \in (\{0,1\}^*\$)^{\omega} \mid \text{for all } i \in \mathbb{N}, b_i = |w_i|_1 \bmod 2\}$ is also recognisable by an MSO formula. The easiest way to prove this is to show that L_{par} is ω -regular by exhibiting an ω -automaton which recognises it (cf Example 3.15), as MSO recognises ω -regular languages (Theorem 3.6).

But let us not get ahead of ourselves: we now establish the result in a direct way. We only give an intuition; details can be found in Example B.1 in Section B.1.1. One first defines a formula $\varphi_{\$}(X)$ which holds whenever the set X is a set of positions between two $\$$. Then, $\varphi_{\$}$ can be refined so as to characterise sets of positions between two $\$$ which are labelled 1. Testing whether a set contains an even number of elements can be expressed in MSO. Finally, one can state that a position is the last of a sequence by asking that the next position is labelled $\$$. Putting it together, one obtains the sought formula.

- $L_{aB} = \{ba^{k_0}ba^{k_1} \dots ba^{k_i}b \dots \mid \exists B \in \mathbb{N}, \forall i \in \mathbb{N}, k_i \leq B\}$ cannot be defined by an MSO formula [59].
- $L_{halt} = \{w_M\$^{\omega} \mid M \text{ halts}\}$ is not recognisable by an MSO formula either, as its membership problem is undecidable.

Example 3.9 (Interleaved alphabets) Over alphabet $\Sigma = I \cup O$, consider

$$\varphi_{IO} := \forall x.(I(x) \Leftrightarrow O(x+1)) \wedge (\text{first}(x) \Rightarrow I(x))$$

where $I(x)$ is a short for $\bigvee_{i \in X} i(x)$ (similarly for $O(x)$), and $\text{first}(x) := \neg \exists y.x = y + 1$ means that x is the first position. Then, $L_{\varphi_{IO}} = (IO)^{\omega}$.

As we can encode relations over ω -words by ω -languages, an MSO sentence φ over alphabet $\Sigma = I \cup O$ defining a language $L_{\varphi} \subseteq (IO)^{\omega}$ can

$\exists X.\varphi$ can be derived as $\neg \forall X.\neg\varphi$.

Again, we do not define MSO satisfaction formally, as we do not use it in our study. See Part VI of [53], and in particular the chapter [57]: Weyer (2001), ‘Decidability of SIS and S2S’ for a more thorough introduction to MSO. Another good presentation can be found in [58]: Comon et al. (2007), *Tree Automata Techniques and Applications*.

$\varphi \wedge \psi$ is a short for $\neg(\neg\varphi \vee \neg\psi)$, $\varphi \Rightarrow \psi$ abbreviates $(\neg\varphi) \vee \psi$, and $z + 1 \in X$ is meant to be understood as $\forall t.t = z + 1 \Rightarrow t \in X$.

As formulas can contain the ‘=’ symbol, we use the ‘:=’ symbol when we define them, for readability. Its intended semantics is of course equality.

[59]: Bojańczyk and Colcombet (2006), ‘Bounds in ω -Regularity’

L_{aB} is indeed not ω -regular (cf Theorem 3.6).

$\varphi \Leftrightarrow \psi$ is a short for $\varphi \Rightarrow \psi \wedge \psi \Rightarrow \varphi$.

As witnessed by Example 3.9, one can express in MSO that words belong to $(IO)^{\omega}$.

be seen as defining the relation:

$$R_\varphi = \rangle_{L_\varphi} \langle = \{(i_0 i_1 \dots, o_0 o_1 \dots) \in I^\omega \times O^\omega \mid i_0 o_0 i_1 o_1 \dots \models \varphi\}$$

Example 3.10 The request-grant specification can be defined by:

$$\varphi_{\text{rg}} := \forall j. (\text{req}(j) \Rightarrow (\exists k. \varphi_\leq(j, k) \wedge \text{grt}(k)))$$

With $I = \{\text{req}, \text{idle}\}$ and $O = \{\text{grt}, \text{idle}\}$.

Büchi showed that the language of a formula is recognised by an ω -automaton (see Section 3.4.3 and Theorem 3.6). Along with the decidability of the emptiness problem for these automata (Theorem 3.5), this yields:

Theorem 3.2 *The satisfiability problem of $\text{MSO}(\mathbb{N}, +1)$ is decidable.*

This translation allows to show that the Church synthesis problem is decidable [8] (Theorem 3.17). See [5] and Section 3.5.3 for a game-theoretic proof of this result.

As $\text{FO}(\mathbb{N}, <)$, $\text{MSO}(\mathbb{N}, +1)$ however has a very high complexity, and its satisfiability problem over finite and infinite words again has a non-elementary lower bound, inherited from that of FO (see also [11]).

Linear Temporal Logic

The high complexity of MSO lead to the introduction of *Linear Temporal Logic* (LTL for short) by Pnueli [12], which is expressively equivalent to first-order logic ($\text{FO}[<]$) [60–62].

Given a set AP of *atomic propositions*, *LTL formulas* are built from *temporal operators* X (*‘next’*) and U (*‘Until’*), as well as disjunction, conjunction (omitted here as it can be derived) and negation:

$$\phi ::= p \mid X\phi \mid \phi U \psi \mid \phi \vee \psi \mid \neg\phi$$

where $p \in \text{AP}$. Intuitively, p means that p holds at the given time. $X\phi$ means that ϕ holds one unit later. $\phi U \psi$ means that ϕ holds until ψ is true. This requires in particular that ψ occurs at some point. On top of derived logical operators like $\top$ and $\perp$, one can then derive temporal operators like *Finally* $F\phi := \top U \phi$, meaning that ϕ will hold at some point in the future and *Globally* $G\phi := \neg(F(\neg\phi))$ meaning that ϕ holds along the entire ω -word.

Example 3.11 Let $w = (ab)^\omega$. w satisfies $\phi_1 = a$, as $w[0] = a$, but not $\phi_2 = b$ (as $w[0] \neq b$), nor $\phi_3 = X(a)$ (as $w[1] \neq a$). It satisfies $\phi_4 := a U b$, but not $\phi_5 := b U a$. Besides, it satisfies $\phi_6 = G(Fa)$ since for all $i \in \mathbb{N}$, one can find a $j \geq i$ in the future such that $w[j] = a$. Finally, w does not satisfy $\phi_7 = F(Ga)$ since for all $i \in \mathbb{N}$, there exists a position $j \geq i$ such that $w[j] \neq a$.

Example 3.12 Consider once more the ω -languages of Example 4.7.2:

- L_{af} can be defined by $\phi_{af} := F(G(\neg a))$.
- $L_{a\infty}$ can be defined by $\phi_{a\infty} := G(Fa) \equiv \neg\phi_{af}$

[8]: J.R. Büchi and L.H. Landweber (1969), ‘Solving sequential conditions finite-state strategies’

[5]: Thomas (2009), ‘Facets of Synthesis: Revisiting Church’s Problem’

[11]: Meyer (1975), ‘Weak monadic second order theory of successor is not elementary-recursive’

[12]: Pnueli (1977), ‘The Temporal Logic of Programs’

[60]: Kamp (1968), ‘Tense Logic and the Theory of Linear Order’

[61]: Gabbay et al. (1980), ‘On the Temporal Basis of Fairness’

[62]: Diekert and Gastin (2008), ‘First-order definable languages’

Equivalence between $\text{FO}[<]$ and LTL is due to [60, 61], and a self-contained proof can be found in the survey [62, Theorem 1.1].

Again, we do not define satisfaction formally, as we do not use it in our study. See [63]: Piterman and Pnueli (2018), ‘Temporal Logic and Fair Discrete Systems’ for a more thorough introduction to LTL.

One can also define ‘Weak until’, which is the same as Until, except that it does not require ψ to eventually occur.

Recall that LTL is equivalent to FO, which is a fragment of MSO.

- L_{par} cannot be defined by any LTL formula, as it is not first-order definable. This can be shown by establishing that L_{par} is not aperiodic [62, Theorem 1.1] (cf Example B.1 for details).
- L_{aB} cannot be defined by any LTL formula, as it is already not MSO-definable.
- L_{halt} cannot either, for the same reason.

Example 3.13 Alternation between alphabets I and O can also be defined in LTL:

$$\phi_{IO} := I \wedge G(I \Leftrightarrow XO)$$

Example 3.14 The request-grant specification is defined by $\phi_{IO} \wedge \phi_{\text{rg}}$, where:

$$\phi_{\text{rg}} := G(\text{req} \Rightarrow F(\text{grt}))$$

As LTL is a fragment of MSO, we know that its reactive synthesis problem is decidable. Pnueli and Rosner further showed:

Theorem 3.3 ([13]) *The reactive synthesis problem for LTL formulas is 2-EXPTIME-complete.*

3.4.2. Discrete Systems

As we have seen, specifications can be expressed as logical formulas. Satisfiability and synthesis are often obtained through translations to automata models. As such discrete systems are at the core of our study, we now introduce a formalism to describe the behaviour of programs which process finite or infinite sequences of input, modelled as words, and later on formally introduce ω -automata.

Discrete systems have a (possibly infinite) set of configurations, and, on reading an input, they transition to some configuration depending on the current configuration and on the input; they are also allowed to perform an action such as outputting a letter in the process.

A central notion is that of a labelled transition system.

Definition 3.2 (Labelled Transition System) A *labelled transition system* is a tuple $\mathcal{T} = (C, I, \Lambda, \Delta)$, where:

- C is a (possibly infinite) set of elements called *configurations*
- $I \subseteq C$ is a set of *initial configurations*
- Λ is a (possibly infinite) set of *labels*
- $\Delta \subseteq C \times \Lambda \times C$ is the *transition relation*; we write $p \xrightarrow[\mathcal{T}]{a} q$ for $t = (p, a, q) \in \Delta$.

$\mathcal{T}$ is *deterministic* if $I = \{c^i\}$ is a singleton and for all $p \in C$ and all $a \in \Lambda$, there exists at most one $q \in C$ such that $p \xrightarrow[\mathcal{T}]{a} q$. It is *complete* if for all $p \in C$ and all $a \in \Lambda$, there exists at least one $q \in C$ such that $p \xrightarrow[\mathcal{T}]{a} q$.

Definition 3.3 (Path) An *infinite path* in $\mathcal{T}$ over an infinite word $x \in \Lambda^\omega$ is an infinite sequence $\rho = q_0 t_0 q_1 t_1 \dots \in (C\Delta)^\omega$ such that for all $i \in \mathbb{N}$, $q_i \xrightarrow[\mathcal{T}]{w[i]}_{t_i} q_{i+1}$. We also write $\rho = q_0 \xrightarrow[\mathcal{T}]{w[0]}_{t_0} q_1 \xrightarrow[\mathcal{T}]{w[1]}_{t_1} \dots$. We then let $\text{configs}(\rho) = q_0 q_1 \dots$ and $\text{trans}(\rho) = t_0 t_1 \dots$, and write $p \xrightarrow[\mathcal{T}]{x}$ when there

[62]: Diekert and Gastin (2008), ‘First-order definable languages’

Where I is an atomic proposition recognising letters of I , similarly for O . Here, we assume that I and O partition the alphabet, i.e. $\Sigma = I \sqcup O$.

[13]: Pnueli and Rosner (1989), ‘On the Synthesis of a Reactive Module’

Infinite sequences of input are also known as *streams*.

We use the term *configuration* to highlight the fact that there can be infinitely many. The term *state* is reserved for systems with finitely many configurations.

Labelled transition systems are also known as *semi-automata*. We favour the first terminology as we also use the concept in the context of games.

Labels are sometimes called *inputs*.

We omit $\mathcal{T}$ and/or t when they are clear from the context.

Some say *executable* for complete.

is an infinite path in $\mathcal{T}$ over x which starts in state p . It is *initial* if $q_0 \in I$. An initial path is also called a *run*.

The notion of *finite path* in $\mathcal{T}$ over a finite word $w \in \Lambda^*$ is defined analogously. $\text{configs}(\rho)$ and $\text{trans}(\rho)$ are extended in a straightforward way, and, as no ambiguity arises due to typing considerations, we overload the notation $\rightarrow$ and write, for all $p, q \in C$ and all $u \in \Lambda^*$, $p \xrightarrow[u]{w} q$ when there exists a finite path in $\mathcal{T}$ over w .

A configuration q is *reachable* if there exists an initial finite path which ends in it, i.e. $q^l \xrightarrow[w]{w} q$ for some initial configuration $q^l \in I$ and some finite word $w \in \Lambda^*$.

Definition 3.4 (Product) Given two labelled transition systems $\mathcal{T} = (C, I, \Lambda, \Delta)$ and $\mathcal{T}' = (C', I', \Lambda', \Delta')$, we can define their *synchronised product* $\mathcal{T} \otimes \mathcal{T}' = (C \times C', I \times I', \Lambda \times \Lambda', \Delta'')$, where $(p, p') \xrightarrow[\mathcal{T} \otimes \mathcal{T}']{(a, a')} (q, q')$ whenever $p \xrightarrow[a]{a} q$ and $p' \xrightarrow[a']{a'} q'$.

Note that if $\mathcal{T}$ is not complete, a finite path is not necessarily the prefix of an infinite path.

It is then straightforward that for all $(w, w') \in \Lambda \times \Lambda'$ such that $|w| = |w'|$ there is a finite path $(p, p') \xrightarrow[\mathcal{T} \otimes \mathcal{T}']{(w, w')} (q, q')$ iff $p \xrightarrow[w]{w} q$ and $p' \xrightarrow[w']{w'} q'$ and similarly for infinite paths.

3.4.3. Automata

To decide satisfiability and synthesis from logical formulas, a common technique is to first convert them to an operational model, i.e. a machine which recognises the models of a formula (cf Theorem 3.6). In the case of MSO and LTL, this model consists in ω -automata, i.e. finite-state automata recognising infinite words. We introduce them here and describe how relations can be encoded as ω -languages, to be able to model specifications directly as automata. Indeed, one of the main contributions of this thesis is to study synthesis problems for specifications given as register automata, which are an extension of finite-state automata to words over infinite alphabets. We also recall the link between ω -automata and MSO($\mathbb{N}, +1$).

The interactions of reactive systems are modelled as infinite words. We correspondingly introduce ω -automata [54, 64, 65], which are a classical extension of finite-state automata to ω -words.

Definition 3.5 (ω -Automaton) An ω -automaton is a tuple $A = (Q, I, \Sigma, \Delta, \Omega)$:

- Q is a *finite* set of *states*
- $I \subseteq Q$ is the set of *initial states*
- Σ is the *alphabet* of the automaton
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*
- (Q, I, Σ, Δ) forms a labelled transition system, called the *underlying transition system*; we write $p \xrightarrow[A]{a} q$ when $t = (p, a, q) \in \Delta$
- $\Omega \subseteq Q^\omega$ is the *acceptance condition*

[54]: Büchi (1962), 'On a decision method in restricted second order arithmetic'

[64]: Muller (1963), 'Infinite sequences and finite machines'

[65]: McNaughton (1966), 'Testing and generating infinite sequences by a finite automaton'

The notion of initial state can be encoded in the acceptance condition, but we keep it explicit to keep closer to the usual setting. It also makes the definition of determinism more natural.

Recall that an alphabet is a *finite* set.

We omit A and t when they are clear from the context.

Acceptance conditions Over infinite words, acceptance is defined through some set that is visited infinitely often. We thus denote, for any set X and for an infinite sequence $w \in X^\omega$, the set $\text{Inf}(w) = \{a \in X \mid w[i] = a \text{ for infinitely many } i \in \mathbb{N}\}$.

Definition 3.6 (Acceptance Condition) Then, an *acceptance condition* is a subset $\Omega \subseteq Q^\omega$ which is of one of the following forms:

Other acceptance conditions exist in the literature, e.g. Streett and Rabin. We do not define them here as they will not be used in this work.

Büchi A has a distinguished set $F \subseteq Q$ of *accepting states*, and a run is accepting if some accepting state is visited infinitely often, i.e. $\text{Buchi}(F) = \{\chi \in Q^\omega \mid \text{Inf}(\chi) \cap F \neq \emptyset\}$.

co-Büchi It is the dual condition: A has a distinguished set $F \subseteq Q$ of *rejecting states*, and a run is accepting if no rejecting state is visited infinitely often, i.e. $\text{coBuchi}(F) = \{\chi \in Q^\omega \mid \text{Inf}(\chi) \cap F = \emptyset\}$.

Muller A has a family $\mathcal{F} \subseteq 2^Q$ of *accepting sets of states*. A run is accepting if the set of states that are visited infinitely often belongs to $\mathcal{F}$: $\text{Muller}(\mathcal{F}) = \{\chi \in Q^\omega \mid \text{Inf}(\chi) \in \mathcal{F}\}$.

Parity A is equipped with a *parity function* $c : Q \rightarrow \{1, \dots, d\}$ for some $d \in \mathbb{N}$, which associates a *colour* to each state. A run is accepting if the maximal colour seen infinitely often is even, i.e. $\text{parity}(c) = \{\chi \in Q^\omega \mid \max \text{Inf}(c(\chi)) \text{ is even}\}$, where $c(\chi)$ is the pointwise application of c to χ .

Note that $\text{coBuchi}(F) = \text{Buchi}(F)^c$.

Recall that the pointwise application of c to χ is the ω -word $c(\chi) \in \{1, \dots, d\}^\omega$ such that for all $i \in \mathbb{N}$, $c(\chi)[i] = c(\chi[i])$.

Remark 3.6 (Relations between acceptance conditions) As noted above, the Büchi and co-Büchi conditions are dual to each other. The Muller condition is closed under complement, by taking $\mathcal{F}^c = 2^Q \setminus \mathcal{F}$, and can naturally encode Büchi (and thus co-Büchi): $\mathcal{F} = \{Q \subseteq F \mid Q \neq \emptyset\}$. It can also encode parity: $\mathcal{F} = \{Q \subseteq F \mid \max c(Q) \text{ is even}\}$. The parity condition is also closed under complement: take $c' : q \mapsto c(q) + 1$. It can easily encode Büchi (respectively co-Büchi) conditions, by setting the colour of accepting states to 2, and 1 for the others; respectively by letting $c(F) = 1$ and $c(F^c) = 0$.

We say that A is a *Büchi* (respectively *co-Büchi*, *Muller*, *Parity*) automaton when its acceptance condition is.

Executions of the model Now, a *run* over input $x \in \Sigma^\omega$ is an infinite path in (Q, I, Σ, Δ) , i.e. an infinite sequence $\rho = q_0 t_0 q_1 t_1 \dots \in (Q\Delta)^\omega$ such that for all $i \in \mathbb{N}$, $q_i \xrightarrow[A]{t_i} q_{i+1}$. It is *initial* if $q_0 \in I$. It is *accepting* if moreover $\text{states}(\rho) \in \Omega$, where $\text{states}(\rho) = q_0 q_1 \dots \in Q^\omega$. We also define $\text{trans}(\rho) = t_0 t_1 \dots \in \Delta^\omega$. A *partial run* $r = q_0 t_0 \dots t_{n-1} q_n \in (Q\Delta)^* Q$ is a finite path $q_0 \xrightarrow[A]{w_0} q_1 \dots q_{n-1} \xrightarrow[A]{w_{n-1}} q_n$ in (Q, I, Σ, Δ) ending in some state $q_n \in Q$, and has as input the finite word $w_0 \dots w_{n-1}$.

Note that the notions of initial and transitions are inherited from labelled transition systems.

Non-determinism and universality Automata can be equipped with two dual semantics. If A is a *non-deterministic automaton*, it *recognises* the following ω -language:

$$L_N(A) = \{w \in \Sigma^\omega \mid \text{there exists an accepting run } \rho \text{ on } w\}$$

Dually, if A is a *universal automaton*, it *recognises* the ω -language:

$$L_U(A) = \{w \in \Sigma^\omega \mid \text{all runs } \rho \text{ on } w \text{ are accepting}\}$$

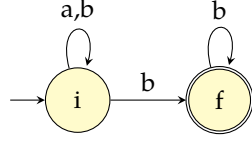
Given a non-deterministic automaton A , we can define a universal automaton A^c which is a copy of A with acceptance condition $\Omega^c = Q^\omega \setminus \Omega$; we then have $L_U(A^c) = \Sigma^\omega \setminus L_N(A)$. If the semantics is clear from context, we simply write $L(A)$.

In other words, the automaton checks that no run is *rejecting*. In particular, in the universal semantics, if $w \in \Sigma^\omega$ does not admit any run, it is accepted.

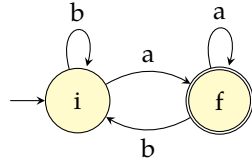
Example 3.15 Consider once more the ω -languages of Example 4.7.2.

A *final state* is either an accepting state for the Büchi acceptance condition, or a rejecting state for the co-Büchi one.

Here are two ω -automata which recognise L_{af} and $L_{a\infty}$ in the non-deterministic Büchi semantics, and conversely as universal co-Büchi. States are depicted as circles, initial states have an incoming arrow, final states are doubly circled, and transitions are labelled arrows.



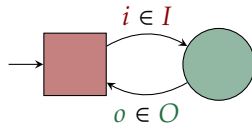
A_1 (Figure 3.4) loops in i until it non-deterministically guesses that the input does not contain a anymore and transitions to f , in which it checks that its guess was correct. In the universal co-Büchi semantics, the automaton universally checks that it sees another a . If it does not, it loops infinitely often in f , which yields a rejecting run.



In A_2 (Figure 3.5), the state f is visited whenever an a is read. Thus, a run visits f infinitely often if and only if the input contains infinitely many a .

Languages L_{aB} and L_{halt} are not ω -regular.

Example 3.16 The following deterministic ω -automaton recognises $L_{IO} = (IO)^\omega$. It has a *trivial acceptance condition*, i.e. all runs are accepting.



Example 3.17 Recall the request-grant specification (Example 3.5). It is recognised by the product of A_{IO} (Figure 3.6) with the universal co-Büchi automaton of Figure 3.7.

The automaton *waits* in w . When it sees a *req*, it universally branches to p , where a request is *pending*. When reading a *grt*, it transitions to state s , in which the run dies out in the next step, which favours acceptance in the universal semantics.

Then, a run of this ω -automaton is rejecting whenever it loops infinitely often in p , which is the case whenever there is some *req* which is never followed by a *grt*.

Remark 3.7 (Alternation) Both semantics can be generalised through *alternation* [27], where states can either be *non-deterministic* or *universal*. Alternating ω -automata are expressively equivalent to non-deterministic and universal automata [66], but they can be exponentially more succinct [27]. We do not define them formally as we mainly study non-deterministic, universal and deterministic automata.

Graphical conventions are summarised in Chapter A.

Figure 3.4.: An ω -automaton A_1 . With the non-deterministic Büchi acceptance condition, it recognises the ω -language $L_{af} = \{w \in \{a, b\}^\omega \mid w \text{ contains finitely many } a\}$. With the dual condition, namely the universal co-Büchi one, it recognises the complement $L_{a\infty} = L_{af}^c = \{w \in \{a, b\}^\omega \mid w \text{ contains infinitely many } a\}$.

Figure 3.5.: A deterministic ω -automaton A_2 . With the Büchi acceptance condition, it recognises $L_{a\infty}$. Again, with the dual, co-Büchi condition, it recognises the complement, L_{af} .

Figure 3.6.: An ω -automaton A_{IO} recognising $L_{IO} = (IO)^\omega$.

$p \xrightarrow{i \in I} q$ is a short for $\left\{ p \xrightarrow{i} q \right\}_{i \in I}$, similarly for O .

[27]: Chandra, Kozen, and Stockmeyer (1981), ‘Alternation’

A run is then a tree such that each non-deterministic state has one child, and each universal state has as children all states to which it has a transition. It is *accepting* whenever all its paths are accepting. Alternation is more easily defined as a game, see Section 3.5.2.

[66]: Miyano and Hayashi (1984), ‘Alternating Finite Automata on omega-Words’

[27]: Chandra, Kozen, and Stockmeyer (1981), ‘Alternation’ (Theorem 5.3). The result is for the finite alphabet case, but can easily be adapted.

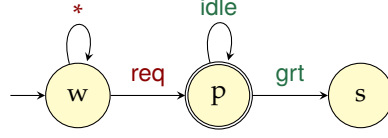


Figure 3.7.: A universal co-Büchi automaton checking that every request is eventually granted.

The red star (*) is a short for any input letter, i.e. for $\{\text{req}, \text{idle}\}$.

A is called *deterministic* if (Q, I, Σ, Δ) is deterministic, i.e. $I = \{q^i\}$ is a singleton and for all $p \in Q$ and all $a \in \Sigma$, there exists at most one $q \in Q$ such that $p \xrightarrow{a} q$. We then write Δ as a function $\delta : Q \times \Sigma \rightarrow Q$.

Example 3.18 In Example 3.15, A_2 is deterministic, but A_1 is not.

Languages recognised by ω -automata are called *ω -regular languages*, as they generalise the notion of regular language to ω -words.

Relations between classes of ω -automata

Fact 3.1 Any non-deterministic ω -automaton is equivalent to a deterministic parity automaton [65, 67]. The construction is more involved than the subset construction that is used in the finite word case, as one needs to maintain additional information. It is exponential in general, more precisely $2^{\Theta(n \log n)}$ (where n is the size of the automaton) when one starts from a non-deterministic Büchi or parity automaton.

Conversely, any deterministic parity automaton can be simulated by a non-deterministic Büchi automaton: the ω -automaton initially guesses the maximal parity. It checks that it is even and that no greater parity occurs infinitely often along the run. The latter can be done by guessing that at some point, greater parities stop occurring; the automaton can check that this is the case by rejecting as soon as it sees one.

Since the parity condition is closed under complement, the above again holds for the universal semantics, by replacing ‘non-deterministic Büchi’ with its dual ‘universal co-Büchi’.

However, not all classes admit determinisation. In particular, the class of non-deterministic Büchi automata does not, i.e. such an automaton does not always admit an equivalent deterministic Büchi automaton. For instance, one can establish that the ω -language L_{af} (see Example 3.15) of words that contain only finitely many a is not recognisable by any deterministic Büchi automaton, using a pumping argument [68, Lemma 3.1].

Finally, note that the non-deterministic co-Büchi condition is strictly weaker. For instance, it cannot recognise the language $L_{a\infty}$ of ω -words that contain infinitely many a . In this work, we only use it in the universal semantics. It is then dual to non-deterministic Büchi.

Overall, in the following, we are mostly concerned with non-deterministic Büchi, universal co-Büchi, and non-deterministic and deterministic parity automata.

We refer to [70] for an exhaustive comparison of ω -automata classes, along with the complexity of the translation between them. $\square$

We say that two automata A and A' are *equivalent* whenever $L(A) = L(A')$.

[65]: McNaughton (1966), ‘Testing and generating infinite sequences by a finite automaton’

[67]: Safra (1988), ‘On the Complexity of ω -Automata’

An automaton class admits *determinisation* when for all automata in this class, there exists a deterministic automaton in the same class which recognises the same language.

[68]: Landweber (1969), ‘Decision Problems for omega-Automata’. See also [69]: Kupferman and Sickert (2021), ‘Certifying Inexpressibility’ for a general approach on how to determine whether an ω -language can be expressed by a given class of automata.

[70]: Boker (2020), *Word-Automata Translations*

The case of finite words

Definition 3.7 (Finite-State Automaton) *Finite-state automata* are defined analogously, by replacing the acceptance condition with a finite set of *accepting states*. A *run* is then a finite sequence of transitions, and it is *accepting* whenever it is initial and ends in an accepting state. The notions of *non-deterministic*, *universal* and *deterministic* automata are easily adapted. All semantics are again equivalent, and languages recognised by finite automata are called *regular languages*.

We do not define them formally as they are only used as a tool in our study. We refer to [71] for a comprehensive introduction.

[71]: Sakarovitch (2009), *Elements of Automata Theory*

Emptiness Problem

An elementary problem in automata theory is to decide whether a given automaton accepts at least a word. It is formalised as the *emptiness problem*.

Problem 3.3: EMPTINESS PROBLEM FOR ω -AUTOMATA

Input: An ω -automaton A
Output: Yes if $L(A) = \emptyset$
 No otherwise

It is well-known that the above problem can be solved in polynomial time for finite-state automata. More precisely, it belongs to $NLOGSPACE$, i.e. it can be solved by a non-deterministic algorithm that runs in logarithmic space. This is the case as well for non-deterministic Büchi automata; the algorithm is essentially the same.

It is easy to see that $NLOGSPACE \subseteq PTIME$, since one can iteratively explore all the runs of a Turing machine in a time that is exponential in the space used by the machine.

Theorem 3.4 *The emptiness problem for non-deterministic Büchi automata is in $NLOGSPACE$.*

This will prove useful for our study, especially in Part II, as we reduce most decision problems to checking emptiness of some well-chosen ω -automaton. The reader can find a more detailed presentation of these complexity considerations in Section B.2.

Since a non-deterministic parity automaton (respectively Muller automaton) can be translated into an equivalent non-deterministic Büchi automaton in quadratic (resp., cubic) time [70, Sections 7.8 and 7.16], we get:

[70]: Boker (2020), *Word-Automata Translations*

Theorem 3.5 *The emptiness problem for non-deterministic ω -automata is in $PTIME$.*

Equivalence with MSO

It has been shown by Büchi [72], and independently by Elgot [73] and Trakhtenbrot, that MSO formulas over finite words can be translated into finite-state automata and conversely. It has been generalised to infinite words by Büchi [54] to show decidability of the satisfiability problem for MSO (cf Theorem 3.2) and McNaughton [65]:

Theorem 3.6 ([54, 65]) *Let $L \subseteq \Sigma^\omega$ be a language of infinite words. The following are equivalent:*

[72]: Büchi (1960), 'Weak Second-Order Arithmetic and Finite Automata'

[73]: Elgot (1961), 'Decision problems of finite automata design and related arithmetics'

[74]: Trakhtenbrot (1961), 'Finite automata and logic of monadic predicates', in Russian.

[54]: Büchi (1962), 'On a decision method in restricted second order arithmetic'

[65]: McNaughton (1966), 'Testing and generating infinite sequences by a finite automaton'

- (i) L is recognised by an ω -automaton, i.e. $L = L(A)$ for some ω -automaton A .
- (ii) L is recognised by an MSO formula, i.e. $L = L(\varphi)$ for some MSO formula φ .

Moreover, both translations are effective.

Remark 3.8 This equivalence also holds for *weak MSO*, i.e. monadic second-order logic where quantification is restricted to finite subsets of $\mathbb{N}$ [54, 57].

Note that the translation from MSO formulas to ω -automata is however non-elementary in general, and this is intrinsic since the satisfiability problem of MSO has a non-elementary lower bound [11].

The result holds in particular for specifications expressed in MSO. Along with game-theoretic results, this yields decidability of the Church synthesis problem for MSO specifications (cf Theorem 3.17 in Section 3.5.3).

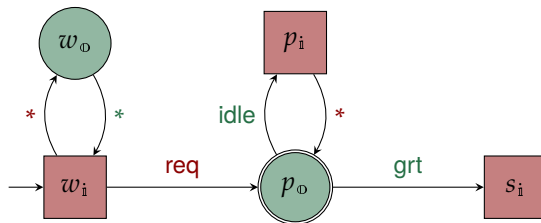
Automatic Specifications

As we have seen earlier, relations over ω -words can be encoded as ω -languages (cf Section 3.3.1). Thus, an ω -automaton A recognising some ω -regular language $L \subseteq (IO)^\omega$ can also be seen as recognising the relation $\rangle L \langle \subseteq I^\omega \times O^\omega$. Note that such automaton can be put in a form where states are partitioned into input (Q^i) and output (Q^o) states, and the transition relation alternates between the two sets. More precisely, we can assume $A = (Q, I, \Sigma, \Delta, \Omega)$, where:

- $Q = Q^i \sqcup Q^o$
- $I \subseteq Q^i$
- $\Sigma = \Sigma^i \cup \Sigma^o$
- $\Delta = \Delta^i \sqcup \Delta^o$, with $\Delta^\sigma \subseteq Q^\sigma \times \Sigma^\sigma \times Q^{\bar{\sigma}}$
- $\Omega \subseteq (Q^i Q^o)^\omega$

We call automata in this form *specification automata*, and denote $\llbracket A \rrbracket = \rangle L(A) \langle$ the relation they encode. Relations recognised by specification automata are called *automatic relations* [75, 76]. When considering synthesis problems, we also use the term *automatic specifications*.

Example 3.19 Consider again Example 3.17. The universal co-Büchi automaton of Figure 3.8 recognises the specification S_{rg} of Example 3.5:



The automaton waits in w , looping between w_i and w_o . When it sees a *req*, it universally branches to p_o , where the request is *pending*. When reading a *grt*, it transitions to state s_i , in which the run dies out in the next step, which favours acceptance in the universal semantics.

In other words, in $\forall X.\varphi$ and $\exists X.\varphi$, X is finite.

[54]: Büchi (1962), ‘On a decision method in restricted second order arithmetic’

[57]: Weyer (2001), ‘Decidability of SIS and S2S’

[11]: Meyer (1975), ‘Weak monadic second order theory of successor is not elementary-recursive’

Recall that an ω -word $w = i_0 o_0 i_1 o_1 \dots$ encodes a pair $\rangle w \langle = (i_0 i_1 \dots, o_0 o_1 \dots)$.

When they are deterministic, such automata naturally induce a game which corresponds to the synthesis game for the specification they encode. Indeed, their underlying transition system is an arena, and their acceptance condition defines a winning condition (cf Section 3.5.2).

For $\sigma \in \{i, o\}$, we set $\bar{i} = o$ and $\bar{o} = i$.

[75]: Khousainov and Nerode (1994), ‘Automatic Presentations of Structures’

[76]: Blumensath and Grädel (2000), ‘Automatic Structures’

See also [48]: Carayol and Löding (2015), ‘Uniformization in Automata Theory’ for a study of uniformisation and synthesis problems on those structures.

Figure 3.8.: A universal co-Büchi automaton recognising S_{rg} .

Input states are red squares with index i . Output states are green circles with index o . The red (respectively green) star $(*,*)$ denotes any input (respectively output) letter.

3.4.4. Programs Computable by Finite-State Machines

Regarding the target implementation, Church requires that it is not only synchronous, but further computed by a *circuit*, i.e. a finite-state machine. In this section, we formalise what is meant with this notion through the model of synchronous transducer. We start by giving a definition of finite-state computability for terminating programs, both to give a flavour of the definitions for non-terminating ones and since it allows us to define what is a finite-memory strategy in the context of games (cf Definition 3.14).

Definition 3.8 (Finite-State Computability) Let $A = (Q, I, \Sigma, \delta, F)$ be a deterministic finite-state automaton. We equip A with an *output function* $o : F \rightarrow \Gamma$. (A, o) then *computes* the function $f : L(A) \rightarrow \Gamma$ defined, for all $w \in L(A)$, by $o(q_f)$, where q_f is the state in which the run of A over w ends.

Then, $f : \Sigma^* \rightarrow \Gamma$ is *finite-state computable* if it can be computed by some finite-state machine (A, o) .

Remark 3.9 Many models have been elaborated to define finite-state computability. The above definition can be generalised to functions over finite words $f : \Sigma^* \rightarrow \Gamma^*$ through the notion of a *Moore machine* [77], which consists in a deterministic finite-state automaton which outputs a letter everytime it visits a state.

[77]: Moore (1956), ‘Gedanken-Experiments on Sequential Machines’

A closely related notion is that of a *Mealy machine* [78], which again consists in a deterministic finite-state automaton, but which outputs a letter on each *transition*. Both models are expressively equivalent, although Mealy machines usually have fewer states. For this reason, we define transducers over infinite words as a generalisation of the latter model.

[78]: Mealy (1955), ‘A method for synthesizing sequential circuits’

Example 3.20 Consider again the functions $f_{\text{grt}} : i \in I^* \mapsto \text{grt}$ and $f_{\text{rg}} : \begin{cases} i \cdot \text{req} \mapsto \text{grt} \\ i \cdot \text{idle} \mapsto \text{idle} \end{cases}$ of Example 3.6. Both are finite-state computable (Figure 3.9).

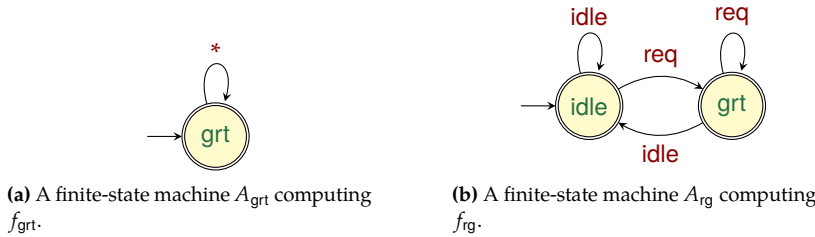


Figure 3.9. Two finite-state machines respectively computing f_{grt} and f_{rg} of Example 3.6.

The output of a state is written in place of its name, which is irrelevant here.

Now, a function $g : \Sigma^\omega \rightarrow \Gamma^\omega$ is *synchronously ω -computable* by a finite-state machine if it is the ω -behaviour of some finite-state computable function $f : \Sigma^* \rightarrow \Gamma$ or, equivalently, if it is computable by some synchronous sequential transducer.

A synchronous transducer has a finite set of states. It starts its computation in its unique initial state, and then processes its stream of input letter by letter as follows: on reading an input letter, it deterministically produces an output letter and transitions to the next state. Given an infinite input word, it thus produces an infinite output word, in a synchronous way.

“Sequential” is to be understood as “deterministic”. The former is however more commonly used in the field of transducers, to avoid the confusion with another notion of determinism, namely that the transducer is deterministic when considered as an automaton which alternately reads inputs and outputs. Some also use the term *input-deterministic*.

We now formally introduce the model, as it will be used thoroughly in our study.

Definition 3.9 (Synchronous Sequential Transducer) A *synchronous sequential transducer* is a tuple $T = (Q, \Sigma, \Gamma, q^i, \delta)$, where:

- Q is a finite set of *states*
- Σ (respectively Γ) is the *input* (resp., *output*) alphabet
- $q^i \in Q$ is the *initial state*
- $\delta : Q \times \Sigma \rightarrow \Gamma \times Q$ is the *transition function*. We write $p \xrightarrow[T]{a|b} q$ whenever $\delta(p, a) = (b, q)$.

As for automata, T and t can be omitted when clear from the context.

In this chapter, we simply write *transducer* to denote synchronous sequential transducers, as no ambiguity arises. A more general notion, that of asynchronous transducers, is studied in Part II.

We now define the semantics of the model: a transducer T naturally induces a labelled transition system $\mathcal{T}_T = (Q, \{\delta\}, \Sigma \times \Gamma, \rightarrow)$, where $p \xrightarrow[\mathcal{T}_T]{(a,b)} q$ whenever $\delta(p, a) = (b, q)$. Then, given an input word $u \in \Sigma^\omega$, the *run* over u is the unique initial path $\rho = q_0 t_0 q_1 t_1 \dots$ in $\mathcal{T}_T$ over a pair (u, v) for some $v \in \Gamma^\omega$. We say that ρ *produces* output v . Note that this run indeed exists and is unique since δ is a (total) function. As usual, we define $\text{states}(\rho) = q_0 q_1 \dots$ and $\text{trans}(\rho) = t_0 t_1 \dots$. Then, T *computes* the function $f_T : \Sigma^\omega \rightarrow \Gamma^\omega$ which, to every $u \in \Sigma^\omega$, associates the output $v \in \Gamma^\omega$ of its unique run ρ .

Recall that ρ is initial means that $q_0 = q^i$.

Note that here, transducers are assumed to accept any input (cf Remark 3.5), the focus being set on the output they produce.

Example 3.21 Consider again Example 3.20. The ω -behaviour of A_{grt} and A_{rg} are respectively modelled by the transducers of Figure 3.10.

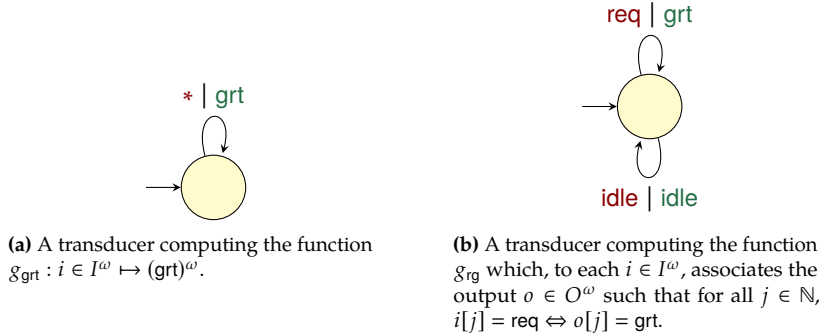


Figure 3.10.: Two synchronous sequential transducers, respectively computing the ω -behaviour of f_{grt} and f_{rg} of Example 3.20.

Accepting states are not depicted, as all states are accepting.

3.4.5. Statement of the Church Synthesis Problem

The *Church synthesis problem* for automatic specifications can now be precisely formulated as follows:

Recall that a specification is automatic if it is recognised by a specification automaton.

Problem 3.4: CHURCH SYNTHESIS PROBLEM

- Input:** An automatic specification $S \subseteq I^\omega \times O^\omega$, given for instance as an MSO formula φ_S
- Output:** A transducer T computing a function $f_T : I^\omega \rightarrow O^\omega$ which realises S_φ (i.e. $f_T \subseteq S_\varphi$) if it exists
No otherwise

In our study, we consider generalisations of the Church synthesis problem over infinite alphabets for different formalisms of specifications (Chapter 5). The notion of synchronous sequential transducer is correspondingly extended (Section 5.3).

3.5. Games for Reactive Synthesis

Reactive synthesis problems are conveniently presented as games, as they capture the core notion of interaction. This section introduces the necessary definitions and results, and presents a game-theoretic formulation of the Church synthesis problem.

3.5.1. Games

As the name suggests, one can think of a game as an everyday game like checkers or chess (cf Example 3.22), except that the interaction lasts forever. More precisely, in a two-player turn-based zero-sum game, two *players* interact, who despite the biblical connotation we choose to call Adam and Eve. They alternately move some token over a graph, called an arena, starting from a distinguished initial vertex which belongs to Adam. The sequence of actions they choose forms a play. The goal of *Eve* is that such play is winning, i.e. belongs to the winning condition of the game. Antagonistically, the goal of *Adam* is to make Eve lose, i.e. that the play does not belong to the winning condition. Each player plays according to some strategy, which specifies the next action to take, given the history of the play, i.e. the sequence of actions which have been played to far. A strategy is winning if all plays that are consistent are winning.

Notation 3.10 We use $\forall$ to denote Adam and $\exists$ for Eve. Given a player $\sigma \in \{\forall, \exists\}$, we denote $\bar{\sigma}$ its *antagonist*. Thus, $\bar{\forall} = \exists$ and $\bar{\exists} = \forall$.

Let us now formalise the above notions:

Definition 3.11 (Arena) A *two-player arena* is a deterministic labelled transition system $A = (V, \{v^i\}, M, \alpha)$ which is partitioned between the two players:

- V is a (possibly infinite) set of *vertices*, partitioned into $V = V^\forall \sqcup V^\exists$, where $V^\forall$ are vertices belonging to Adam and $V^\exists$ belong to Eve
- $v^i \in V^\forall$ is the *initial vertex*, which belongs to Adam.
- M is a set of *moves*, which are also partitioned into $M = M^\forall \sqcup M^\exists$
- Its *edges* are $\alpha = \alpha^\forall \sqcup \alpha^\exists$, where $\alpha^\sigma : V^\sigma \times M^\sigma \rightarrow V^{\bar{\sigma}}$. Note that we require players to alternate.

Definition 3.12 (Plays, histories and winning condition) A *play* is an infinite sequence $v_0 m_0 v_1 m_1 \dots \in (VM)^\omega$ such that $v_0 = v^i$ and for all $j \geq 0$, $v_{j+1} = \alpha(v_j, m_j)$. An *history* $h = v_0 m_0 \dots m_{n-1} v_n \in (VA)^*V$ is a finite prefix of a play which ends in some vertex v_n . If $v_n \in V^\forall$, we say that it *belongs* to Adam, otherwise to Eve. We respectively denote by $\text{Plays}(A)$ and $\text{Hist}(A)$ the set of plays and histories of an arena, and

Eve is also known as the *Protagonist*, the *Existential player*, or the *System* or *Output player* in our synthesis setting[†].

Adam is also known as the *Adversary*, the *Universal player* or the *Environment* or the *Input player*^{*}.

Recall that a labelled transition system is a (possibly infinite) labelled graph with an initial vertex.

Remark 3.10 Arenas can also be defined without labels, i.e. as a non-labelled graph with a distinguished initial vertex, and one can switch to this latter formalism by taking as vertices $V' = V \times M$. Conversely, vertices can be omitted and encoded into actions; such games are known as Gale-Stewart games. We keep both as they allow for a more intuitive construction of games and strategies that correspond to automata, as vertices then correspond to states and actions with transitions.

[†]: You might also encounter her under the name of *Eloise* (and occasionally as the First of Her Name, The Unburnt, Queen of the Andals, the Rhoynar and the First Men, Queen of Meereen, Khaleesi of the Great Grass Sea, Protector of the Realm, Lady Regent of the Seven Kingdoms, Breaker of Chains and Mother of Dragons).

^{*}: Or *Abelard* when he plays with Eloise.

further let $\text{Hist}(A) = \text{Hist}^\forall(A) \sqcup \text{Hist}^\exists(A)$, where $\text{Hist}^\forall(A)$ (respectively $\text{Hist}^\exists(A)$) is the set of histories belonging to Adam (respectively to Eve).

A *winning condition* is a subset $W \subseteq (VA)^\omega$.

We extend the edge function α^σ from vertices to histories as follows: given a history h^σ whose last vertex is v^σ , we let, for all moves $m^\sigma \in M^\sigma$, $\alpha^\sigma(h^\sigma m^\sigma) := \alpha^\sigma(v^\sigma, m^\sigma)$

Definition 3.13 (Game) A *game* is then a tuple $G = (A, W^\exists)$, where:

- A is the arena
- $W^\exists \subseteq \text{Plays}(A)$ is the *winning condition* for Eve.

The games we study are *zero-sum*, which means that Eve wins if and only if Adam loses. Thus, the *winning condition* of Adam is the complement $W^\forall = \text{Plays}(A) \setminus W^\exists$.

Definition 3.14 (Strategy) A *strategy* for player σ is a total function $\lambda^\sigma : \text{Hist}^\sigma(A) \rightarrow M^\sigma$. We say that a play ρ is *consistent* with λ^σ if for all finite prefixes $hm < \rho$ where $h \in \text{Hist}^\sigma(A)$ and $m \in M^\sigma$, we have $m = \lambda^\sigma(h)$. We denote again $\text{Plays}(\lambda^\sigma)$ the set of plays consistent with λ^σ .

A strategy λ^σ is *winning* for player σ if all plays consistent with it belong to W^σ , i.e. $\text{Plays}(\lambda^\sigma) \subseteq W^\sigma$.

Remark 3.11 We assume that a strategy is a total function $\lambda^\sigma : \text{Hist}^\sigma(A) \rightarrow M^\sigma$. Actually, it suffices to define it over histories that are consistent with it. Thus, a strategy λ^σ can be given as a set of histories H_{λ^σ} which:

- Contains the history consisting in the initial vertex: $v^\iota \in H_{\lambda^\sigma}$
- Is *universally closed* for histories belonging to $\bar{\sigma}$: for all $h^\sigma \in H_{\lambda^\sigma} \cap \text{Hist}^\sigma(A)$, for all moves $m^\sigma \in M^\sigma$, by letting $v^\sigma = \alpha^\sigma(h^\sigma m^\sigma)$, we have $h^\sigma m^\sigma v^\sigma \in H_{\lambda^\sigma}$.
- Is *existentially closed* for histories belonging to σ : for all $h^\sigma \in H_{\lambda^\sigma} \cap \text{Hist}^\sigma(A)$, there exists a unique move $m^\sigma \in M^\sigma$, such that, by letting $v^\sigma = \alpha^\sigma(h^\sigma m^\sigma)$, we have $h^\sigma m^\sigma v^\sigma \in H_{\lambda^\sigma}$.

The limit of such set is the set of plays that are consistent with λ^σ .

To build strategies, we often build this set by induction.

A strategy λ^σ is *finite-memory* if it is finite-state computable.

It is *positional* if, moreover, it only depends on the current state, i.e. its memory set is a singleton. In other words, there exists a function $\mu : V^\sigma \rightarrow M^\sigma$ such that for all histories $hv \in \text{Hist}^\sigma(A)$, $\lambda^\sigma(hv) = \mu(v)$.

Definition 3.15 (Determinacy) Let G be a game. We say that G is *determined* if either Adam has a winning strategy in G , or Eve has one.

In some cases, not only is the game determined, but one can also restrict to subclasses of strategies, typically finite-memory or positional ones. We say that G is *finite-memory determined for Eve* (respectively *for Adam*) if G is determined and whenever Eve (resp. Adam) has a winning strategy, she (resp. he) has a finite-memory one. When G is finite-memory determined for both players, we simply say that G is *finite-memory determined*.

Similarly, one can define *positional determinacy* (again for Eve, for Adam and for both) by replacing ‘finite-memory’ with ‘positional’.

Determinacy holds for a large class of games:

No ambiguity arises due to typing considerations.

Again, ambiguity is resolved with typing.

Thus, λ^σ is *finite-memory* if there exists a finite memory set Q , an initial memory $q^\iota \in Q$ a memory update function $\delta : Q \times M^\sigma \rightarrow Q$ and a move selection function $o : V^\sigma \times Q \rightarrow M^\sigma$ such that λ^σ can be computed as follows: initially, the memory is q^ι . When the play is in vertex v^σ and the player’s memory state is q , s/he plays $o(v^\sigma, q)$. When its antagonist plays a^σ , player σ updates its memory to $\delta(q, a^\sigma)$.



Figure 3.11.: Adam and Eve in the Garden of Eden, devising over whether they should eat the forbidden fruit. Eve’s objective (as convinced by the serpent, but this is not modelled here) is that they do, and Adam is that they do not (as instructed by God, again not modelled). Fortunately for Science, less so for eternal bliss, Eve wins the play. It is open whether she had a winning strategy, as it amounts to deciding whether the Fall was inevitable.

Theorem 3.7 ([79]) *Let $G = (A, W)$ be a game. If A and W are Borel sets, then G is determined.*

Definition 3.16 (ω -regular game) A game $G = (A, W)$ is ω -regular when A is finite and W is ω -regular.

Theorem 3.8 ([80, 81]) *ω -regular games are finite-memory determined.*

Corollary 3.9 *Let G be an ω -regular game. If Eve (respectively Adam) has a winning strategy in G , then she (respectively he) has a finite-memory winning strategy in G .*

Example 3.22 Everyday games, somewhat unsurprisingly, fit the framework of games, although not all of them are two-player turn-based zero-sum games:

- Chess and checkers fit this setting, up to the fact that one must arbitrarily choose whether draws are winning for white (Adam, as white starts) or black (Eve). Go can also be modelled. The arena consists in all possible states of the board, and moves correspond to moving pieces on the board. Note that in those games (and most everyday games), termination of every play is enforced by the rules. This is modelled by adding two extra sink vertices, which are respectively winning for Adam and Eve, and which are reached after the corresponding player won. The goal of each player is thus to reach his/her *winning vertex*. Games with such winning condition are called *reachability games*. Note that reachability games are ω -regular games. This implies that chess is finite-memory determined (Theorem 3.8; reachability games are even positionally determined). However, the combinatorics are too high for a winning strategy to be computed in an exact way. Yet, the brute computational power of modern computers, along with well-designed search algorithms, yield approximations that are good enough to win against world champions [82]. Statistical methods for machine-learning furthers this advantage [83].
- Multi-player games, like most card games, obviously do not fit the two-player setting.
- Games where a player hides his/her hand to the others require to add *imperfect information*. Under certain conditions, these games can be reduced to games with perfect information (see e.g. [84]).
- *Concurrent games*, like rock-paper-scissors, are not turn-based.
- Finally, *cooperative games* are not zero-sum, as when one player wins, the other does not necessarily lose.

[80]: Gurevich and Harrington (1982), ‘Trees, Automata, and Games’

[81]: Büchi (1983), ‘State-Strategies for Games in $F_{\sigma\delta} \cap G_{\delta\sigma}$ ’

[82]: Hsu, Campbell, and Jr. (1995), ‘Deep Blue System Overview’

[83]: Silver et al. (2018), ‘A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play’

[84]: Doyen and Raskin (2011), ‘Games with imperfect information: theory and algorithms’

Parity Games

Parity games form a class of ω -regular games. They were introduced in [85], and provide a core tool for the study of problems involving ω -regularity, and in particular synthesis problems. As we use them extensively in this study, we define them formally and describe some of their properties.

Definition 3.17 (Parity Game) A *parity game* is a game $G = (A, W)$, where $A = (V, \{v^i\}, M, \alpha)$ and W is defined through a function $c : V \rightarrow \{1, \dots, d\}$ called the *parity function*, where $d \in \mathbb{N}$ is the *parity index* of G .

[85]: Mostowski (1985), ‘Regular expressions for infinite trees and a standard form of automata’

This is of course reminiscent of the parity acceptance condition for automata.

A play is then winning if the maximal colour seen infinitely often is even, i.e. $W = \{\rho \in \text{Plays}(A) \mid \max \text{Inf}(c(\text{vertices}(\rho))) \text{ is even}\}$.

Note that W can be defined by a parity automaton. More generally, all acceptance conditions defined for ω -automata (Definition 3.6) induce a corresponding winning condition and a class of ω -regular games. Section 3.5.2 explores the relations between games and automata in more detail.

One of the main properties of parity games is positional determinacy: not only are they finite-memory determined (since they are ω -regular games), but the player who has a winning strategy has a positional one.

Theorem 3.10 ([86, 87]) *Let $G = (A, W)$ be a parity game. Then, either Adam has a positional winning strategy, or Eve has one.*

Corollary 3.11 *Let G be a parity game. If Eve (respectively Adam) has a winning strategy in G , then she (respectively he) has a positional winning strategy in G .*

Remark 3.12 Note that this holds even if A is infinite [88].

Moreover, such games can be solved in exponential time:

Theorem 3.12 ([88]) *Let G be a parity game with finite arena A of size $|A| = n$ and with parity index d . It can be decided in time $\mathcal{O}(n^d)$ whether Eve has a winning strategy in G .*

The corresponding winning strategy can be obtained with the same complexity.

Remark 3.13 The problem of solving a parity game is in $\text{NP} \cap \text{coNP}$. It is a long-standing open problem whether parity games can be solved in polynomial time. In 2017, there was a breakthrough result establishing that parity games can be solved in quasi-polynomial time $\mathcal{O}(n^{\log d})$ [89], which was followed by the discovery of three other algorithms of equivalent complexity [90–92]. It was also established that Zielonka’s algorithm [88] can be modified to yield the same upper bound [93]. However, the same paper notes that as of today, Zielonka’s algorithm outperforms the others in practice. As it does not affect our complexity upper bounds, we use this algorithm to solve parity games.

3.5.2. Games and Automata

Acceptance as a Game and Alternation

Automata and games are closely related concepts. First, acceptance of a word w by a non-deterministic automaton A can be defined as a one-player game $G_{A,w}$, where Eve is tasked with finding an accepting run, and has a winning strategy λ_w if and only if w is accepted. Dually, if A is instead universal, w is accepted if and only if Adam is not able to exhibit a rejecting run. It is then natural to generalise those two semantics through *alternation* [27], where both non-deterministic and universal states are allowed, and Eve solves non-determinism while Adam solves universality.

[86]: Emerson and Jutla (1988), ‘The Complexity of Tree Automata and Logics of Programs (Extended Abstract)’

[87]: Mostowski (1991), *Games with Forbidden Positions*

[88]: Zielonka (1998), ‘Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees’

[89]: Calude et al. (2017), ‘Deciding parity games in quasipolynomial time’

[90]: Jurdzinski and Lazic (2017), ‘Succinct progress measures for solving parity games’

[91]: Fearnley et al. (2017), ‘An ordered approach to solving parity games in quasi polynomial time and quasi linear space’

[92]: Lehtinen (2018), ‘A modal μ perspective on solving parity games in quasipolynomial time’

[93]: Parys (2019), ‘Parity Games: Zielonka’s Algorithm in Quasi-Polynomial Time’

A *one-player game* is a two-player game where one player only has a trivial move.

[27]: Chandra, Kozen, and Stockmeyer (1981), ‘Alternation’

While all semantics are expressively equivalent in the finite alphabet case (cf Remark 3.7), this is not the case anymore in the infinite alphabet case, as we will see in Section 4.3.

Games over Specification Automata

On the other hand, let $S = (Q, q^i, \Sigma, \Delta, \Omega)$ be a deterministic specification automaton. Its states are thus partitioned into input (Q^i) and output (Q^o) states. We define the *synthesis game* G_S over S , where Adam ($\forall$) is in charge of the input (i), and Eve ($\exists$) of the output (o), so we let $V = Q^i \sqcup Q^o$ and $M = \Delta^i \sqcup \Delta^o$. Note that $q^i \in Q^i$. Thus, $(Q, \{q^i\}, \Sigma, \Delta)$ is an arena. We then equip such arena A with the winning condition $W = \{\rho \in \text{Plays}(A) \mid \text{states}(\rho) \in \Omega\}$.

Now, it follows from the definitions that:

Proposition 3.13 *Let S be a specification automaton. The following are equivalent:*

1. *There exists a (finite-state computable) synchronous program which implements $\llbracket S \rrbracket$*
2. *Eve has a (finite-memory) winning strategy in G_S*

As a consequence, we get the following result:

Theorem 3.14 *The reactive synthesis problem for specifications given as non-deterministic parity automata is ExpTime -complete.*

Proof. The upper bound was first established in [8] and [13]. It can be obtained by computing a deterministic parity automaton D which is equivalent to the specification automaton S , and solving the resulting parity game G_D . Such algorithm runs in exponential time, as D can be computed in exponential time and has $2^{\Theta(n \cdot \log n)}$ states and $\Theta(n \cdot d)$ colours (where n is the number of states of S and d its parity index) [94, 95]. Finally, the winner of G_D , along with a positional winning strategy, can be computed in $2^{\Theta(n^2 \cdot d \cdot \log n)}$ by Theorem 3.12).

Hardness is folklore, but a proof in the particular case of finite words (easily adapted to the ω -word setting) can be found in [96, Proposition 6]. $\square$

Moreover, since parity games are positionally determined, we know that Eve has a winning strategy if and only if she has a positional one. As a consequence, the reactive synthesis problem is equivalent to the Church synthesis problem, and we get

Theorem 3.15 *The Church synthesis problem for specifications given as non-deterministic parity automata is ExpTime -complete.*

It is often convenient to represent the winning condition as an ω -automaton, that we call an observer.

Definition 3.18 Let $G = (A, W)$ be game. If W is defined by an automaton Ob , we say that Ob is an observer for G .

By building the synchronised product of the arena A and of the underlying transition system of Ob , we can easily show that:

Proposition 3.16 *Let $G = (A, W)$ be a game whose winning condition is given by an observer Ob with states Q and acceptance condition Ω . If Ob is deterministic, G is equivalent to a game $G \otimes \text{Ob} = (A \otimes \text{Ob}, \pi_Q^{-1}(\Omega))$, where π_Q denotes the projection on the Q component.*

Recall that in a specification automaton, transitions alternate between the two sets, i.e. $\Delta^o \subseteq Q^o \times \Sigma \times Q^i$.

Recall that the initial state is an input state, so it belongs to Adam.

To be precise, we must ask that such strategy is moreover a computable object, i.e. that it can be defined by a Turing machine.

[8]: J.R. Büchi and L.H. Landweber (1969), ‘Solving sequential conditions finite-state strategies’

[13]: Pnueli and Rosner (1989), ‘On the Synthesis of a Reactive Module’

The result is originally due to [94]: Safra (2006), ‘Exponential Determinization for omega-Automata with a Strong Fairness Acceptance Condition’. As the construction is quite involved, we refer to [95] for a simpler presentation, as well as lower bounds for the constructions.

[96]: Filiot et al. (2016), ‘On Equivalence and Uniformisation Problems for Finite Transducers’

G is equivalent to G' whenever player σ has a winning strategy in G iff it has one in G' . Here, it is moreover easy to convert such strategy from one game to the other.

Remark 3.14 (Good-for-Games Automata) The above construction actually works for a larger class of automata, namely good-for-games automata [97], also known as *history-deterministic automata* [98]. An automaton is *good-for-games* if Eve can resolve non-determinism knowing only the history. In other words, she has a uniform strategy to win $G_{A,w}$ for all $w \in L(A)$. Such notion proves useful for synthesis applications, as good-for-games automata can be exponentially more succinct than their deterministic counterpart for the co-Büchi acceptance condition; for the Büchi condition, they are only quadratically smaller, and the question is open for parity automata [99].

3.5.3. The Church Game

In this section, we outline a proof of the decidability of the Church synthesis problem. The reader can refer to [30, 100] for details. A more detailed exposition of the links between games and reactive synthesis is also available in [9].

The Church synthesis problem can be modelled as a game: Eve is the system, and interacts with Adam, the environment. Thus, Adam provides an input $i \in I$, to which Eve responds with an output $o \in O$, and so on *ad infinitum*. A play thus models an infinite interaction, and it is winning whenever it satisfies the specification.

Definition 3.19 (Church Game) Formally, to a specification $S \subseteq I^\omega \times O^\omega$, we associate the following Gale-Stewart game G_S : The game arena has two vertices $V = \{v_\forall, v_\exists\}$. Edges are $v_\forall \xrightarrow{i} v_\exists$ for all $i \in I$, and $v_\exists \xrightarrow{o} v_\forall$ for all $o \in O$. A play $v_\forall i_0 v_\exists o_0 v_\forall i_1 v_\exists o_1 \dots$ is then winning if $\text{actions}(\rho) = i_0 o_0 i_1 o_1 \dots \in \langle S \rangle$, i.e. $(i_0 i_1 \dots, o_0 o_1 \dots) \in S$.

Now, in a similar way as Proposition 3.13, we can show that S admits a synchronous implementation if and only if Eve has a computable winning strategy in G_S , and that moreover, one can build a finite-state computable implementation if and only if Eve has a (computable) finite-memory winning strategy in G_S . Thus, the Church synthesis problem for a specification S can be reformulated as deciding whether Eve has a finite-memory winning strategy in G_S and, if yes, to compute it.

Our study thus leads us to the following result:

Theorem 3.17 ([8, 13]) *The Church synthesis problem is decidable.*

Moreover, the specification admits a synchronous implementation if and only if it admits one which is finite-state computable.

Proof. First, if the specification is given as a MSO formula, convert it to an ω -automaton A_φ such that $L(A_\varphi) = L(\varphi)$ (Theorem 3.6). Such automaton can then be converted into a deterministic parity automaton D . Finally, deciding whether φ is realisable amounts to deciding the winner of the synthesis game associated with D . As it is a parity game, it is positionally determined (Theorem 3.10), so the procedure yields a finite-memory strategy (even a positional one), which corresponds to a finite-state computable implementation if Eve is the winner. $\square$

[97]: Henzinger and Piterman (2006), ‘Solving Games Without Determinization’

[98]: Colcombet (2009), ‘The Theory of Stabilisation Monoids and Regular Cost Functions’

[99]: Kuperberg and Skrzypczak (2015), ‘On Determinisation of Good-for-Games Automata’

[30]: Thomas (2008), ‘Church’s Problem and a Tour through Automata Theory’

[100]: Thomas (2008), ‘Solution of Church’s Problem: A tutorial’

[9]: Bloem, Chatterjee, and Jobstmann (2018), ‘Graph Games and Reactive Synthesis’

[8]: J.R. Büchi and L.H. Landweber (1969), ‘Solving sequential conditions finite-state strategies’

[13]: Pnueli and Rosner (1989), ‘On the Synthesis of a Reactive Module’

The construction in [54] yields a non-deterministic Büchi automaton.

Register Automata over a Data Domain

4.

The goal of this work is to extend the study of the reactive synthesis problem to the case of infinite alphabets, where both the system and the environment pick their actions in an infinite domain. To model this setting, we use the formalism of data words, i.e. words over an infinite alphabet, called the data domain (cf Section 4.1). This domain consists in an infinite set along with a finite set of predicates over data, for instance $(\mathbb{N}, =)$ or $(\mathbb{Q}, <)$. Although they can be encoded in data values, we also equip data words with labels taken from a finite alphabet. This is done for modelling purposes: the setting of infinite alphabets aims at extending the study of synthesis problems from the *control* to the *data-processing* perspective, i.e. to be able to specify how the incoming data should be processed by the system.

The finite alphabet component allows to preserve the *control* aspect. More precisely, in the classical setting, labels indicate how the internal states of the system should evolve, what signals it should raise and at which moment. They also play this role in our setting, and additionally specify how the system should process the data, itself modelled as elements of an infinite alphabet. For instance, if we come back to the unmissable request-grant example (Example 3.5) where a server has to grant requests from its clients, one can enrich the modelisation by attributing an identifier to each client, and ask that the request of client i has to be matched by a grant with data value i . Executions of the system are then modelled as infinite sequences of pairs (σ, i) , where $\sigma \in (\text{req}, \text{idle}, \text{grt})$ indicates the role of its associated identifier (the data value associated with *idle* is irrelevant).

We express specifications with register automata [21, 22, 101] (defined in Section 4.3) that alternately read one input and one output element, as in the finite alphabet case (cf Section 3.4.3). Those machines consist in an ω -automaton equipped with a finite set of registers. On reading a data value, the machine compares it with the content of its registers with regard to the predicates of the domain. Depending on the result of the test, it then transitions to some state and overwrites the content of some registers (or none) with the data value it read. As this model is at the basis of our study, we dedicate this chapter to the study of some of its properties that prove useful for our applications. We consider two different semantics, namely non-determinism and its dual, universality, as well as the deterministic restriction.

Correspondingly, the target implementation is a synchronous sequential register transducer, i.e. a synchronous sequential transducer equipped with a finite set of registers. On reading an input data value, it compares it with the content of its registers and deterministically stores it in some of them (or none). It then outputs the content of one of its registers, and transitions to some state. Both the specification and implementation formalisms are detailed in the next chapter (Chapter 5), respectively in Sections 5.2 and 5.3.

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

[22]: Segoufin (2006), ‘Automata and Logics for Words and Trees over an Infinite Alphabet’

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

There is a slight terminology clash here: the universal semantics, which is to be understood as co-non-determinism, has little to do with the universality problem, that asks whether some automaton accepts all words.

Summary

4.1. Data Words	73
4.1.1. Labelled Data Words	74
4.2. Tests and Valuations	75
4.3. Register Automata	76
4.3.1. Initialisation of Registers	76
4.3.2. Tests	77
4.3.3. Operations over Data	77
4.3.4. The Model	78
4.4. General Properties	83
4.4.1. Link with ω -regular Languages	83
4.4.2. Projection over Labels	84
4.4.3. Union and Intersection	84
4.4.4. A Normal Form for Tests	86
4.4.5. On Assignments	87
Reassignments	87
Single-Assignment Register Automata	89
4.4.6. Closure under Automorphisms	90
4.5. Non-Deterministic Register Automata with Equality	92
4.5.1. Constraints	92
4.5.2. The Renaming Property	95
Projection over Labels	96
Emptiness Problem	97
Universality Problem	97
4.5.3. Local Concretisability	98
Explicit Tests	99
Update of Constraints	100
Ensuring Local Concretisability	100
4.5.4. Action Sequences	102
Feasible Action Sequences	102
4.5.5. Constraints and Types	104
Types	104
4.6. Non-Deterministic Register Automata with a Dense Order	106
4.7. Register Automata with a Discrete Order	108
4.7.1. Non-Regular Behaviours	108

4.7.2. Constraint Sequences	109
Feasibility of Constraint Sequences	110
4.8. Universal Register Automata	118
4.9. Register Automata and Arithmetic	121
4.9.1. Register Automata over $(\mathbb{N}, +1, 0)$	121
4.9.2. Register Automata over $(\mathbb{N}, \times, 2, 3)$	121
4.10. Extensions of the Model	122
4.10.1. Non-Deterministic Reassignment	122
Closure Properties	123
4.10.2. Fresh-Register Automata	127

4.1. Data Words

Definition 4.1 (Data domain, data words) A *data domain* is a triple $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ where $\mathbb{D}$ is a countably infinite set of *data values*, $\mathcal{R}$ is a finite set of interpreted *relational symbols*, also known as *predicates* and $\mathcal{C}$ a finite set of interpreted *constant symbols*, disjoint from $\mathcal{R}$. The set $\mathcal{S} = \mathcal{R} \sqcup \mathcal{C}$ is called the *signature* of $\mathcal{D}$. We assume that the symbol $=$ is in $\mathcal{R}$, and we interpret it as the equality relation.

A *data ω -word*, or simply *data word*, is an infinite sequence of data $x = d_0 d_1 \dots \in \mathbb{D}^\omega$. *Finite data words* are defined analogously.

Assumption 4.2 To be able to initialise registers, we assume that there is at least one distinguished constant symbol $\#$. This assumption is benign: given a data domain $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ with no constants, one can instead consider $\mathcal{D}^\# = (\mathbb{D} \sqcup \{\#\}, \mathcal{R}, (\mathcal{C} \sqcup \{c_0\}))$, where c_0 is a new constant symbol interpreted as $\#$, and the equality relation is extended to include $(\#, \#)$.

Definition 4.3 (Data language) A *data word language*, or simply *data language*, is a set of data words $L \subseteq \mathbb{D}^\omega$.

Remark 4.1 Actually, we further require that it is closed under predicate-preserving automorphisms (cf Definition 4.18, recalled in the margin). This notion is made precise in Part II under the name of equivariance (Definition 12.2) but the reader does not need to bother with it now, since this will be the case of all the languages that we manipulate. Indeed, data languages recognised by register automata are necessarily equivariant, see [21] and [25]. We give an intuition of the notion in Example 4.1 and prove the result for our setting in Proposition 4.10.

We consider the following data domains:

- A countably infinite set $\mathbb{D}$ with the equality predicate $(\mathbb{D}, =, \mathcal{C})$, with a finite set of constants $\mathcal{C} \subset_f \mathbb{D}$ containing $\#$ whose choice is irrelevant. This setting is isomorphic to $(\mathbb{N}, =, \{n_1, \dots, n_c\})$ for some arbitrary $n_1, \dots, n_c \in \mathbb{N}$, where $c = |\mathcal{C}|$ (in particular, one can take $\{1, \dots, c\}$).
- The set of rational numbers with its usual order: $(\mathbb{Q}, <, 0)$. Again, the choice of $\#$ does not matter; we simply choose 0 for simplicity.

Data domains without equality constitute pathological cases that we want to rule out.

For simplicity, in the following, we confuse the symbols with their interpretation.

Note that data words are infinite, otherwise they are called finite data words.

μ is an automorphism of $\mathcal{D}$ if for all relations P of arity k and for all k -tuples $(d_1, \dots, d_k), (d'_1, \dots, d'_k) \in P \Leftrightarrow (\mu(d_1), \dots, \mu(d_k)) \in P$, and $\mu(c) = c$ for all constants. $L \subseteq \mathbb{D}^\omega$ is equivariant when for all such automorphisms, $\mu(L) = L$.

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

[25]: Bojańczyk, Klin, and Lasota (2014), ‘Automata theory in nominal sets’

For readability, we write $(\mathbb{D}, =, \#)$ instead of $(\mathbb{D}, \{=\}, \{\#\})$. In the following, we generally omit the braces when no ambiguity arises, and we do not explicitly include $=$ when it can be derived. We might also omit $\#$ when its choice is irrelevant.

- The set of natural numbers with its usual order and 0: $(\mathbb{N}, <, 0)$. Here, being able to access 0 is important, as it is the minimal element of $\mathbb{N}$ with regard to $<$.

Part II describes a setting which encompasses both $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, 0)$, namely oligomorphic data domains.

Remark 4.2 Note that we do not consider additional constants in our treatment of $(\mathbb{Q}, <, 0)$, as it induces considerable notational overhead. However, the study can be extended to this setting, and we give the main ideas on how to do this. As distinguishing constants does not affect oligomorphicity, such an extension also results from our study of oligomorphic data domains (see Section 12.2.1 and Proposition 12.15.)

Example 4.1 Consider the following sets:

- The first data value appears again:

$$L_{\text{first}} = \{d_0 d_1 \dots \in \mathbb{N}^\omega \mid \exists i > 0, d_i = d_0\}$$

- At least two data are equal:

$$L_{\exists=} = \{d_0 d_1 \dots \in \mathbb{N}^\omega \mid \exists i \neq j, d_i = d_j\}$$

- All data are distinct:

$$L_{\forall \neq} = (L_{\exists=})^c = \{d_0 d_1 \dots \in \mathbb{N}^\omega \mid \forall i \neq j, d_i \neq d_j\}$$

- Data are in increasing order, in $\mathbb{N}$:

$$L_{<}^{\mathbb{N}} = \{d_0 d_1 \dots \in \mathbb{N}^\omega \mid \forall i < j, d_i < d_j\}$$

- Data are in increasing order, in $\mathbb{Q}$:

$$L_{<}^{\mathbb{Q}} = \{d_0 d_1 \dots \in \mathbb{Q}^\omega \mid \forall i < j, d_i < d_j\}$$

- Consider a deterministic Minsky machine [102] M . Define the language of the successive values of its counters:

$$L_M = \left\{ c_1^0 c_2^0 c_1^1 c_2^1 \dots \in \mathbb{N}^\omega \mid \begin{array}{l} \text{for } i = 1, 2; c_i^0 c_i^1 \dots \text{ is the sequence} \\ \text{of values taken by counter } i \end{array} \right\}$$

According to the above definition, all of them are data languages over $(\mathbb{N}, =, 0)$, except for $L_{<}^{\mathbb{Q}}$ which is defined over $\mathbb{Q}$. However, $L_{<}^{\mathbb{N}}$ intuitively does not belong to this data domain, as it relies on an ordering of $\mathbb{N}$, which cannot be expressed in $(\mathbb{N}, =, 0)$, and it can indeed be ruled out as it is not equivariant. It however belongs to the richer domain $(\mathbb{N}, <, 0)$.

A similar argument applies to L_M , as increment and decrement cannot be expressed in $(\mathbb{N}, =, 0)$. Furthermore, domains which can express those operations yield a model that is too expressive, due to its equivalence with Minsky machines (see Property 4.51). Note that testing for 0 also requires the constant 0 in the domain.

4.1.1. Labelled Data Words

As explained above, we equip data words with labels as it better models the reactive synthesis setting. Note that this could be done internally, by

[102]: Minsky (1961), 'Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines'

A *Minsky machine* is a machine with finitely many *states*, equipped with two *counters* which take their values in $\mathbb{N}$, and that it can *increment*, *decrement* and *test for 0*.

$L_{<}^{\mathbb{N}}$ is not equivariant in $(\mathbb{N}, =, 0)$: by taking the transposition t of 1 and 2 (i.e. $t(1) = 2, t(2) = 1$ and $t(n) = n$ for all $n \neq 1, 2$), which is an automorphism of $\mathbb{N}$ preserving $=$ and 0 , we have that $w = 123 \dots \in L_{<}^{\mathbb{N}}$ but $t(w) = 213 \dots \notin L_{<}^{\mathbb{N}}$. Note that t is not an automorphism of $(\mathbb{N}, <)$ as it does not preserve $<$.

As witnessed by L_M , register automata over $(\mathbb{N}, +1, 0)$ can recognise accepting runs of Minsky machines, which implies that the emptiness problem for deterministic register automata over this domain (and richer ones) is undecidable (see Theorem 4.52).

instead considering the data domain $\Sigma \times \mathcal{D}$, where Σ is the finite alphabet of labels and the signature is extended by adding predicates for labels while interpreting existing predicates on the data component (cf Remark 10.1).

Definition 4.4 (Labelled Data Word) Let Σ be a finite alphabet of *labels*, and $\mathcal{D}$ be a data domain. A *labelled data value* over data domain $\mathcal{D}$ with labels Σ is a pair $(\sigma, d) \in \Sigma \times \mathcal{D}$. Its label is $\text{lab}(\sigma, d) = \sigma$ and its data value is $\text{dt}(\sigma, d) = d$. A *labelled data ω -word* is then an infinite sequence of labelled data, i.e. $x = (\sigma_0, d_0)(\sigma_1, d_1) \cdots \in (\Sigma \times \mathcal{D})^\omega$. lab and dt are extended pointwise, i.e. $\text{lab}(x) = \sigma_0\sigma_1 \dots$ and $\text{dt}(x) = d_0d_1 \dots$.

In the rest of this chapter, we sometimes say *data word* for labelled data ω -word, as no ambiguity arises. In Part II, we drop the explicit treatment of labels, since it induces unnecessary notational complexity.

Convention 4.5 In the following, we fix a finite alphabet Σ of labels and a data domain $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$.

4.2. Quantifier-Free Formulas, Tests and Valuations

In our setting, tests of the automaton over input data are modelled as quantifier-free formulas, and the content of the registers is a valuation of variables. This section defines these concepts taken from logics, as well as the notion of type that proves useful for abstracting the behaviour of our model.

Definition 4.6 (Quantifier-free formula over $\mathcal{D}$, Valuation) Let X a finite set of *variables*. We consider the set of *quantifier-free formulas* φ in $\mathcal{D}$ over variables X , which is generated by the following grammar:

$$\varphi ::= P(t_1, \dots, t_k) \mid \top \mid \perp \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg \varphi$$

where P is a relation of $\mathcal{R}$ of *arity* k for some $k \in \mathbb{N}$ and $(t_1, \dots, t_k)$ is a k -tuple of terms, a *term* being a constant of $\mathcal{C}$ or a variable in X . Note that to ease the writing of tests, we defined a grammar that is not minimal, as we included $\top := \varphi \vee \neg \varphi$ (for some formula φ not containing $\top$ nor $\perp$), $\perp := \neg \top$ and $\varphi \wedge \psi := \neg(\neg \varphi \vee \neg \psi)$.

Definition 4.7 (Valuation) A *valuation* of variables X over data domain $\mathcal{D}$ is a total function $v : X \rightarrow \mathbb{D}$. We denote by $\text{Val}_{\mathcal{D}}(X) = X^{\mathbb{D}}$ the set of valuations of X over $\mathcal{D}$.

Given a variable a (not necessarily in X) and a data value $d \in \mathbb{D}$, we define the corresponding *update* of v with regard to a and d as

$$v\{a \leftarrow d\} : \begin{cases} x \neq a & \mapsto v(x) \\ a & \mapsto d \end{cases}$$

Remark 4.3 Note that if $a \notin X$, this operation extends the domain of v .

When several updates $a_1 \leftarrow d_1, \dots, a_n \leftarrow d_n$ are conducted, we write $v\{a_1 \leftarrow d_1, \dots, a_n \leftarrow d_n\}$ for $v\{a_1 \leftarrow d_1\} \dots \{a_n \leftarrow d_n\}$. Finally, for a set

We often write $\varphi(X)$ to explicit the fact that φ is over variables X .

$P(\mathbf{t})$ is called an *atomic formula*.

$A = \{a_1, \dots, a_n\}$, $v\{A \leftarrow d\}$ is a shorthand for $v\{a_1 \leftarrow d, \dots, a_n \leftarrow d\}$, with the convention that $v\{\emptyset \leftarrow d\} = v$.

Satisfaction of a formula $\varphi(X)$ by a valuation v over variables X , written $v \models \varphi$, is defined as usual, by induction over φ :

- ▶ $v \models P(t_1, \dots, t_k)$ if $P(v(t_1), \dots, v(t_k))$ holds in $\mathcal{D}$, where v is extended to constants by setting $v^{\mathcal{C}}(c) = c$ for all constants $c \in \mathcal{C}$.
- ▶ $v \models \top$
- ▶ $v \not\models \perp$
- ▶ $v \models \varphi \wedge \psi$ if both $v \models \varphi$ and $v \models \psi$
- ▶ $v \models \varphi \vee \psi$ if $v \models \varphi$ or $v \models \psi$
- ▶ $v \models \neg\varphi$ if $v \not\models \varphi$

A formula is *satisfiable* if there exists a valuation which satisfies it.

Example 4.2 Over data domain $(\mathbb{N}, =, 0)$, the valuation v defined as $v(r_1) = 0$, $v(r_2) = 1$ and $v(r_3) = 1$ satisfies $\varphi_1 := \neg(r_1 = r_2)$ and $\varphi_2 := r_2 = r_3$, but not $\varphi_3 := r_1 = r_3$. It also satisfies $\varphi_4 := r_1 = 0$.

The same valuation seen as belonging to $(\mathbb{Q}, <, 0)$ satisfies $\varphi_5 := r_1 < r_2$ but not $\varphi_6 := \neg(r_1 < r_2 \vee r_2 < r_1)$. Note that φ_6 holds whenever $v(r_1) = v(r_2)$, meaning that equality can be derived in $(\mathbb{Q}, <)$, which can thus simulate $(\mathbb{N}, =)$.

Assumption 4.8 In the following, we assume that the satisfaction of a formula by a given valuation can be decided.

As usual, equality is written infix, i.e. $r_1 = r_2$ means $=(r_1, r_2)$, and similarly for $<$.

We will mostly be concerned with data domains where the satisfiability of a formula can be checked as well.

4.3. Register Automata

Register automata were introduced for finite data words over data domain $(\mathbb{D}, =)$ by Kaminski and Francez under the name of *finite-memory automaton* [21]. The aim was to generalise finite-state automata to the setting of infinite alphabets. This model consists in a finite-state automaton, which is equipped with a finite set of registers, used to store data. These registers are initialised with a distinguished value $\#$. On reading an input data value, the automaton compares it with the content of its registers (with regard to the predicates of the data domain) by conducting a *test*. It then stores it in some (or none) of its registers, conducting an *assignment*, and transitions to some state.

Other automata models have been defined for the infinite alphabet setting [103], but register automata offer an interesting compromise between expressiveness and decidability, even if they do not enjoy all the closure properties of finite-state automata. We first present the different aspects of the model, before defining it.

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

Depending on the application, it can be better to assume that $\#$ does not belong to the data domain, and can be thought of as `null`. This can be simulated (cf Remark 4.5), but we do not assume it in general as it induces considerable notational overhead.

See [103]: Björklund and Schwentick (2010), ‘On notions of regularity for data languages’ for a study of the various automata models over infinite alphabets.

4.3.1. Initialisation of Registers

As explained above, we need a special value to initialise registers. We use the distinguished constant $\#$.

Definition 4.9 (Initial valuation) Given a set R of registers, we denote $v_R^{\#} : R \rightarrow \mathbb{D}$ the constant valuation defined, for all $r \in R$, by $v_R^{\#}(r) = \#$.

4.3.2. Tests

Register automata conduct tests over input data with regard to a valuation of their registers. A test is a quantifier-free formula over $\mathbb{D}$ with a distinguished variable denoting the input data value.

Remark 4.4 Tests could have been defined as existential first-order formulas without affecting our results. We choose quantifier-free formulas for simplicity's sake. It does not make any difference for structures which admit quantifier elimination such as $(\mathbb{N}, \{=\})$ or $(\mathbb{Q}, \{<\})$. Except for those structures however, allowing the full-fledged power of first-order logic would be too too strong. For instance, from $(\mathbb{N}, <, 0)$, one can define the successor relation $+1$ in FO, which yields a model that is equivalent to Minsky machines (see Property 4.51), that are Turing complete [102, Theorem I].

Definition 4.10 (Test) Let R be a finite set of variables, that we call registers, and let $\star \notin R$ be a variable, denoting the input data value. A test in $\mathbb{D}$ over R is a quantifier-free formula ϕ over variables the $R \sqcup \{\star\}$.

The set of tests in $\mathbb{D}$ over R is denoted $\text{Tests}_{\mathbb{D}}(R)$. It is infinite in general, but finite up to logical equivalence (see Proposition 4.5).

For a valuation $\nu : R \rightarrow \mathbb{D}$, a data value $d \in \mathbb{D}$ and a test ϕ , we often write $\nu, d \models \phi$ for $\nu\{\star \leftarrow d\} \models \phi$.

Remark 4.5 It is sometimes relevant to assume that the initial content of registers $\#$ does not appear in the input. One can restrict to data values that are distinct from $\#$ by considering tests of the form $\phi \wedge (\star \neq \#)$, since $\#$ is a constant of the domain.

Convention 4.11 In the following, we let $\star$ be a distinguished variable, that we use to denote the input data value. Implicitly, we always assume $\star \notin R$.

[102]: Minsky (1961), 'Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines'

$\mathbb{D}$ is omitted when clear from the context.

Recall that we assume that equality can be expressed. $\star \neq \#$ is a short for $\neg(\star = \#)$.

4.3.3. Operations over Data

We now explain how data is processed, as it eases the formalisation of the semantics. Initially, registers start with the initial valuation ν_R^i . On reading a data value, a register automaton first tests it, and then assigns it to some register.

Definition 4.12 (Assignments, Actions and Data Processing) Given a set of registers R , an *assignment* over R is a subset $\text{asgn} \subseteq R$. An *action* is a pair $(\phi, \text{asgn}) \in \text{Tests}_{\mathbb{D}}(R) \times 2^R$. The set of actions is denoted $\text{Actions}_{\mathbb{D}}(R) = \text{Tests}_{\mathbb{D}}(R) \times 2^R$.

Data processing is then described through a labelled transition system $\mathcal{T}_R^{\mathbb{D}}$, called the *data processing transition system* over R . Its configurations are register configurations, i.e. valuations. It is labelled with pairs $(d, (\phi, \text{asgn}))$, where d is a data value and (ϕ, asgn) is an action. In register configuration ν , on reading a data value $d \in \mathbb{D}$, when it executes action (ϕ, asgn) , the machine:

- (i) Tests whether d satisfies the test ϕ from valuation ν ($\nu, d \models \phi$)
- (ii) Assigns d to all registers of asgn ($\nu\{\text{asgn} \leftarrow d\}$)

Formally, we define $\mathcal{T}_R^{\mathcal{D}} = (\text{Val}_{\mathcal{D}}(R), \nu_R^i, \mathbb{D} \times \text{Actions}_{\mathcal{D}}(R), \rightarrow)$ where, for two valuations ν and λ , we have

$$\nu \xrightarrow[\mathcal{T}_R^{\mathcal{D}}]{d, (\phi, \text{asgn})} \lambda \text{ whenever } \begin{cases} \nu\{\star \leftarrow d\} \models \phi & \text{and} \\ \lambda = \nu\{\text{asgn} \leftarrow d\} \end{cases}$$

4.3.4. The Model

As we are mainly concerned with infinite data words, we define register automata over labelled data ω -words.

Definition 4.13 (Register Automaton) An ω -register automaton, or simply register automaton, over data domain $\mathcal{D}$ is a tuple $A = (Q, I, \Sigma, R, \Delta, \Omega)$, where:

- Q is a finite set of *states*
- $I \subseteq Q$ is a finite set of *initial states*
- Σ is a finite alphabet of labels
- R is a finite set of *registers*
- $\Delta \subseteq_f Q \times \Sigma \times \text{Tests}_{\mathcal{D}}(R) \times 2^R \times Q$ is a finite set, called the *transition relation*. We write $p \xrightarrow[A]{\sigma, \phi, \text{asgn}} q$ for $t = (p, \sigma, \phi, \text{asgn}, q) \in \Delta$.
- $\Omega \subseteq Q^\omega$ is the *acceptance condition*. As for ω -automaton, it can be any ω -regular condition. The acceptance conditions we use in this chapter are:

Büchi $\Omega = \{\chi \in Q^\omega \mid \text{Inf}(\chi) \cap F \neq \emptyset\}$ for F a set of accepting states.

co-Büchi $\Omega = \{\chi \in Q^\omega \mid \text{Inf}(\chi) \cap F = \emptyset\}$ for F a set of rejecting states.

Parity $\Omega = \{\chi \in Q^\omega \mid \max \text{Inf}(c(\chi)) \text{ is even}\}$, where $c : Q \rightarrow \{1, \dots, d\}$ is a parity function.

Remark 4.6 On top of this shift to infinite words with labels, our definition departs from the seminal one on the following aspects:

- Registers are initialised with constants from the data domain
- Registers can take duplicate values
- Tests are expressed as formulas
- The model is defined for any data domain

Over $(\mathbb{D}, =, \mathbb{C})$, those models are essentially equivalent but this definition yields flexibility (Remark 4.8) and succinctness (cf Remark 4.14).

Register automata with a Büchi (respectively co-Büchi, parity) condition are called *Büchi* (resp. *co-Büchi*, *parity*) register automata.

Configurations A configuration of A is a pair $C = (q, \nu)$ where $q \in Q$ and $\nu : R \rightarrow \mathbb{D}$. A configuration $C' = (q', \nu')$ is a *successor* of C on reading the labelled data value $(\sigma, d) \in \Sigma \times \mathbb{D}$ following transition $t = q \xrightarrow[A]{a, \phi, \text{asgn}} q'$, written $C \xrightarrow[A]{\sigma, d, \phi, \text{asgn}}_t C'$, if:

- (i) The labels coincide, i.e. $\sigma = a$
- (ii) The input data value d satisfies the test ϕ from configuration (q, ν) , i.e. $\nu\{\star \leftarrow d\} \models \phi$

When Σ is unary, we omit the mention of labels.

Assuming Δ is finite yields a finitely presented model and is harmless since there are finitely many non-equivalent tests.

As for ω -automata and transducers, A and t can be omitted when depicting transitions if they are clear from the context.

Recall that for a word $w \in X^\omega$ (for some set X), $\text{Inf}(w)$ is defined as $\text{Inf}(w) = \{a \in X \mid w[i] = a \text{ for infinitely many } i \in \mathbb{N}\}$.

- (iii) v is *updated* to v' with regard to asgn and d , i.e. $v' = v\{\text{asgn} \leftarrow d\}$ (d is assigned to all registers in asgn).

Note that items (ii) and (iii) can be summarised as $v \xrightarrow[\mathcal{T}_R^{\mathcal{D}}]{d, \phi, \text{asgn}} v'$, where

$\mathcal{T}_R^{\mathcal{D}}$ is the transition system of Definition 4.12.

We then say that d allows to *cross* transition t from valuation v . We denote $\text{Configs}(A) = Q \times \text{Val}_{\mathcal{D}}(R)$ the set of configurations of A .

As usual, A and t can be omitted. We also omit ϕ and/or asgn when they are not needed.

Runs A run of A over the labelled data word $w = (\sigma_0, d_0)(\sigma_1, d_1) \dots$ is an infinite sequence

$$\rho = C_0 t_0 C_1 t_1 \dots \in (\text{Configs}(A) \Delta)^{\omega} \text{ such that for all } i \in \mathbb{N}, C_i \xrightarrow[A]{\sigma_i, d_i}_{t_i} C_{i+1}.$$

We define:

- $\text{trans}(\rho) = t_0 t_1 \dots \in \Delta^{\omega}$
- $\text{configs}(\rho) = C_0 C_1 \dots \in \text{Configs}(A)^{\omega}$

By writing $C_i = (q_i, v_i)$ for all $i \in \mathbb{N}$, we moreover let

- $\text{states}(\rho) = q_0 q_1 \dots \in Q^{\omega}$ and
- $\text{valuations}(\rho) = v_0 v_1 \dots \in \text{Val}_{\mathcal{D}}(R)^{\omega}$

A run ρ is *initial* if $q_0 \in I$ and $v_0 = v_R^l$. It is moreover *accepting* if its sequence of states belongs to the acceptance condition, i.e. $\text{states}(\rho) \in \Omega$.

Partial Runs *Partial runs* are defined analogously, as finite sequences of configurations and transitions that are consistent with the successor relation. A configuration C is *reachable* if there exists a partial run that is initial and that ends in C .

Syntactical Automaton Note that a register automaton A can be seen as an ω -automaton $A^{\text{synt}} = (Q, I, \Sigma \times \text{Actions}_{\mathcal{D}}(R), \Delta, \Omega)$ that we call the *syntactical ω -automaton*. It recognises the (ω -regular) language $L_A^{\pi} \subseteq (\Sigma \times \text{Actions}_{\mathcal{D}}(R))^{\omega}$ of paths of the automaton. They do not necessarily correspond to runs, as there might not exist a data word that is consistent with the sequence of actions (see Section 4.5.3).

Remark 4.7 A register automaton is an extension of an ω -automaton which has infinitely many states ($Q = \text{Configs}(A)$), processing inputs from an infinite alphabet $\Sigma \times \mathbb{D}$.

Semantics As for ω -automata, register automata can be endowed with dual semantics. A *non-deterministic register automaton* A (NRA for short) recognises the data language:

$$L_N(A) = \{w \in (\Sigma \times \mathbb{D})^{\omega} \mid \text{there exists an accepting run } \rho \text{ on } w\}$$

Dually, a *universal register automaton* A (URA for short) recognises the data language:

In particular, if $w \in (\Sigma \times \mathbb{D})^{\omega}$ does not admit any run, it is accepted.

$$L_U(A) = \{w \in (\Sigma \times \mathbb{D})^\omega \mid \text{all initial runs } \rho \text{ on } w \text{ are accepting}\}$$

Given a non-deterministic register automaton A , we can define a universal register automaton A^c which is a copy of A with acceptance condition $\Omega^c = Q^\omega \setminus \Omega$; we then have $L_U(A^c) = (\Sigma \times \mathbb{D})^\omega \setminus L_N(A)$. If the semantics is clear from the context, we simply write $L(A)$.

Determinism A is a *deterministic register automaton* (DRA for short) if it has a unique initial state and for all reachable configurations $(p, v) \in \text{Configs}(A)$ and for all labelled data $(\sigma, d) \in \Sigma \times \mathbb{D}$, there exists at *most* one transition $t = p \xrightarrow[A]{\sigma, \phi, \text{asgn}} q$ such that $p \xrightarrow[A_t]{(\sigma, d)} q$. This semantical property is equivalent to the syntactic condition that tests over the input data are mutually exclusive, i.e., for any two distinct transitions of the form $q \xrightarrow{a, \phi, \text{asgn}} q'$ and $q \xrightarrow{b, \psi, \text{asgn}'} q''$, either $a \neq b$ or $\phi \wedge \psi$ is not satisfiable. However, contrary to the finite alphabet case, deterministic register automata are strictly less expressive than non-deterministic ones (cf Section Subsection 4.3.4 on the following page).

Completeness A is said to be *complete* if for all reachable configurations $(p, v) \in \text{Configs}(A)$ and for all $(\sigma, d) \in \Sigma \times \mathbb{D}$, there exists at *least* one transition $t = p \xrightarrow[A]{\sigma, \phi, \text{asgn}} q$ such that $p \xrightarrow[A_t]{\sigma, d} q$. Completeness can be enforced without loss of expressive power by the syntactic condition that for all $p \in Q$ and all $\sigma \in \Sigma$, the disjunction of all tests is valid, i.e. $\bigvee \left\{ \phi \in \text{Tests}(R) \mid p \xrightarrow{\sigma, \phi, \text{asgn}} q \text{ for some asgn and } q \right\}$ is valid.

The class of languages recognised by non-deterministic (respectively, universal, deterministic) register automata is written NRA (resp., URA, DRA).

Example 4.3 Consider again the data languages of Example 4.1. They are recognised by the register automata of Figure 4.1 on the next page.

Definition 4.14 (Register automaton over finite data words) Register automata can also be defined over finite data words, by replacing the acceptance condition with a finite set F of *accepting states*. The notion of configuration is unchanged, and the notions of run and recognised language are easily adapted. Unless otherwise stated, the properties that we exhibit also hold in the finite data word case.

We do not define register automata over finite data words formally as we only use them as a tool.

In the remainder of this chapter, we explore in some depth the properties of register automata that we use in our study, and in particular the expressiveness and closure properties under the different semantics. However, let us already make a few remarks, to give a general idea of the behaviour of the model.

Remark 4.8 (Variations on the Theme) Many models of register automata coexist in the literature, depending on different parameters:

- whether they process finite or infinite data words
- whether labels are explicitly handled
- by restricting the form of tests
- whether registers are allowed to take duplicate values
- whether registers are initialised or not, and

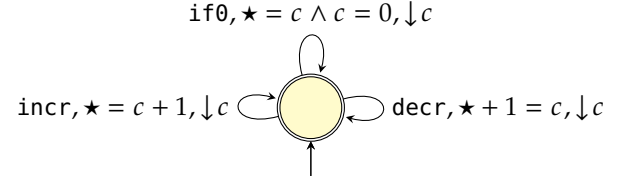
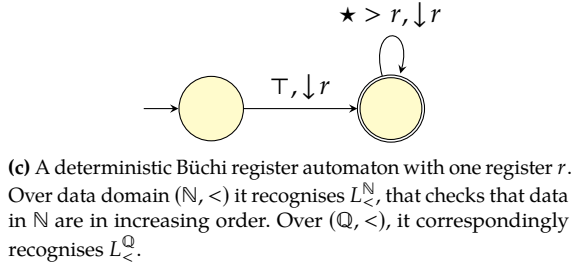
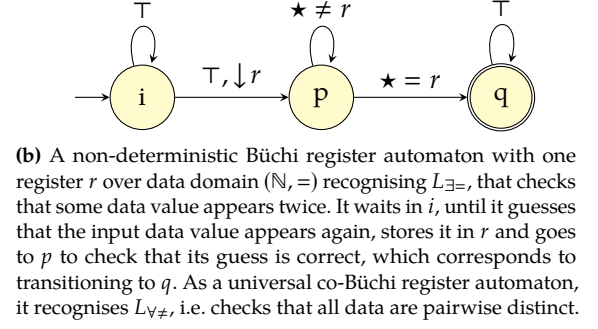
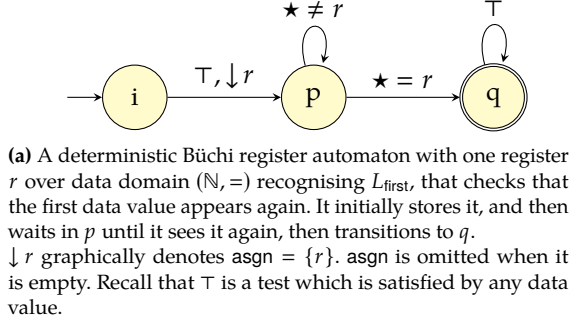


Figure 4.1: A series of register automata, recognising the data languages of Example 4.1. Recall that when it is unary, the labels alphabet is omitted.

► whether registers can be erased along the run.

Murawski, Ramsay, and Tzevelekos [104] provide a comparison of the different models with regard to bisimilarity checking. All those variants morally have the same expressive power for data domains with equality only, although allowing duplicate values increases succinctness, and thus affects the complexity of the emptiness problem (cf Remark 4.14).

Expressiveness under the different semantics Contrary to ω -automata, the different semantics are not equivalent for all the data domains we consider. In $(\mathbb{D}, =)$, deterministic register automata are strictly less expressive than non-deterministic ones: the data language $L_{\exists=} = \{d_0 d_1 \dots \in \mathbb{N}^\omega \mid \exists i \neq j, d_i = d_j\}$ recognised by the non-deterministic register automaton of Figure 4.1b cannot be recognised by a deterministic register automaton [21, Remark 2]. Moreover, data languages recognised by non-deterministic register automata are not closed under complement, as witnessed by $(L_{\exists=})^c$ [21, Proposition 5] (see also Property 4.18). As a consequence, non-deterministic and universal register automata recognise dual classes of data languages. The same example works for data domains $(\mathbb{Q}, <)$ and $(\mathbb{N}, <)$.

One can further define *alternating register automata*, by partitioning states into *non-deterministic* and *universal states*, as was done for ω -automata. This class of automata strictly contains both classes above. Since its emptiness problem over finite words is non-elementary for one register, and undecidable for infinite words and for more registers [28, Theorems 4.2 and 4.1], we focus on the non-deterministic, universal and deterministic semantics. Their associated decision problems will be treated in due course (Section 4.5 and following ones).

[104]: Murawski, Ramsay, and Tzevelekos (2015), ‘Bisimilarity in Fresh-Register Automata’

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

See also [29]: Figueira (2010), ‘Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données)’, *PhD thesis*.

In contrast to this strict hierarchy, it was freshly established that over data domains with equality only, a language of finite data words that is both recognisable by a non-deterministic and a universal register automaton is also recognisable by a deterministic register automaton. In short, $\text{NRA} \cap \text{URA} = \text{DRA}$ over $(\mathbb{D}, =)$ [105, Theorem 1]. The proof is rather involved, and the property breaks over ordered data domains. It also breaks when non-deterministic reassignment (cf *infra*) is allowed. The result however holds for a large class of data domains when only one register is allowed.

Two-way Model One can allow register automata to go back and forth on their input, as was done for two-way finite automata [106, Section 7]. This does not increase expressive power in the finite alphabet case [106, Theorem 15]. However, in the data words case, two-way register automata are strictly more expressive [26]. We do not explore this formalism, as its emptiness and universality problems are both undecidable [101, Theorem 5.3]; this dampens hopes of getting decidability result for synthesis problems, as those are more complex.

Single-Use Register Automata Recently, Bojańczyk and Stefanski considered the *single-use restriction*, that asks that a register is immediately emptied after it is tested [107, 108]. Emptying a register is not allowed in our formalism, but this is equivalent to asking that it is reset to the initial data value. They showed that it yields a model that is robust, in the sense that it can be presented in different ways: the two-way model is equivalent with the one-way one, itself equivalent to orbit-finite regular list functions [108, Theorem 6] (we refer to the paper for the definitions). Moreover, the class of recognised data languages can be characterised algebraically, as languages recognised by *orbit-finite* monoids (a notion introduced in [109]), that are equivalent to rigidly guarded MSO logic [110, Theorems 4.2 and 5.1]. This comes at the price of a loss of expressiveness: for instance, the language L_{first} of data words whose first data value appears again cannot be recognised by any single-use register automaton [108, Example 7], while it can be recognised by a deterministic register automaton (Example 4.3). As pointed out in [103], in the realm of data words it seems impossible to have one's cake and eat it too as there is a fragile balance between expressiveness, closure properties, robustness and decidability.

Guessing Another relevant extension is non-deterministic reassignment, also known as *guessing*: on taking a transition, the automaton is allowed to guess a data value and assign it to some of its registers [38]. This feature strictly increases expressiveness [38, Remark 5] (see also Example 4.9). This extension is presented in Section 4.10.1, and included in the model in Part II.

As stated in [26, Section 1.4], most of these variations are inequivalent. Figure 4.2 on the following page reproduces a figure from this book that summarises these results.

[105]: Klin, Lasota, and Torunczyk (2021), 'Nondeterministic and co-Nondeterministic Implies Deterministic, for Data Languages'

[106]: Rabin and Scott (1959), 'Finite Automata and Their Decision Problems'

[26]: Bojańczyk (2019), *Atom Book*

[101]: Neven, Schwentick, and Vianu (2004), 'Finite state machines for strings over infinite alphabets'

[107]: Bojańczyk and Stefanski (2019), 'Single use register automata for data words'
[108]: Bojańczyk and Stefanski (2020), 'Single-Use Automata and Transducers for Infinite Alphabets'

[109]: Bojanczyk (2011), 'Data Monoids'

[110]: Colcombet, Ley, and Puppis (2015), 'Logics with rigidly guarded data tests'

[103]: Björklund and Schwentick (2010), 'On notions of regularity for data languages'

[38]: Kaminski and Zeitlin (2010), 'Finite-Memory Automata with Non-Deterministic Reassignment'

[26]: Bojańczyk (2019), *Atom Book*

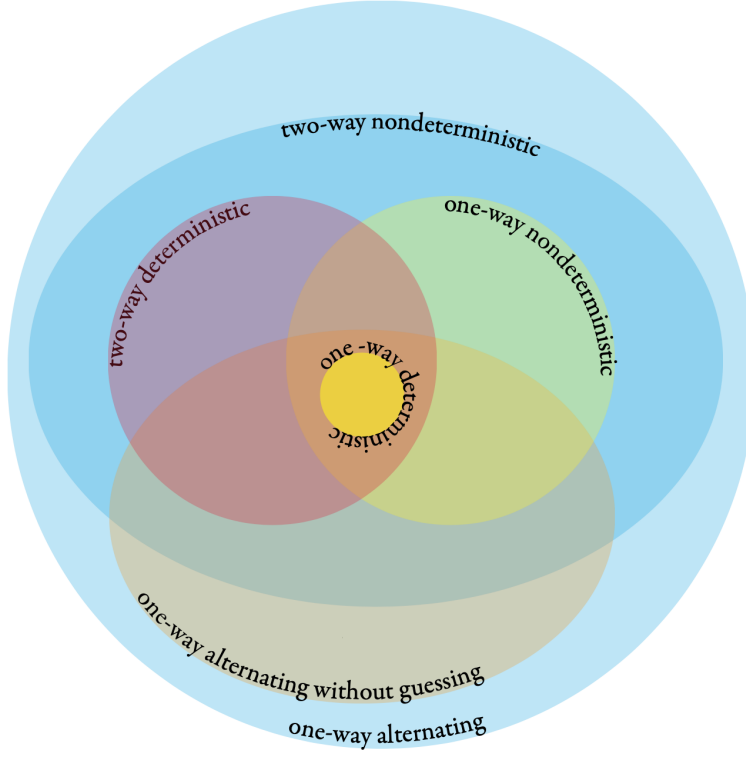


Figure 4.2.: Comparison of the different variations of register automata over $(\mathbb{D}, =)$. The term *guessing* is to be understood as non-deterministic reassignment, but the inclusions also hold when it is disallowed (except obviously for the strict inclusion between one-way alternating *without* guessing and one-way alternating) Source: [26, Figure 1.1 p.24].

4.4. General Properties of Register Automata

We now detail elementary properties of the model that we need for our study, and which hold independently of the data domain we consider.

4.4.1. Link with ω -regular Languages

First, one can notice that a register automaton with no registers corresponds to an ω -automaton over Σ . It is not much harder to check that when restricted to a finite subset of data values, register automata exactly recognise ω -regular languages.

Proposition 4.1 ([21, Proposition 1]) *Let A be a register automaton over data domain $\mathcal{D}$. For any finite subset $X \subseteq \mathbb{D}$, $L(A) \cap (\Sigma \times X)^\omega$ is ω -regular.*

More precisely, $L(A) \cap (\Sigma \times X)^\omega$ is recognised by an ω -automaton A_X with $n(s+1)^k$ states, where n is the number of states of A , k its number of registers, and $s = |X|$ is the size of X . Moreover, if A is deterministic, so is A_X .

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

Proof. Our setting is more general, as we allow richer data domains, but the proof of [21, Proposition 1] can easily be generalised. The idea is that there are finitely many possible configurations, as registers take their values in X , so they can be stored in the states. Assumption 4.8 ensures that the existence of a transition on input $(a, d) \in \Sigma \times X$ can be decided.

Formally, given a register automaton $A = (Q, I, \Sigma, R, \Delta, \Omega)$ and a finite set X , we define an ω -automaton A_X with states $Q_X = Q \times (X \sqcup \{\#\})^R$. Its initial states are $I \times \{v_R^i\}$, its input alphabet is $\Sigma \times X$, and its acceptance condition is $(\pi_Q)^{-1}(\Omega)$, i.e. a sequence $(q_0, v_0)(q_1, v_1) \dots$ is accepting

whenever $q_0q_1 \cdots \in \Omega$. Finally, for all states $(p, v), (q, \lambda) \in Q_X$ and all labelled data $(\sigma, x) \in \Sigma \times X$, $(q, v) \xrightarrow[A_X]{(\sigma, x)} (q, \lambda)$ whenever it is the case in A (i.e. $(q, v) \xrightarrow[A]{(\sigma, x)} (q, \lambda)$). As stated above, the existence of a transition in A_X is decidable by Assumption 4.8. The fact that A_X accepts the expected ω -language follows from the definitions. $\square$

4.4.2. Projection over Labels

The proof techniques of Chapter 6 rely on projecting a data language over its labels. Formally, given a language of labelled data words $L \subseteq (\Sigma \times \mathbb{D})^\omega$, we define its *projection over labels* as $\text{lab}(L) = \{\text{lab}(w) \mid w \in L\}$.

Contrary to the *restriction* to a finite set of data values, the *projection* over labels of the data language recognised by a register automaton is not always ω -regular. Already over data domains with equality only, universal register automata do not have this property, as witnessed by Property 4.49; actually, it can be any recursively enumerable language (Theorem 4.50). Non-deterministic register automata are better behaved: their projection over labels is ω -regular over $(\mathbb{D}, =)$ (see Section 4.5 and Proposition 4.19), as well as over $(\mathbb{Q}, <)$ (cf Section 4.6 and Proposition 4.35). This again holds for *finite* data words over $(\mathbb{N}, <, 0)$, since then the projection over labels is the same over $(\mathbb{Q}_+, <, 0)$ and over $(\mathbb{N}, <, 0)$, by multiplying all data values by some common multiple of their denominators (see Lemma 12.44). However, this does not hold anymore for *infinite* data words over $(\mathbb{N}, <, 0)$, see Property 4.38. Projections over labels of such data languages exactly correspond to ω B-regular languages [39, Theorem 17], as defined in [59]. This makes such machines harder to abstract, and is at the source of the technical difficulties encountered in Section 7.3.4, as well as those of Section 12.3 in Part II.

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

[59]: Bojańczyk and Colcombet (2006), ‘Bounds in ω -Regularity’

4.4.3. Union and Intersection

The constructions for finite-state automata and ω -automata can be generalised to register automata to get closure under union and intersection.

Proposition 4.2 (Union and Intersection of NRA) *Let A_1, A_2 be two non-deterministic register automata over the same data domain $\mathbb{D}$, with respectively n_1 and n_2 states and k_1 and k_2 registers, and with acceptance condition Ω_1 and Ω_2 . Then:*

- $L(A_1) \cup L(A_2)$ is recognised by a non-deterministic register automaton with $n_1 + n_2$ states and $\max(k_1, k_2)$ registers, and with acceptance condition $\Omega_1 \cup \Omega_2$.
- $L(A_1) \cap L(A_2)$ is recognised by a non-deterministic register automaton with $n_1 \cdot n_2$ states and $k_1 + k_2$ registers, and with acceptance condition $\pi_1^{-1}(\Omega_1) \cap \pi_2^{-1}(\Omega_2)$.

Proof. The proof is adapted from [21, Theorem 3], which establishes the above properties in the case of $(\mathbb{D}, =, \mathbb{C})$. For $i = 1, 2$, let $A_i = (Q_i, I_i, \Sigma_i, R_i, \Delta_i, \Omega_i)$ be two non-deterministic register automata.

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

Union In our definition, register automata can have multiple initial states, so $L(A_1) \cup L(A_2)$ is recognised by the union of A_1 and A_2 (we can assume without loss of generality that their set of states are disjoint). A naive construction yields an automaton with $k_1 + k_2$ registers, but one can notice that the registers of A_1 can be used to simulate those of A_2 (or conversely if $|R_2| > |R_1|$).

Intersection To recognise the intersection, one generalises the product construction of ω -automata, which consists in simulating A_1 and A_2 in a synchronised way. Let $A_1 = (Q_1, I_1, \Sigma_1, R_1, \Delta_1, \Omega_1)$ and $A_2 = (Q_2, I_2, \Sigma_2, R_2, \Delta_2, \Omega_2)$. Assume without loss of generality that $R_1 \cap R_2 = \emptyset$, and let $\Sigma = \Sigma_1 \cup \Sigma_2$. We define their *product* as $A_1 \otimes A_2 = (Q_1 \times Q_2, I_1 \times I_2, \Sigma, R_1 \cup R_2, \Delta, \Omega)$, where $(p, q) \xrightarrow[A_1 \otimes A_2]{\sigma, \phi_1 \wedge \phi_2, \text{asgn}_1 \cup \text{asgn}_2} t_1 \otimes t_2$ (p', q') whenever $p \xrightarrow[A_1]{\sigma, \phi_1, \text{asgn}_1} p'$ and $q \xrightarrow[A_2]{\sigma, \phi_2, \text{asgn}_2} q'$. The acceptance condition is then $\Omega_{1,2} = \pi_1^{-1}(\Omega_1) \cap \pi_2^{-1}(\Omega_2) = \{(p_0, q_0)(p_1, q_1) \cdots \mid p_0 p_1 \cdots \in \Omega_1 \text{ and } q_0 q_1 \cdots \in \Omega_2\}$. It is routine to show that there exists a run $((p_0, v_0)(q_0, \lambda_0))(t_0^1 \otimes t_0^2)((p_1, v_1)(q_1, \lambda_1)) \dots$ of $A_1 \otimes A_2$ over $w \in (\Sigma \times \mathbb{D})^\omega$ if and only if $(p_0, v_0)t_0^1(p_1, v_1) \dots$ is a run of A_1 over w and $(q_0, \lambda_0)t_0^2(q_1, \lambda_1) \dots$ is a run of A_2 over w . $\square$

By duality, we obtain:

Proposition 4.3 (Union and Intersection of URA) *Let A_1, A_2 be two universal register automata over the same data domain $\mathcal{D}$, with respectively n_1 and n_2 states and k_1 and k_2 registers, and with acceptance condition Ω_1 and Ω_2 . Then:*

- $L(A_1) \cup L(A_2)$ is recognised by a universal register automaton with $n_1 \cdot n_2$ states and $k_1 + k_2$ registers, and with acceptance condition $\pi_1^{-1}(\Omega_1) \cap \pi_2^{-1}(\Omega_2)$.
- $L(A_1) \cap L(A_2)$ is recognised by a universal register automaton with $n_1 + n_2$ states and $\max(k_1, k_2)$ registers, and with acceptance condition $\Omega_1 \cup \Omega_2$.

Recall that for universal automata, the acceptance condition defines which runs are rejecting.

Finally, note that the product construction that we use for intersection of NRA preserves determinism. It can also be applied to recognise the union, so we get:

Proposition 4.4 (Union and Intersection of DRA) *Let A_1, A_2 be two deterministic register automata over the same data domain $\mathcal{D}$, with respectively n_1 and n_2 states and k_1 and k_2 registers, and with acceptance condition Ω_1 and Ω_2 . Then:*

- $L(A_1) \cup L(A_2)$ is recognised by a deterministic register automaton with $n_1 \cdot n_2$ states and $k_1 + k_2$ registers, and with acceptance condition $\pi_1^{-1}(\Omega_1) \cup \pi_2^{-1}(\Omega_2)$.
- $L(A_1) \cap L(A_2)$ is recognised by a deterministic register automaton with $n_1 \cdot n_2$ states and $k_1 + k_2$ registers, and with acceptance condition $\pi_1^{-1}(\Omega_1) \cap \pi_2^{-1}(\Omega_2)$.

Remark 4.9 In [21, Theorem 3], the authors demonstrate that register automata over finite data words in data domain $(\mathbb{D}, =)$ are closed under concatenation and Kleene star. The data domain actually plays no role, so this property again holds for any data domain. We do not provide a formal proof, as our study is focused on infinite data words.

Remark 4.10 Contrary to automata over finite alphabets, register automata are not closed under complement (cf Property 4.18). This missing property, shared with many models over infinite alphabets [103], is the source of many difficulties as illustrated by the undecidability of the universality problem for non-deterministic register automata [101, Theorem 18] (see also Theorem 4.21).

4.4.4. A Normal Form for Tests

In our definition, we assume that a test can be any quantifier-free formula over variables $R \sqcup \{\star\}$, to give the most flexibility for defining transitions. However, for computational purposes, it is often easier to manipulate elements from a *finite* set, and indeed, register automata are often introduced using a finite set of tests straight away [21–23].

It is well-known that any quantifier-free formula φ admits a unique *full disjunctive normal form* $\text{DNF}(\varphi)$, i.e. it is equivalent to a disjunction of maximally consistent conjunctions of literals, where a *literal* is either $P(\mathbf{t})$ or $\neg P(\mathbf{t})$, for a relation $P \in \mathfrak{R}$ of arity $k \in \mathbb{N}$ and $\mathbf{t} \in (X \sqcup \mathbb{C})^k$ (where X is the set of variables of φ) (Equation 4.1). Note that the size of $\text{DNF}(\varphi)$ is exponential in the number of relation and constant symbols as well as in the number of variables. Given a transition $t = p \xrightarrow{\sigma, \phi, \text{asgn}} q$, we write

$$\phi \equiv \bigvee_{i \in I} \kappa_i, \text{ where, for } i \in I, \kappa_i = \bigwedge_{\substack{P \in \mathfrak{R} \\ \text{of arity } k}} \bigwedge_{\substack{(x_1, \dots, x_k) \\ \in ((R \sqcup \{\star\}) \sqcup \mathbb{C})^k}} l_{P, x_1, \dots, x_k} \quad (4.1)$$

Thus, each κ_i is a maximally consistent conjunction of literals. Here, each $l_{P, x_1, \dots, x_k} \in \{P(x_1, \dots, x_k), \neg P(x_1, \dots, x_k)\}$, where k is the arity of P , and I is chosen so that each κ_i is satisfiable. Then, t can be replaced with the set of transitions $\left\{ p \xrightarrow{\sigma, \kappa_i, \text{asgn}} q \right\}_{i \in I}$. We denote $\text{MCTests}_{\mathfrak{D}}(R)$ the set of tests that consist in maximally consistent conjunctions of literals. Note that any such test is characterised by the choice of its literals $l_{P, x_1, \dots, x_k}$ for all $P \in \mathfrak{R}$ and all $x_1, \dots, x_k \in (R \sqcup \{\star\})^k$, where k is the arity of P . Thus, there are exponentially many (with regard to the number of registers, predicates and constants) in general. We thus have the following:

Proposition 4.5 *Any register automaton is equivalent with a register automaton whose tests are maximally consistent conjunctions of literals.*

More precisely, let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be a register automaton. There exists a register automaton $A' = (Q, I, \Sigma, R, \Delta', \Omega)$, where $\Delta' \subseteq Q \times \Sigma \times \text{MCTests}_{\mathfrak{D}}(R) \times 2^R \times Q$.

This at a price: the size of $\text{MCTests}_{\mathfrak{D}}(R)$ is $2^{\prod_{P \in \mathfrak{R}} (r+1+c)^{\text{ar}(P)}}$, where $r = |R|$, $c = |\mathbb{C}|$ and $\text{ar}(P)$ denotes the arity of P .

A register automaton with maximally consistent tests is sometimes said to be in *explicit form*.

[103]: Björklund and Schwentick (2010), ‘On notions of regularity for data languages’

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

[22]: Segoufin (2006), ‘Automata and Logics for Words and Trees over an Infinite Alphabet’

[23]: Benedikt, Ley, and Puppis (2010), ‘What You Must Remember When Processing Data Words’

Of course, uniqueness of the DNF is up to reordering of the conjunctions.

As for ω -automata, two register automata A and A' are said *equivalent* whenever $L(A) = L(A')$.

4.4.5. On Assignments

Reassignments

First, we can show that allowing register automata to reassign the content of their registers to other registers along a run does not increase expressiveness. This gives a bit of flexibility to derive similar constructions, and in particular the conversion from multi-assignment register automata to single-assignment ones in the next section.

Definition 4.15 (Register Automaton with Reassignments) A *register automaton with reassignments* is a register automaton that additionally conducts reassignments on its transitions. A *reassignment* is modelled as a function $f : R \rightarrow R$. Intuitively, $\text{reasn}(r) = s$ means that when the automaton conducts reassignment reasn , r takes the value of s . If reasn is the identity, we are back to the usual definition.

Formally, it is a tuple $A = (Q, I, \Sigma, R, \Delta, \Omega)$ where Q, I, Σ, R, Ω are defined as for a register automaton without reassignments. The type of Δ is enriched to $Q \times \Sigma \times \text{Tests}_{\mathcal{D}}(R) \times 2^R \times R^R \times Q$. We write $p \xrightarrow{\sigma, \phi, \text{asgn}, \text{reasn}} q$ when $(p, \sigma, \phi, \text{asgn}, \text{reasn}q) \in \Delta$.

We only define how a valuation is updated with regard to a reassignment and the corresponding successor relation for configurations; the notion of run and the semantics of the model are then defined as for register automata.

Given a valuation $v : R \rightarrow \mathbb{D}$, a data value $d \in \mathbb{D}$ and a reassignment function $\text{reasn} : R \rightarrow R$, the *update* of v following reasn is the valuation $\lambda = v \circ \text{reasn}$, i.e. for all $r \in R$, $\lambda(r) = v(\text{reasn}(r))$. Note that reassignments do not chain. For instance, if we have $\text{reasn}_q : r \mapsto s$ and $s \mapsto t$, it means that in the new valuation λ , $\lambda(r) = v(s)$ and $\lambda(s) = v(t)$.

If we have a transition $t = p \xrightarrow{\sigma, \phi, \text{asgn}, \text{reasn}} q$ then, for all configurations (p, v) and (q, λ) and all data values $d \in \mathbb{D}$, we have that $(p, v) \xrightarrow{\sigma, d, \phi, \text{asgn}, \text{reasn}}_t (q, \lambda)$ whenever:

- $v, d \models \phi$ (as in the usual definition)
- $\lambda = \mu \circ \text{reasn}_q$, where $\mu = v \setminus \{\text{asgn} \leftarrow d\}$ is obtained by assigning d to asgn .

Proposition 4.6 *Reassignments do not increase expressive power.*

More precisely, if $A = (Q, I, \Sigma, R, \Delta, \Omega)$ is a register automaton with reassignments, then there exists a register automaton (without reassignments) $A' = (Q \times R^R, I \times \{\text{id}_R\}, \Sigma, R, \Delta', \pi_Q^{-1}(\Omega))$ such that $L(A) = L(A')$.

Proof. The proof is similar to that of [21, Theorem 2], which establishes that one can convert an M-automaton to a finite-memory automaton. The idea is to store in the states a function that associates each register of A with the register of A' in which its value is stored. In the original construction, the authors need an additional register as their syntax disallows to read a locally fresh data value without storing it. It also simplifies the proof. However, $|R|$ registers are actually sufficient, as we demonstrate in the next section. But for now, let us focus on the removal of reassignments.

Recall that X^Y denotes the set of (total) functions from Y to X .

[21]: Kaminski and Francez (1994), 'Finite-Memory Automata'

An M-automaton is a register automaton that is allowed to assign the input data value to multiple registers, and to do so even if the data value is already present in some of its registers; this corresponds to our definition of register automaton. In contrast, finite-memory automata are only allowed to assign the input data value to a single register, and only when it is locally fresh. This ensures that the content of the registers is pairwise distinct along the run. This is detailed in the next section.

Let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be some register automaton with reassignments. For clarity, we let R' be a primed copy of R ; the result is obtained by removing the primes. We define $A' = (Q \times (R')^R, I \times \{\text{prime}_R\}, \Sigma, R', \Delta', \Omega')$, where $\text{prime}_R : r \in R \mapsto r'$ and $\Omega' = \pi_1^{-1}(\Omega)$, i.e. $(q_0, f_0)(q_1, f_1) \cdots \in \Omega'$ whenever $q_0 q_1 \cdots \in \Omega$. Then, Δ' is constructed as follows: assume there is a transition $p \xrightarrow[A]{\sigma, \phi, \text{asgn}, \text{reasgn}}_t q$. Let $f : R \rightarrow R'$. There are two cases:

- f is bijective, i.e. it is a permutation of R . Then, we have that $(p, f) \xrightarrow[A']{\sigma, f(\phi), f(\text{asgn})}_{t'} (q, f \circ \text{reasgn})$, where $f(\phi)$ consists in ϕ where each $r \in R$ is replaced with $f(r)$. This case does not appear in the proof of [21, Theorem 2], since when there is an additional register in R' , we cannot have that f is bijective.
- f is not bijective. Thus, it is not surjective, since $|R| = |R'|$. Pick some $r_0 \notin f(R)$ in a canonical way (e.g. by ordering the registers and taking the minimal one). Then, we let $(p, f) \xrightarrow[A']{\sigma, f(\phi), \{r_0\}}_{t'} (q, f')$, where, by letting $f'' : \begin{cases} r \in \text{asgn} \mapsto r_0 \\ r \notin \text{asgn} \mapsto f(r) \end{cases}$, we have $f' = f'' \circ \text{reasgn}$.

We provide a formal proof of the correctness of the construction, that essentially consists in pushing symbols. Let $w \in (\Sigma \times \mathbb{D})^\omega$, and let $\rho = (p_0, v_0) \cdot t_0 \cdot (p_1, v_1) \cdot t_1 \dots$ be a run of A over w . Let us show by induction on $i \in \mathbb{N}$ that $((p_0, \text{id}_R), v_0) \cdot t'_0 \cdot ((p_1, f_1), v'_1) \cdot t'_1 \dots$ is a run of A' over w , where for all $i \in \mathbb{N}$ the t'_i are obtained from the t_i by the above construction and the v'_i are such that $v'_i \circ f_i = v_i$.

The initial step is trivial. Now, assume the induction hypothesis holds at step $i \geq 0$. A' is in configuration $((q_i, f_i), v'_i)$, where $v'_i \circ f_i = v_i$. It reads $(\sigma_i, d_i) = w[i]$. We know that A takes transition $t_i = q_i \xrightarrow[A]{\sigma_i, \phi_i, \text{asgn}_i, \text{reasgn}_i} q_{i+1}$. Valuations evolve as follows: first, d_i is assigned to asgn_i , yielding $\mu_i = v_i \{\text{asgn}_i \leftarrow d_i\}$. Then, A reassigns according to reasgn_i , which yields $v_{i+1} = \mu_i \circ \text{reasgn}_i$. Overall, $(q_i, v_i) \xrightarrow[A]{\sigma_i, d_i}_{t_i} (q_{i+1}, v_{i+1})$. Since t_i is enabled on reading d_i , we have that $v_i, d_i \models \phi_i$. Thus, $v_i \circ f_i \models f_i(\phi_i)$, which means that t'_i is enabled from $((q_i, f_i), v'_i)$ on reading (σ_i, d_i) . There are two cases:

- f_i is surjective (hence bijective). Then, in A' , there is a transition $((q_i, f_i), v'_i) \xrightarrow[A']{\sigma_i, d_i, f_i(\phi_i), f_i(\text{asgn}_i)}_{t'_i} ((q_{i+1}, f_{i+1}), v'_{i+1})$, where $f_{i+1} = f_i \circ \text{reasgn}_i$ and $v'_{i+1} = v'_i \{f_i(\text{asgn}_i) \leftarrow d_i\}$. Since $v'_i \circ f_i = v_i$, we get that $\mu_i = v'_{i+1} \circ f_i$. Besides, we know that $v_{i+1} = \mu_i \circ \text{reasgn}_i$. Thus, $v_{i+1} = \mu_i \circ \text{reasgn}_i = v'_{i+1} \circ f_i \circ \text{reasgn}_i = v'_{i+1} \circ f_{i+1}$, which means that the inductive invariant holds at step $i + 1$.
- f_i is not surjective. Then, on reading (σ_i, d_i) , A' can take the transition $((q_i, f_i), v'_i) \xrightarrow[A']{\sigma_i, d_i, f_i(\phi_i), \{r_i\}}_{t'_i} ((q_{i+1}, f_{i+1}), v'_{i+1})$ for some canonically chosen $r_i \notin f_i(R)$, where f_{i+1} is defined as $f''_i \circ \text{reasgn}_i$, for $f''_i : \begin{cases} r \in \text{asgn}_i \mapsto r_i \\ r \notin \text{asgn}_i \mapsto f_i(r) \end{cases}$. Then, we have $v'_{i+1} = v'_i \{r_i \leftarrow d_i\}$. Thus, $v'_{i+1} \circ f_{i+1} = \mu_i$. As a consequence, $v'_{i+1} \circ f_{i+1} = v_{i+1}$.

Recall that μ_i is the intermediate valuation (after d_i has been assigned, but before reassignment reasgn_i), i.e. $\mu_i = v_i \{\text{asgn}_i \leftarrow d_i\}$.

In both cases, from a run of A , we are able to construct a run of A' . Similarly, it can be show that any initial run of A' corresponds to a run of A . Since the acceptance condition is obtained by projecting away the additional information contained in the f_i , we obtain that for all $w \in (\Sigma \times \mathbb{D})^\omega$, there is an accepting run of A over w if and only if there is an accepting run of A' of w . $\square$

Single-Assignment Register Automata

In our definition, registers are allowed to take duplicate values. In their seminal paper, Kaminski and Francez instead enforced that along any run, register values are always pairwise distinct. This is done by only assigning a single register, and conducting an assignment only when the input data value is *locally fresh*, i.e. it is not already present in the registers.

Definition 4.16 Formally, a *single-assignment register automaton* is a register automaton such that for all transitions $p \xrightarrow{\sigma, \phi, \text{asgn}} q$, if $\phi \wedge (\star = r)$ is satisfiable for some $r \in R$, then $\text{asgn} = \emptyset$, and otherwise $\text{asgn} = \{r\}$ or $\text{asgn} = \emptyset$.

Note that this property also makes sense for register automata with reassignments, and we extend it in the expected way.

It is routine to show by induction on the length of a partial run that along any run of a single-assignment register automaton *without* reassignments, the contents of registers which do not contain $\#$ are all pairwise distinct. Formally:

Proposition 4.7 Let A be a single-assignment register automaton, and let ρ be a run of A , whose valuations are $\text{valuations}(\rho) = v_0 v_1 \dots$. Then, for all $i \in \mathbb{N}$, we have that for all $r \neq s \in R$, if $v_i(r) = v_i(s)$ then $v_i(r) = v_i(s) = \#$.

Notation 4.17 (Locally fresh) In the following, data values that are *locally fresh* play a special role. Thus, given a set R of registers, we denote $\text{locFresh}_R := \bigwedge_{r \in R} \star \neq r$ a formula that is satisfied by a data value d if and only if it is locally fresh with regards to the current valuation.

In their paper, Kaminski and Francez show that the single-assignment model is equivalent to the multiple-assignment model that we defined, which they call *M-automaton*. We can actually generalise this property to register automata over any data domain. Besides, our construction does not require an additional register.

Proposition 4.8 ([21, Theorem 2]) Any register automaton is equivalent with a single-assignment register automaton.

More precisely, let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be a register automaton. There exists a single-assignment register automaton A' with set of states $Q \times R^R$ such that $L(A) = L(A')$.

Proof. We leverage Proposition 4.6, and show how to convert a register automaton to a single-assignment one, which conducts reassignments. Observe that the reassignment removal procedure (proof of Proposition 4.6) preserves the syntactical single-assignment property. The general idea of the construction is the same as that of [21, Theorem 2], with two

Note that Proposition 4.7 does not hold for register automata with reassignments.

When we write ‘register automaton’, we implicitly mean that it does not conduct reassignments. They are only a technical tool that we use to ease the constructions.

differences. First, as pointed out earlier, we are able to keep the same number of registers, instead of adding one. Second, in our definition, register automata operate with tests given as logical formulas, so each transition has to be replaced with $|R| + 1$ transitions, depending on whether $\star = r$ for some $r \in R$, or whether it is locally fresh. For instance, the test $\top$ is consistent with all possibilities, but assignment is allowed only if the data value is locally fresh.

Let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be a register automaton. We define a register automaton with reassignments $A' = (Q \times R^R, I, \Sigma, R, \Delta', \Omega)$. Assume there is a transition $p \xrightarrow[A]{\sigma, \phi, \text{asgn}}_t q$ in A . There are two cases:

- If $\text{asgn} = \emptyset$, then we let $p \xrightarrow[A']{\sigma, \phi, \emptyset, \text{id}_R}_{t'} q$, where id_R is the identity reassignment, i.e. no reassignment is conducted
- Otherwise, we add the following transitions, where the register automaton acts differently depending on whether the input data value is locally fresh or not:
 - (the data value is contained in some register) For all $r \in R$, we let $p \xrightarrow[A']{\sigma, \phi \wedge (\star = r), \emptyset, \text{asgn} := r} q$, where $\text{asgn} := r$ denotes the reassignment function $f : \begin{cases} s \in \text{asgn} \mapsto r \\ s \notin \text{asgn} \mapsto s \end{cases}$
 - (the data value is locally fresh) $p \xrightarrow[A']{\sigma, \phi \wedge \text{locFresh}_R(\star), \{r_0\}, \text{asgn} := r_0}_{t_t} q$ for some canonically chosen $r_0 \in \text{asgn}$.

Then, by construction, for all $(p, \nu), (q, \lambda) \in \text{Configs}(A)$, we have:

- for all $(\sigma, d) \in \Sigma \times \mathbb{D}$ such that $d = \nu(r)$ for some $r \in R$, $(p, \nu) \xrightarrow[A]{\sigma, d, \phi, \text{asgn}}_t (q, \lambda)$ if and only if $(p, \nu) \xrightarrow[A']{\sigma, d, \phi \wedge (\star = r), \emptyset, \text{asgn} := r} (q, \lambda)$
- for all $(\sigma, d) \in \Sigma \times \mathbb{D}$ such that $d \notin \nu(R)$, we have that $(p, \nu) \xrightarrow[A]{\sigma, d, \phi, \text{asgn}}_t (q, \lambda)$ if and only if $(p, \nu) \xrightarrow[A']{\sigma, d, \phi \wedge \text{locFresh}_R(\star), \{r_0\}, \text{asgn} := r_0}_{t_t} (q, \lambda)$ (for the canonically chosen r_0)

As a consequence, A' recognises the same data language as A . By Proposition 4.6, removing the reassignments yields an equivalent single-assignment register automaton with states $Q \times R^R$. $\square$

This property proves quite useful, as it allows us to target single-assignment register transducers as implementations in our study of the Church synthesis problem over data words (see Chapter 5).

4.4.6. Closure under Automorphisms

Now, another important property of register automata is that the data languages they recognise are closed under automorphisms.

Definition 4.18 (Automorphism of a data domain) Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathbb{C})$ be a data domain. A function $\mu : \mathbb{D} \rightarrow \mathbb{D}$ is an *automorphism* of $\mathcal{D}$ if it is bijective and preserves relations and constants, i.e.:

Recall that since we are working with ‘implicit’ tests, a single transition might accept data values that are locally fresh and ones that are not.

When we say that r_0 is canonically chosen, we mean that we do not add one transition for each possible r , but instead pick e.g. $r_0 = \min \text{asgn}$ and add this specific transition.

In the terminology of nominal sets, closure under automorphisms is called equivariance. For now, we avoid this terminology so as not to scare the reader that are not mathematically inclined. This notion is however detailed in Part II, which has a more abstract flavour.

For simplicity, we confuse a symbol of the signature with its interpretation.

- for all constants $c \in \mathbb{C}$, $\mu(c) = c$
- for all relation $P \in \mathbb{R}$ of arity k and for all tuples $(d_1, \dots, d_k) \in \mathbb{D}^k$, we have $(d_1, \dots, d_k) \in P$ if and only if $(\mu(d_1), \dots, \mu(d_k)) \in P$.

The set of automorphisms of $\mathbb{D}$ is denoted $\text{Aut}(\mathbb{D})$. Note that it has a group structure for the composition operation.

Automorphisms are extended to labelled data in the obvious way: $\mu(\sigma, d) = (\sigma, \mu(d))$ for all $(\sigma, d) \in \Sigma \times \mathbb{D}$.

We now show the following lemma, that allows to conclude that languages recognised by register automata are closed under automorphisms (Proposition 4.10):

Lemma 4.9 *Let $\mathbb{D}$ be a data domain, R a set of registers, and let $v, v' : R \rightarrow \mathbb{D}$ be two valuations over R . Finally, let $\mu \in \text{Aut}(\mathbb{D})$. Then, for all data $d \in \mathbb{D}$ and for all actions $(\phi, \text{asgn}) \in \text{Actions}_{\mathbb{D}}(R)$, we have:*

$$v \xrightarrow[\mathcal{T}_R^{\mathbb{D}}]{d, (\phi, \text{asgn})} v' \text{ if and only if } \mu(v) \xrightarrow[\mathcal{T}_R^{\mathbb{D}}]{\mu(d), (\phi, \text{asgn})} \mu(v')$$

Proof. The key point of the proof is the following: since μ preserves both the relations and constants of $\mathbb{D}$, it preserves satisfiability. More precisely, for any finite set X and any valuation $\tau : X \rightarrow \mathbb{D}$, and for any atomic formula ϕ , we have $\tau \models \phi$ if and only if $\mu(\tau) \models \phi$ (the cases of $\top$ and $\perp$ are trivial, and the case of relations $P(\mathbf{t})$ is by definition). This can be extended by induction to formulas, and hence to tests: for any valuation $v : R \sqcup \{\star\} \rightarrow \mathbb{D}$, for any test ϕ over $\mathbb{D}$, $v \models \phi$ iff $\mu(v) \models \phi$.

Thus, let $\mathbb{D}$ be a data domain, R be a set of registers, $v, v' : R \rightarrow \mathbb{D}$ and $\mu \in \text{Aut}(\mathbb{D})$. Moreover, let $d \in \mathbb{D}$ and $(\phi, \text{asgn}) \in \text{Actions}_{\mathbb{D}}(R)$. Assume that $v \xrightarrow[\mathcal{T}_R^{\mathbb{D}}]{d, (\phi, \text{asgn})} v'$. This means that $v\{\star \leftarrow d\} \models \phi$ and $v' = v\{\text{asgn} \leftarrow d\}$. Then, $\mu(v)\{\star \leftarrow \mu(d)\} = \mu(v\{\star \leftarrow d\})$. By the above reasoning, this implies that $\mu(v\{\star \leftarrow d\}) \models \phi$. Moreover, observe that $\mu(v)\{\text{asgn} \leftarrow \mu(d)\} = \mu(v\{\text{asgn} \leftarrow d\})$. We obtain that $\mu(v) \xrightarrow[\mathcal{T}_R^{\mathbb{D}}]{\mu(d), (\phi, \text{asgn})} \mu(v')$.

The converse direction is obtained by applying the above reasoning to the inverse μ^{-1} of μ , which also belongs to $\text{Aut}(\mathbb{D})$. $\square$

We are now ready to prove the following key property:

Proposition 4.10 (Closure under automorphisms) *Let A be a register automaton over data domain $\mathbb{D}$ recognising a data language $L \subseteq (\Sigma \times \mathbb{D})^\omega$, and let $\mu \in \text{Aut}(\mathbb{D})$. Then, $\mu(L) = L$.*

In other words, for all labelled data words $w \in (\Sigma \times \mathbb{D})^\omega$, $w \in L \Leftrightarrow \mu(w) \in L$.

Remark 4.11 Note that we make no hypothesis on the semantics of A : it can be non-deterministic or universal.

Proof. Let A be a register automaton. Since labels are left unchanged by automorphisms, they play no role in the argument. To avoid cluttering the notations, we treat the case of a unary label alphabet and omit it. Let $w \in \mathbb{D}^\omega$. We show that for all runs ρ of A over w , $\mu(\rho)$ is a run of A over $\mu(w)$, where we apply μ to valuations of ρ pointwise.

Composition of functions is associative, and it is routine to check that the composition of two automorphisms again preserves relations and constants. The neutral element of $\text{Aut}(\mathbb{D})$ is the identity function $\text{id}_{\mathbb{D}}$, and the inverse of an automorphism is its inverse function.

Recall that $\mathcal{T}_R^{\mathbb{D}}$ is the transition system of Definition 4.12. It defines the semantics of register automata.

$\mu(v)$ is to be understood as function composition $\mu \circ v$, i.e. for all $r \in R$, we define $\mu(v)(r) = \mu(v(r))$.

Let $\rho = (q_0, v_0)t_0(q_1, v_1)t_1 \dots$ be a run of A over w , if it exists. Let $\mu(\rho) = (q_0, \mu(v_0))t_0(q_1, \mu(v_1))t_1 \dots$, where for all valuations $v : R \rightarrow \mathbb{D}$, $\mu(v) : r \mapsto \mu(v(r))$. By the above Lemma 4.9, we have that for all i , $(q_i, \mu(v_i)) \xrightarrow{\mu(d_i)}_{t_i} (q_{i+1}, \mu(v_{i+1}))$, so we get that $\mu(\rho)$ is indeed a run of A over $\mu(w)$.

Moreover, note that $\mu(v_R^l) = v_R^l$, so $\mu(\rho)$ is initial whenever ρ is initial. Since μ does not affect the states, we get that if w is accepted by A , then so is $\mu(w)$. The converse is obtained by applying the above reasoning to μ^{-1} (or, equivalently, by using the other direction of the above lemma). Finally, note that since our analysis establishes a one-to-one correspondence between initial runs over w and $\mu(w)$, the argument holds for both the non-deterministic and universal semantics. $\square$

4.5. Non-Deterministic Register Automata over Labelled Data Words with Equality

Register automata were first introduced as *finite-memory automaton* [21] over data domains with the equality predicate only, with the non-deterministic semantics and over finite data words. We start with this data domain, as it is the most elementary one; it is also the most well-behaved. The results that we present here are a mild generalisation of those of [21], in that we consider *labelled* data values and allow the use of constants. This proves useful in many of our reductions, on top of slightly enriching the modelling power. We already pointed out earlier that finite-memory automaton correspond with *single-assignment* register automata, but, as they show, this is not a restriction [21, Theorem 2] (see also Proposition 4.8). Finally, in their setting, the initial value $\#$ of the registers is not part of the input alphabet. We chose to include it since it simplifies some reasonings, but it can be simulated by considering tests of the form $\phi \wedge (\star \neq \#)$ (Remark 4.5).

In the rest of this section, we fix a finite label alphabet Σ . The data domain is $\mathcal{D} := (\mathbb{D}, =, \mathbb{C})$, where $\mathbb{D}$ is a countably infinite set, $=$ is interpreted as the equality predicate and $\mathbb{C} \subset_f \mathbb{D}$ is a finite non-empty set of constants (for simplicity, we confuse the constant symbols and their interpretation, as no ambiguity arises). Among them, we arbitrarily distinguish some $\# \in \mathbb{C}$ that consists in the initial content of the registers. We denote $\gamma = |\mathbb{C}|$ the number of constants. This setting is isomorphic to $(\mathbb{N}, =, \{0, \dots, \gamma - 1\})$, and one can take e.g. $\# = 0$.

4.5.1. Constraints

Over $(\mathbb{D}, =, \mathbb{C})$, atomic formulas over variables X are of the form $x = y$ for $x, y \in X \sqcup \mathbb{C}$. Thus, given a valuation $v : X \rightarrow \mathbb{D}$, the information whether $v(x) = v(y)$, along with if $v(x) = c$ for some constant $c \in \mathbb{C}$ allows to decide whether a given formula is satisfied by a valuation. In the field of register automata, this information is stored through constraints, that allow to abstract their behaviour with regards to tests. This notion is pervasive, and is used for instance in [28] through *abstract states* to

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

We use $:=$ to avoid a confusion with $=$.

We prefer to write the study for an arbitrary $\mathbb{D}$ to highlight the fact that its structure does not matter. This also helps preventing a mixup with Section 12.3.1, which is dedicated to register automata over $(\mathbb{N}, <, 0)$.

decide emptiness of register automata and in [36] when defining *boolean associates*.

We first introduce constraints in the case of equality, before generalising it to other data domains through its relation with the notion of type (Section 4.5.5). For now, we prefer to keep things elementary. They allow to establish that any register automaton over $(\mathbb{D}, =, \mathbb{C})$ can be turned into a locally concretisable one (Proposition 4.25), and to recognise feasible action sequences (Section 4.5.4). We also use constraints in the proof of the renaming property as a tool to represent information about valuations.

Notation 4.19 It is often useful to extend valuations to constants as follows: for $v : R \rightarrow \mathbb{D}$, we let $v^{\mathbb{C}}(r) = v(r)$ for all $r \in R$, and $v^{\mathbb{C}}(c) = c$ for all $c \in \mathbb{C}$. When it is clear from the context, we again denote v this extended valuation.

Definition 4.20 (Constraint associated with a valuation) Let $v : R \rightarrow \mathbb{D}$ be a valuation over R . Its associated *constraint* is the equivalence relation over $R \sqcup \mathbb{C}$ such that for all $x, y \in R \sqcup \mathbb{C}$, $x \sim_v y$ whenever $v^{\mathbb{C}}(x) = v^{\mathbb{C}}(y)$. We denote $[v] = (R \sqcup \mathbb{C}) / \sim_v$. To a constraint τ , we associate the *constraint formula* $\psi_\tau := \bigwedge_{x,y} x \bowtie_{x,y} y$, where $\bowtie_{x,y} \in \{=, \neq\}$ is $=$ whenever $x \sim y$ in τ . Note that $v \models \psi_\tau$ if and only if $[v] = \tau$, so a constraint is equivalent to a formula ψ_τ . It is represented as an equivalence relation to distinguish it from tests.

Extending valuations to constants allows for a uniform treatment of equalities $v(r) = v(r')$ for $r, r' \in R$ and $v(r) = c$ for $c \in \mathbb{C}$.

Note that a constraint stores in particular the information whether r contains $\#$ or not.

Example 4.4 Assume that $\mathbb{C} = \{\#, c\}$. Consider the valuation $v : R = \{r_0, r_1, r_2, r_3\} \rightarrow \mathbb{D}$ defined by letting $v(r_0) = \#, v(r_1) = a, v(r_2) = b$ and $v(r_3) = b$, where $a, b \in \mathbb{D} \setminus \mathbb{C}$ are some arbitrary data values such that $a \neq b$. The constraint associated with v is the equivalence relation over $R \sqcup \mathbb{C}$ whose equivalence classes are $\{\{\#, r_0\}, \{r_1\}, \{r_2, r_3\}, \{c\}\}$.

Let $v' : R \rightarrow \mathbb{D}$ where a and b are swapped, i.e. defined by letting $v'(r_0) = \#, v'(r_1) = b, v'(r_2) = a$ and $v'(r_3) = a$. The constraint associated with v' is the same as that of v , i.e. $[v'] = [v]$.

Over $(\mathbb{D}, =, \mathbb{C})$, constraints allow to abstract the behaviour of register automata. We establish a few properties to that effect. First, a constraint determines which formulas can be satisfied.

Lemma 4.11 Let X be a finite set of variables, and let $v, v' : X \rightarrow \mathbb{D}$ be two valuations over R such that $[v] = [v']$. For all quantifier-free formulas $\varphi \in \text{QF}(X)$, $v \models \varphi$ if and only if $v' \models \varphi$.

Proof. It suffices to observe that for all atomic formulas, we have that $v \models \psi \Leftrightarrow v' \models \psi$. Indeed, they are of the form $\psi := x = y$, for $x, y \in X \sqcup \mathbb{C}$. Since $[v] = [v']$, by definition we have $v(x) = v(y)$ whenever $v'(x) = v'(y)$. The result then follows by structural induction on the form of φ . $\square$

A consequence of Lemma 4.11 is that a constraint determines which tests can be satisfied from a valuation.

Lemma 4.12 *Let R be a finite set of registers, and let $v, v' : R \rightarrow \mathbb{D}$ be two valuations over R such that $[v] = [v']$. For all tests $\phi \in \text{Tests}(R)$, there exists $d \in \mathbb{D}$ such that $v, d \models \phi$ if and only if there exists $d' \in \mathbb{D}$ such that $v', d' \models \phi$.*

Proof. Let R be a finite set of registers, and $v, v' : R \rightarrow \mathbb{D}$ be two valuations over R such that $[v] = [v']$. Let $\phi \in \text{Tests}(R)$, and assume that there exists $d \in \mathbb{D}$ such that $v, d \models \phi$. Define $E = \{x \in R \sqcup \mathbb{C} \mid v(x) = d\}$. There are two cases:

- $E = \emptyset$. It means that d is locally fresh, and distinct from all constants. Pick $d' \in \mathbb{D} \setminus (v'(R) \cup \mathbb{C})$.
- $E \neq \emptyset$. E is an equivalence class of $[v]$, so, since $[v] = [v']$, we get that $v'(E)$ is a singleton. Pick d' the only element of $v'(E)$.

In both cases, we obtain that $[v\{\star \leftarrow d\}] = [v'\{\star \leftarrow d'\}]$: all equivalence classes are left unchanged, except for E , which is appended $\star$ in both constraints. By Lemma 4.12, we obtain that for all $\psi \in \text{QF}(R \sqcup \{\star\})$, $v\{\star \leftarrow d\} \models \psi$ if and only if $v'\{\star \leftarrow d'\} \models \psi$. This is the case in particular for ϕ , so we have that $v', d' \models \phi$.

The converse implication is obtained by interverting v and v' , as they play the same role. $\square$

As a consequence, a constraint contains enough information to determine whether a given transition is enabled from a configuration:

Proposition 4.13 *Let A be a register automaton with registers R , and $v, v' : R \rightarrow \mathbb{D}$ be two valuations over R such that $[v] = [v']$. For all transitions $t = p \xrightarrow{\sigma, \phi, \text{asgn}} q$, t is enabled from (p, v) if and only if it is enabled from (p, v') .*

Further, on taking a transition, one can end in a configuration that has the same constraint. This last result implies that constraints allow to abstract the behaviour of a register automaton, as only the constraint matter to decide the existence of a transition.

Proposition 4.14 *Let A be a register automaton with registers R , and $v, v' : R \rightarrow \mathbb{D}$ be two valuations over R such that $[v] = [v']$. Finally, let $t = p \xrightarrow{\sigma, \phi, \text{asgn}} q$ be some transition of A , and assume that $(p, v) \xrightarrow{\sigma, d} (q, \lambda)$ for some $d \in \mathbb{D}$ and some $\lambda \in \text{Val}(R)$.*

Then, there exists $d' \in \mathbb{D}$ such that $[v\{\star \leftarrow d\}] = [v'\{\star \leftarrow d'\}]$, and for all such d' , we have that $(p, v') \xrightarrow{\sigma, d'} (q, \lambda')$, where $\lambda' = v'\{\text{asgn} \leftarrow d'\}$, and $[\lambda] = [\lambda']$ holds.

Proof. From the proof of Lemma 4.12, we know that there exists $d' \in \mathbb{D}$ such that $[v\{\star \leftarrow d\}] = [v'\{\star \leftarrow d'\}]$. Moreover, if d' satisfies this property, we have by Lemma 4.11 that $v', d' \models \phi$, so $(p, v') \xrightarrow{\sigma, d'} (q, \lambda')$, where $\lambda' = v'\{\text{asgn} \leftarrow d'\}$. Then, we have that $[v\{\text{asgn} \leftarrow d\}] = [v'\{\text{asgn} \leftarrow d'\}]$, i.e. $[\lambda] = [\lambda']$. $\square$

Remark 4.12 One can recover the renaming property from the above result as follows: first, recall that $(\mathbb{D}, =, \mathbb{C})$ is isomorphic to $(\mathbb{N}, =, \{0, \dots, c-1\})$. Then, at every step, pick the minimal $d' \in \mathbb{N}$ such that $[v\{\star \leftarrow d\}] =$

Recall that v is extended to constants by letting $v(c) = c$ for all constants $c \in \mathbb{C}$ (Notation 4.19).

A transition t is enabled for configuration (p, v) if there exists $d \in \mathbb{D}$ and (q, v') such that $(p, v) \xrightarrow{d}_t (q, v')$.

$[v' \{ \star \leftarrow d' \}]$. It is easy to show by induction that along the run, valuations take their values in $\{0, \dots, c + k\}$.

In Section 4.5.3, we show that storing constraints in the states allows to statically decide whether a transition is enabled from a state, without knowing the valuation. But for now, we move to a core property of the model that proves sufficient to establish decidability of the register-bounded synthesis problem for universal register automata over $(\mathbb{D}, =, \mathbb{C})$ (Chapter 6, and in particular Section 6.2.1).

4.5.2. The Renaming Property

A key property of non-deterministic register automata with equality only is the *renaming property* [21, Proposition 4]. It states that if an automaton with k registers recognises a data word w , then the data of such word can be renamed so that they only take $k + \gamma + 1$ distinct data values, where γ is the number of constants. Indeed, since it only has k registers, it can only distinguish the values that it has stored and the constants, but all data values that is neither of those look the same to the automaton. And, once the content of some register has been erased, it is indistinguishable from a value that has never been stored.

Correspondingly, a renaming set denotes any set that contain constants, along with sufficiently many data values so that one can rename a data word using those values while preserving acceptance.

Definition 4.21 (Renaming set) Let $k \geq 1$. A k -renaming set, or simply renaming set when k is clear from the context, is a set of the form $X = Y \sqcup \mathbb{C}$, where $Y \subseteq \mathbb{D} \setminus \mathbb{C}$ is of size $|Y| > k$.

Proposition 4.15 (Renaming property [21, Proposition 4]) Let A be a non-deterministic register automaton with k registers. Assume that A accepts some data word $w \in (\Sigma \times \mathbb{D})^\omega$. For all k -renaming sets $X \subseteq \mathbb{D}$, there exists a data word $w' \in (\Sigma \times X)^\omega$ such that $\text{lab}(w) = \text{lab}(w')$ and w' is accepted by A .

Proof. Let A be a non-deterministic automaton with k registers that accepts some data word $w \in (\Sigma \times \mathbb{D})^\omega$ through a run ρ . Let $X \subseteq \mathbb{D}$ be a k -renaming set, i.e. such that $X = Y \sqcup \mathbb{C}$ with $|Y| > k$.

We build by induction on i a data word $w' \in (\Sigma \times X)^\omega$ along with a run ρ' such that ρ' is a run of A on w' , $\text{trans}(\rho) = \text{trans}(\rho')$ and for all $i \in \mathbb{N}$, $[v_i] = [v'_i]$.

First, let $v'_0 = v_0 = v_R^t$.

Now, suppose that $\rho'[:i]$ and $w'[:i-1]$ have been constructed. We write $w[i] = (\sigma_i, d_i)$. By the induction hypothesis, we have that $v'_i : R \rightarrow X$ is such that $[v'_i] = [v_i]$. Let $t_i = q_i \xrightarrow{a_i, \phi_i, \text{asgn}_i} q_{i+1}$. Let $E_i = \{x \in R \sqcup \mathbb{C} \mid v_i(x) = d_i\}$ be the set of registers in v_i and of constants that are equal to d_i . There are two cases:

- $E_i \neq \emptyset$. It means that the input data value is equal to some register or constant. Then, E_i is an equivalence class of $[v_i] = [v'_i]$, so $v'_i(E_i)$ is a singleton and we let d'_i be its single element.

[21]: Kaminski and Francez (1994), 'Finite-Memory Automata'

Note that there is no γ in [21, Proposition 3], as there is only one constant $\#$ and it does not belong to the input alphabet.

Handling constants complicates the matter a bit as they break some symmetries and imply the introduction of a notion of renaming set, which is trivial when there are none. We treat them as they will be needed later on, and add an ounce of generality to the study.

The renaming property does not hold for universal register automata over $(\mathbb{D}, =)$, see Section 4.8, nor for register automata over $(\mathbb{Q}, <, 0)$, see Property 4.33.

In particular, $\text{states}(\rho) = \text{states}(\rho')$.

We write $\text{valuations}(\rho) = v_0 v_1 \dots$ and $\text{valuations}(\rho') = v'_0 v'_1 \dots$.

We write $\text{trans}(\rho) = t_0 t_1 \dots$.

Recall that valuations are extended to constants by letting $v(c) = c$ for all $c \in \mathbb{C}$.

Note that $E_i \sqcup \{\star\}$ is an equivalence class of $[v_i \{ \star \leftarrow d \}]$.

- $E_i = \emptyset$ (d_i is not in v_i). Then, let $d'_i \in X \setminus (v'_i(R) \cup \mathfrak{C})$ be a value distinct from all the registers of $v'_i(R)$, and from all constants. This value exists as $|X \setminus \mathfrak{C}| > k$, which implies that $|X \setminus (v'_i(R) \cup \mathfrak{C})| > 0$.

In both cases, we get that $[v_i\{\star \leftarrow d_i\}] = [v'_i\{\star \leftarrow d'_i\}]$. This entails that for each atomic formula $\psi := x = y$ for $x, y \in R \sqcup \mathfrak{C}$, $v_i\{\star \leftarrow d_i\} \models \varphi$ if and only if $v'_i\{\star \leftarrow d'_i\} \models \varphi$. A straightforward structural induction on ϕ_i yields that $v'_i, d'_i \models \phi_i$, since $v_i, d_i \models \phi_i$ (see Lemma 4.11).

Let $w'[i] = (\sigma_i, d'_i)$, and extend ρ with $\text{rho}[i+1] = t_i v'_{i+1}$, where $v'_{i+1} = v'_i\{\text{asgn}_i \leftarrow d'_i\}$. We then have that $(q_i, v'_i) \xrightarrow{\sigma_i, d'_i}_{t_i} (q_{i+1}, v'_{i+1})$, where $v'_{i+1} : R \rightarrow X$ satisfies $[v'_{i+1}] = [v_{i+1}]$ (Proposition 4.14).

Overall, ρ' is a run of A over w' , and $\text{states}(\rho') = \text{states}(\rho)$. Since ρ is accepting, ρ' is accepting as well, so w' is accepted by A , which concludes the proof. $\square$

This property has two immediate consequences:

Corollary 4.16 *Let A be a non-deterministic register automaton over $(\mathbb{D}, =, \mathfrak{C})$ with k registers such that $L(A) \neq \emptyset$.*

For any k -renaming set $X \subseteq \mathbb{D}$, $L(A) \cap (\Sigma \times X)^\omega \neq \emptyset$.

Corollary 4.17 *Let A be a non-deterministic automaton with k registers. For all k -renaming sets $X \subseteq \mathbb{D}$,*

$$\text{lab}(L(A)) = \text{lab}(L(A) \cap (\Sigma \times X)^\omega)$$

In particular, this implies that non-deterministic automata with the equality predicate cannot recognise the language of data words that contain pairwise distinct data values.

Property 4.18 ([21, Proposition 5]) *The data language $L_{\neq} = \{d_0 d_1 \dots \mid \forall i \neq j, d_i \neq d_j\}$ of Example 4.1 cannot be recognised by any non-deterministic register automaton over $(\mathbb{D}, =, \mathfrak{C})$.*

As it is recognised by the universal register automata over $(\mathbb{D}, =, \#)$ of Figure 4.1b on page 81, languages recognised by non-deterministic register automata over $(\mathbb{D}, =, \#)$ are not closed under complement.

[21]: Kaminski and Francez (1994), 'Finite-Memory Automata'

Projection over Labels

Since non-deterministic register automata behave like ω -automata over finite subsets of $\mathbb{D}$ (Proposition 4.1), we get:

Proposition 4.19 *Let A be a non-deterministic register automaton. Then, $\text{lab}(L(A))$ is ω -regular.*

More precisely, it is recognised by a non-deterministic ω -automaton A^{lab} with $n(k + \gamma + 1)^k$ states, where n is the number of states of A , k its number of registers, and γ the number of constants. Its acceptance condition is $\pi_1^{-1}(\Omega)$, where Ω is the acceptance condition of A .

Proof. By Corollary 4.17, $\text{lab}(L(A)) = \text{lab}(L(A) \cap (\Sigma \times X)^\omega)$ for any k -renaming set X . Take some k -renaming set of size $k + \gamma + 1$, and use Proposition 4.1 to get an ω -automaton that recognises $L(A) \cap (\Sigma \times X)^\omega$. Finally, project away X . $\square$

Remark 4.13 Note that over $(\mathbb{D}, =, \mathbb{C})$ the satisfaction of a formula φ by a valuation v can be evaluated in time linear in φ , so A^{lab} can be constructed in time linear in its size (which is exponential in k). It can moreover be constructed *on-the-fly*, as the existence of a transition $(p, v) \xrightarrow[A^{\text{lab}}]{\sigma} (q, \lambda)$ in A^{lab} only depends on p, q, v, λ and σ .

Emptiness Problem

This property, along with the lower bound established in [28, Theorem 5.1], yields:

Theorem 4.20 ([21, Theorem 1] and [28, Theorem 5.1]) *The emptiness problem for non-deterministic register automata over $(\mathbb{D}, =, \mathbb{C})$ is PSPACE-complete.*

As for ω -automata, determinism does not reduce the complexity of the problem: the emptiness problem for deterministic register automata is PSPACE-complete.

Remark 4.14 (Duplicate values increase succinctness) In [111, Theorem 4], it was shown that the emptiness problem for finite-memory automaton is NP-complete. This lower complexity results from the fact that registers cannot take duplicate data values in the latter model. The PSPACE-hardness proof of [28, Theorem 5.1] heavily relies on this property, as each register is used to store a bit, i.e. either 0 or 1.

We have now gathered enough properties of the model to solve the register-bounded synthesis problem for specifications expressed as universal register automata over data domains with equality only. The reader who is mainly interested in this problem can thus jump to the next chapter (Chapter 5).

Universality Problem

In contrast with Theorem 4.20, it was shown in [101, Theorem 18] that given a non-deterministic register automaton, it is undecidable whether it accepts all data words, i.e. $L(A) = (\Sigma \times \mathbb{D})^\omega$.

Theorem 4.21 ([101]) *The universality problem for non-deterministic register automata is undecidable.*

Proof. The proof consists in a reduction from the Post Correspondence Problem, showing that a non-deterministic register automaton can recognise data words that are *not* encodings of a valid PCP solution, and we do not redo it here. However, in Section 4.8 (Universal Register Automata), we show how to construct a universal register automaton that recognises halting runs of a Turing machines, which yields another proof by duality (Theorem 4.50). $\square$

As a consequence, the containment and language equivalence problems are also undecidable.

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

[111]: Sakamoto and Ikeda (2000), ‘Intractability of decision problems for finite-memory automata’

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

Remark 4.15 (Unambiguity) Those problems are however decidable for the subclass of unambiguous register automata. A non-deterministic register automaton is *unambiguous* if any data word admits at most one run (whether it is accepting or not). Any deterministic register automaton is in particular unambiguous, but unambiguous automata are strictly more expressive than deterministic ones. For instance, the automaton of Figure 4.3 recognises the language of finite data words whose last data value appears before:

$$L_{\neq} = \{d_0 \dots d_n \mid \exists 0 \leq i < n, d_i = d_n\}$$

This language cannot be recognised by a deterministic register automaton,

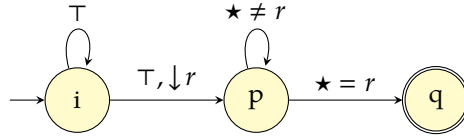


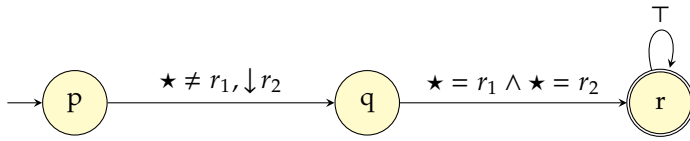
Figure 4.3.: An unambiguous register automaton with one register r over data domain $(\mathbb{D}, =)$ recognising $L_{\neq}$, that checks that last data value appears before. It is unambiguous as the $\star \neq r$ self-loop over p ensures that the transition from i to p was taken on the last occurrence of d_n before the end of the word.

since its complement $L_{\neq}^c$ cannot be recognised by any non-deterministic register automaton [38, Example 4] (without guessing; see Example 4.9).

Decidability of the containment and language equivalence problems for unambiguous register automata was first established in [112, Theorem 5], with a 2-ExpSPACE upper bound. It was later on improved to 2-ExpTIME by [113, Theorem 2]. Finally, [114, Theorem VIII.1] provides an ExpTIME upper bound, even when non-deterministic reassignment is allowed.

4.5.3. Local Concretisability

Unlike ω -automata, a path in a register automaton does not always correspond to a run (not necessarily accepting) over some data word. For instance, consider the register automaton of Figure 4.4. The path



$p \xrightarrow{\star \neq r_1, \{r_2\}} q \xrightarrow{\star = r_1 \wedge \star = r_2} r$ is not instantiable by any data word, since the first transition implies that in q , the content of r_1 and r_2 are distinct, while the second transition asks that they are equal. A desirable property would be to ensure that from all reachable configurations, any test labelling an outgoing transition corresponds to at least one data value. This way, a run in the syntactical automaton corresponds to a run in the semantical register automaton. In other words, we want to prune transitions that cannot be taken. We call this property local concretisability.

Definition 4.22 (Local concretisability) A register automaton A is said to be *locally concretisable* if for all reachable configurations $(p, \nu) \in \text{Configs}(A)$, and for all transitions $p \xrightarrow[A]{\sigma, \phi, \text{asgn}}_t q$, there exists $d \in \mathbb{D}$ such that $\nu, d \models \phi$.

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

[112]: Mottet and Quaas (2019), ‘The Containment Problem for Unambiguous Register Automata’

[113]: Barloy and Clemente (2021), ‘Bidimensional Linear Recursive Sequences and Universality of Unambiguous Register Automata’

[114]: Bojanczyk, Klin, and Moerman (2021), ‘Orbit-Finite-Dimensional Vector Spaces and Weighted Register Automata’

Figure 4.4.: A register automaton with two registers r_1 and r_2 that is syntactically non-empty, but accepts no data words.

In a locally concretisable register automaton, any path is a run. Thus, we get:

Observation 4.22 Let A be a locally concretisable register automaton. $L(A) = \emptyset$ if and only if $L(A^{\text{synt}}) = \emptyset$.

Recall that the syntactical automaton A^{synt} is the same automaton, where actions are interpreted as letters from a finite alphabet.

This property cannot be enforced for all data domains (cf Property 4.39) for the case of $(\mathbb{N}, <, 0)$, but we show that it is the case for $(\mathbb{D}, =, \mathbb{C})$ (Proposition 4.25), by storing additional information in the states. We later on prove that this is again the case of $(\mathbb{Q}, <, 0)$ (Proposition 4.37). Its applications are twofold. First, it yields yet another PSPACE decision procedure for the emptiness problem (Remark 4.18 and Theorem 4.36): once the register automaton is locally concretisable, it suffices to check that its syntactical automaton is empty (Observation 4.22). Second, it is instrumental in showing that the unbounded synthesis problem is decidable for specifications given by deterministic register automata over $(\mathbb{D}, =, \mathbb{C})$ (Theorem 7.15) and $(\mathbb{Q}, <, 0)$ (Theorem 7.16).

Explicit Tests

We already know that tests admit a normal form, that relies on full disjunctive normal form (Proposition 4.5). In the case of data domains with equality, maximally consistent tests are of the form

$$\phi := \bigwedge_{x,y \in R \sqcup \mathbb{C} \sqcup \{\star\}} x \bowtie_{x,y} y, \text{ where } \bowtie_{x,y} \in \{=, \neq\}$$

This can be decomposed as

$$\bigwedge_{x \in E} \star = x \wedge \bigwedge_{x \notin E} \star \neq x \wedge \bigwedge_{x,y \in R \sqcup \mathbb{C}} x \bowtie y \text{ for some } E \subseteq R \sqcup \mathbb{C}$$

Notation 4.23 In the following, for $E \subseteq R \sqcup \mathbb{C}$ we let

$$\phi_E := \bigwedge_{x \in E} \star = x \wedge \bigwedge_{x \notin E} \star \neq x$$

We call it an *explicit test*.

Then, observe that its constraint determines which tests can be taken from a valuation.

Observation 4.23 Any maximally consistent test ϕ can be written as $\phi_E \wedge \psi_\tau$, where ϕ_E is an explicit test and ψ_τ is a constraint formula.

Then, given a valuation $v : R \rightarrow \mathbb{D}$, we have that $v, d \models \phi$ if and only if $[v] = \tau$ and $E = \{x \in R \sqcup \mathbb{C} \mid v(x) = d\}$, i.e. $E = \emptyset$ or E is an equivalence class of $[v]$.

Remark 4.16 Explicit tests correspond to the formalism of [22, Definition 3.1], with the addition of constants.

[22]: Segoufin (2006), ‘Automata and Logics for Words and Trees over an Infinite Alphabet’

Definition 4.24 Given a constraint τ and an explicit test ϕ_E , we say that τ *enables* ϕ_E if $E = \emptyset$ or E is an equivalence class of τ . We extend the notion to maximally consistent tests by saying that τ *enables* $\phi_E \wedge \psi_\tau$ whenever τ enables ϕ_E and, additionally, $\tau = \tau'$.

It follows from the above observation that given a valuation ν and a transition $t = p \xrightarrow{\phi, \text{asgn}} q$ such that ϕ is explicit or maximally consistent, t is enabled from (p, ν) if and only if $[\nu]$ enables ϕ .

Update of Constraints

In Proposition 4.14, we established that constraints contain enough information to decide whether a transition can be crossed from a valuation. We now show that they can be updated.

Definition 4.25 Let τ be a constraint, and $\phi = \phi_E$ be an explicit test that is enabled by τ . For all $\text{asgn} \in 2^R$, we define the *successor* τ' of τ following action (ϕ, asgn) , written $\tau \xrightarrow{\phi, \text{asgn}} \tau'$, as the equivalence relation such that $r \sim_{\tau'} r'$ if either:

- $r, r' \notin \text{asgn} \cup E$ and $r \sim_{\tau} r'$
- $r, r' \in \text{asgn} \cup E$

In other words, it has the same equivalence classes as τ except that the class of E is extended with the elements of asgn . If E is empty, asgn thus forms a new equivalence class.

The notion is extended to maximally consistent tests of the form $\phi_E \wedge \psi_{\tau}$.

It is easy to check that the abstraction is correct:

Proposition 4.24 For all $\nu, \mu : X \rightarrow \mathbb{D}$ and all maximally consistent tests ϕ , we have that $[\nu] \xrightarrow{\phi, \text{asgn}} [\mu]$ if and only if there exists $d \in \mathbb{D}$ such that $\nu \xrightarrow{d, \phi, \text{asgn}} \mu$.

Proof. For the $\Rightarrow$ direction, take d such that $(\nu^{\mathbb{C}})^{-1}(d) = \phi$. The $\Leftarrow$ direction is routine. $\square$

Making a Register Automaton Locally Concretisable

We now have all the tools in hand to show the following:

Proposition 4.25 Any register automaton over $(\mathbb{D}, =, \mathbb{C})$ can be turned into a locally concretisable one.

More precisely, let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be a register automaton over $(\mathbb{D}, =, \mathbb{C})$. There exists a locally concretisable register automaton $A' = (Q \times \text{ER}(R \sqcup \mathbb{C}), I', \Sigma, R, \Delta', \Omega')$ such that $L(A) = L(A')$, and whose transitions are of the form $p \xrightarrow{\phi_E, \text{asgn}} q$ for some $E \subseteq R \sqcup \mathbb{C}$.

Proof. By Proposition 4.5, one can turn a register automaton into one whose tests are maximally consistent conjunctions of literals. The above study established that we can store the constraints in memory and update them along the run, which allows to determine whether a transition is enabled or not.

Let us formalise the argument. Let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be a register automaton over $(\mathbb{D}, =, \mathbb{C})$. Let $A'' = (Q, I, \Sigma, R, \Delta'', \Omega)$, be an equivalent

register automaton whose tests consist in maximal conjunctions of formulas of the form $x = y$ or $x \neq y$, for $x, y \in R \sqcup \mathbb{C} \sqcup \{\star\}$ (Proposition 4.5). Then, define $A' = (Q', I \times \{\llbracket v_R^t \rrbracket\}, \Sigma, R, \Delta', \Omega')$, where $Q' = Q \times \text{ER}(R \sqcup \mathbb{C})$ and $\Omega' = \pi_1^{-1}(\Omega)$. There is a transition $(p, \tau) \xrightarrow{\sigma, \phi_E, \text{asgn}} (q, \tau')$ if and only if $p \xrightarrow[\text{A}']{\sigma, \phi_E \wedge \psi_\tau, \text{asgn}} q$ and $\tau \xrightarrow{\phi_E, \text{asgn}} \tau'$.

Using Proposition 4.24, a straightforward induction establishes that there is a run $(p_0, v_0) \xrightarrow{\sigma_0, \phi_{E_0} \wedge \psi_{\tau_0}, \text{asgn}_0} (p_1, v_1) \dots$ in A'' if and only if there is a run $((p_0, v_0), \tau_0) \xrightarrow{\sigma_0, \phi_{E_0}, \text{asgn}_0} ((p_1, v_1), \tau_1) \dots$ in A' , where, for all $i \in \mathbb{N}$, $\llbracket v_i \rrbracket = \tau_i$. Moreover, since $\Omega' = \pi_1^{-1}(\Omega)$ the run in A' is accepting if and only if that in A'' is accepting. As a consequence, $L(A') = L(A'') = L(A)$.

Finally, A' is locally concretisable: as a consequence of Proposition 4.24, for any reachable configuration $C = ((p, \tau), v)$, we have that $\llbracket v \rrbracket = \tau$. The $\Rightarrow$ direction of Proposition 4.24 yields that for any transition $t = (p, \tau) \xrightarrow{\sigma, \phi_E, \text{asgn}} (q, \tau')$, there exists $d \in \mathbb{D}$ such that t is enabled from C . $\square$

Once constraints are stored in memory, explicit tests are a bit overkill, in the sense that they contain too much information.

Observation 4.26 Let $v : X \rightarrow \mathbb{D}$. For all explicit test ϕ_E , we have:

- If $E \cap R \neq \emptyset$, then v enables ϕ_E if and only if v enables $\star = r$
- If $E \cap \mathbb{C} \neq \emptyset$, then v enables ϕ_E if and only if v enables $\star = c$
- If $E = \emptyset$ then v enables ϕ_E if and only if v enables $\text{locFresh} := \bigwedge_{r \in R} \star \neq r$ (Notation 4.17).

As a consequence, the tests of the transition can be replaced with the above ones, yielding a more precise evaluation of the number of transitions.

Proposition 4.27 Let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be a register automaton over $(\mathbb{D}, =, \mathbb{C})$. There exists a locally concretisable register automaton $A' = (Q \times \text{ER}(R \sqcup \mathbb{C}), I', \Sigma, R, \Delta', \Omega')$ such that $L(A) = L(A')$, and whose transitions are of the form:

- $p \xrightarrow{\star = r, \text{asgn}} q$ for some $r \in R$
- $p \xrightarrow{\star = c, \text{asgn}} q$ for some $c \in \mathbb{C}$
- $p \xrightarrow{\text{locFresh}(\star), \text{asgn}} q$.

Remark 4.17 In a single-assignment register automaton, the constraints can be simplified. Indeed, along a run, valuations are such that $v(r) = v(r') \Leftrightarrow v(r) = \#$ (Proposition 4.7). As a consequence, we do not need to keep equality relations between registers, but simply whether or not they are equal to some constant. This amounts to replacing constraints $\tau \in \text{ER}(R \sqcup \mathbb{C})$ by a function $f : R \rightarrow \mathbb{C} \sqcup \{\_ \}$, where $f(r) = c \in \mathbb{C}$ means that $v(r) = c$, and $f(r) = \_$ means that $v(r) \notin \mathbb{C}$.

Remark 4.18 By definition, a locally concretisable register automaton is non-empty if and only if its syntactical automaton is non-empty (Observation 4.22). Since the construction of Proposition 4.27 can be conducted on-the-fly, we get another proof that the emptiness problem

for register automata is in PSPACE, hence PSPACE-complete by the lower bound of [28, Theorem 5.1] (Theorem 4.20).

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

4.5.4. Action Sequences

Constraints also prove useful to show that feasible action sequences are ω -regular for $(\mathbb{D}, =, \mathbb{C})$ (Proposition 4.29) and for $(\mathbb{Q}, <, 0)$ (Proposition 4.34). The notions of action sequence and compatible data words formalise the connection between syntax and semantics for register automata, and are defined for all data domains. We later on show that they are not ω -regular over $(\mathbb{N}, <, 0)$ (Property 4.38).

Definition 4.26 (Action Sequence) An (infinite) *action sequence* over data domain $\mathcal{D}$ and set of registers R is an infinite sequence

$$\alpha = \phi_1 \text{asgn}_1 \phi_2 \text{asgn}_2 \cdots \in (\text{Tests}_{\mathcal{D}}(R) 2^R)^\omega$$

We denote by $\text{ActSeq}_{\mathcal{D}}(R)$ the set of action sequences over $\mathcal{D}$ and R .

Definition 4.27 (Compatible Data Word) A data word $x = d_0 d_1 \cdots \in \mathbb{D}^\omega$ is *compatible* with α whenever there exists an infinite sequence of valuations $(v_i)_{i \in \mathbb{N}} \in (\text{Val}_{\mathcal{D}}(R))^\omega$ such that $v_0 = v_R^l$ and for all $i \in \mathbb{N}$, $v_i \{\star \leftarrow d_i\} \models \phi_i$ and $v_{i+1} = v_i \{\text{asgn}_i \leftarrow d_i\}$. In other words, $x \in \mathbb{D}^\omega$ is compatible with α whenever there is a path over $x \otimes \alpha$ in the data processing transition system $\mathcal{T}_R^{\mathcal{D}}$.

Remark 4.19 This is equivalent to saying that x is compatible with α whenever there exists a register automaton (with no labels) which admits an initial run over x whose transitions are labelled by α .

Definition 4.28 (Feasible Action Sequence) Given an action sequence α , we then define the set of compatible data words as:

$$\text{Comp}(\alpha) = \{x \in \mathbb{D}^\omega \mid x \text{ is compatible with } \alpha\}$$

An action sequence α is *feasible* if $\text{Comp}(\alpha) \neq \emptyset$. The set of feasible action sequences over R is denoted as:

$$\text{Feasible}_{\mathcal{D}}(R) = \{\alpha \in \text{ActSeq}_{\mathcal{D}}(R) \mid \text{Comp}(\alpha) \neq \emptyset\}$$

It is straightforward to extend the notion of action sequence to that of labelled action sequence, which is an infinite sequence in $(\Sigma \times \text{Tests}_{\mathcal{D}}(R) 2^R)^\omega$. The notion of compatible labelled data words is correspondingly extended. Since labels play no role in the construction, the focus is rather set on action sequences with no labels.

Feasible Action Sequences over $(\mathbb{D}, =, \mathbb{C})$

Constraints represent the equality relations between the registers of a valuation. As such, they allow to check that an action sequence is feasible. Since any test is equivalent to a disjunction of maximally consistent tests (Equation 4.1), we can extend the successor relation of constraints to tests. In the process, we however lose determinism, in the sense that a constraint can now have multiple successors.

$\mathcal{D}$ may be omitted when it is clear from the context.

Definition 4.29 Let τ be a constraint, and $(\phi, \text{asgn}) \in \text{Actions}_{\mathbb{D}}(R)$ be an action. We say that τ' is a *successor* of τ on action (ϕ, asgn) whenever there exists a maximally consistent test ψ such that $\phi \wedge \psi$ is satisfiable and $\tau \xrightarrow{\psi, \text{asgn}} \tau'$.

Example 4.5 Let $v : r_1 \rightarrow a, r_2 \rightarrow b, r_3 \rightarrow b$ where $a, b \in \mathbb{D}$. For simplicity, we assume there are no constants in this example. Let $\phi := \star = r_1 \wedge \star \neq r_2$. Let $\tau = [v] = \{\{r_1\}, \{r_2, r_3\}\}$, and let $\psi := \star = r_1 \wedge \star \neq r_2 \wedge \star \neq r_3 \wedge \psi_\tau$. We have that $\tau \xrightarrow{\psi, \{r_2\}} \tau'$, where $\tau' = \{\{r_1, r_2\}, \{r_3\}\}$.

Correspondingly, $v \xrightarrow{a, \phi, \{r_2\}} \lambda$, where $\lambda : r_1 \rightarrow a, r_2 \rightarrow a, r_3 \rightarrow b$ is such that $\tau\lambda = \tau'$.

We can then extend Proposition 4.28 to all tests.

Proposition 4.28 For all $v, \mu : X \rightarrow \mathbb{D}$ and all tests ϕ , we have that $[v] \xrightarrow{\phi, \text{asgn}} [\mu]$ if and only if there exists $d \in \mathbb{D}$ such that $v \xrightarrow{d, \phi, \text{asgn}} \mu$.

As a consequence, we get that the set of feasible action sequences is ω -regular:

Proposition 4.29 For any finite subset $\Phi \subset_f \text{Tests}(R)$, there exists a non-deterministic Büchi automaton A_Φ^R over the alphabet $\Phi \times 2^R$ with states $\text{ER}(R)$ and a trivial acceptance condition which accepts $\text{Feasible}(R) \cap (\Phi 2^R)^\omega$.

Proof. The ω -automaton starts in $[v_R^t]$. When in state τ , on reading an action (ϕ, asgn) , it guesses a successor constraint τ' and transitions to state τ' . The successor relation over constraints can be computed as follows: guess a maximally consistent test ψ such that $\phi \wedge \psi$ is satisfiable and check that τ' is indeed the successor of τ on action (ψ, asgn) . Now, let $\alpha \in (TA)^\omega$ be an action sequence. Assume that there exists $w \in \text{Comp}(\alpha)$. Then, there exists a sequence of valuations $(v_i)_{i \in \mathbb{N}}$ such that $v_0 = v_R^t$ and for all $i \in \mathbb{N}$, $v_i \xrightarrow{w[i], \phi_i, \text{asgn}_i} v_{i+1}$. Then, for each $i \in \mathbb{N}$, $[v_i] \xrightarrow{\phi_i, \text{asgn}_i} [v_{i+1}]$, so this yields a run in the automaton, which is accepting since all runs are.

Conversely, let $\rho = \tau_0 t_0 \tau_1 t_1 \dots$ be a run of the automaton over some action sequence α . We can build by induction a sequence of valuations $(v_i)_{i \in \mathbb{N}}$ and a word $w \in \mathbb{D}^\omega$ such that $w \in \text{Comp}(\alpha)$ and for all $i \in \mathbb{N}$, $[v_i] = \tau_i$. We take $v_0 = v_R^t$. Now, if v_i and $w[:i]$ have been built, we know that $\tau_i \xrightarrow{\phi_i, \text{asgn}_i} \tau_{i+1}$. Thus, there exists a maximally consistent test

ψ_i such that $[v_i] \xrightarrow{\psi_i, \text{asgn}_i} \tau_{i+1}$. $\psi_i = \phi_{E_i} \wedge \psi_{\tau_i}$, where E_i is either an equivalence class of $[v_i]$ or the empty set. In both cases, there exists $d_i \in \mathbb{D}$ such that $(v_i)^{-1}(d_i) = E_i$. By letting $w[i] = d_i$ and $v_{i+1} = v\{\text{asgn}_i \leftarrow d_i\}$, we have that $[v_{i+1}] = \tau_{i+1}$. Overall, we get that $w \in \text{Comp}(\alpha)$. $\square$

This property yields another proof that the projection over labels of a data word language recognised by a non-deterministic register automaton is ω -regular (Proposition 4.19)

Proposition 4.30 Let A be a non-deterministic register automaton. $\text{lab}(L(A))$ is ω -regular.

An acceptance condition W is *trivial* when $W = Q^\omega$. A word is thus accepted as soon as it admits a run.

We intersect with the finite subset Φ to get a finite input alphabet.

By Remark 4.12, it actually suffices to work with valuations that take their values in $\{0, \dots, c+k\}$ instead of constraints, where $k = |R|$ and $c = |\mathbb{C}|$. Then, the successor relation over constraints corresponds to the successor relation over configurations, restricted to this finite subset of data values. This is more efficient, but we prefer to describe a generic method, since it is applied later on to $(\mathbb{Q}, <, 0)$.

The state space is bigger, but we provide this result to prepare for the case of $(\mathbb{Q}, <, 0)$, where the renaming property does not hold anymore (cf Property 4.33).

More precisely, it is recognised by a non-deterministic ω -automaton A^{lab} with states $Q \times \text{ER}(R)$ and acceptance condition $\pi_1^{-1}(Q)$.

Proof. Let Φ be the set of tests used by A . Let A_Φ^R be an automaton recognising $\text{Feasible}(R) \cap (\Phi 2^R)^\omega$ (Proposition 4.29). Then, the product of A and A_Φ^R recognise the accepting runs of A . Projecting on Σ yields $\text{lab}(L(A))$. $\square$

4.5.5. Constraints and Types

We hinted at the fact that constraints entertain a tight relation with the notion of type from model theory. We now make the connection explicit, to shed a different light on the reason why they abstract the behaviour of register automata, that allows to generalise them to the case of $(\mathbb{Q}, <, 0)$. Actually, this method provides a way to decide emptiness for any data domain whose finite tuples have finitely many possible types, even in the presence of non-deterministic reassignment, as shown in Section 12.2.1 in Part II.

Types

The notion is defined for any data domains; in the following, we apply it to $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, 0)$. A type characterises a valuation with regard to predicates of the domain.

Definition 4.30 (Type) Let $\nu : X \rightarrow \mathbb{D}$ be a valuation over X . The *type* of ν in $\mathcal{D}$ is the set $\text{type}_{\mathcal{D}}(\nu)$ of *first-order* formulas $\varphi(X)$ (i.e. we allow the use of quantifiers) that ν satisfies over domain $\mathcal{D}$.

The set of types over $\mathcal{D}$ is denoted $\text{Types}(\mathcal{D})$.

Some data domains, in particular $(\mathbb{N}, =)$ and $(\mathbb{Q}, <)$, admit only finitely many distinct types for all finite sets of variables. Actually, they even enjoy quantifier elimination (see, e.g., [115, Section 2.3]), which is a sufficient condition.

Proposition 4.31 Let $\mathcal{D} = (\mathbb{D}, =, \mathbb{C})$. For all $\nu, \nu' : X \rightarrow \mathbb{D}$, we have that $[\nu] = [\nu']$ if and only if $\text{type}_{\mathcal{D}}(\nu) = \text{type}_{\mathcal{D}}(\nu')$.

Proof. Let $\nu, \nu' : X \rightarrow \mathbb{D}$. First, assume that $\text{type}(\nu) = \text{type}(\nu')$. Let $\varphi(X) := \bigwedge_{x,y \in X \sqcup \mathbb{C}} x \bowtie_{x,y} y$, where $\bowtie_{x,y} \in \{=, \neq\}$ is $=$ if and only if $\nu(x) = \nu(y)$. By construction, $\nu \models \varphi$, so $\nu' \models \varphi$ as they have the same type. This implies that $[\nu] = [\nu']$, since φ characterises $[\nu]$.

Conversely, assume that $[\nu] = [\nu']$, and let φ be some first-order formula over X . Since $(\mathbb{D}, =, \mathbb{C})$ admits quantifier elimination [115, Section 2.3], we know that there exists a quantifier-free formula $\psi(X)$ such that $\psi \equiv \varphi$. Since $[\nu] = [\nu']$, they satisfy the same quantifier-free formulas (Lemma 4.11), which concludes the proof. $\square$

Remark 4.20 The above result allows to get Lemma 4.11, as it even holds for the more general case of first-order formulas.

In $(\mathbb{Q}, <, 0)$, constraints can be enriched to again characterise types:

Note that in general, $\text{type}(\nu)$ is infinite.

We simply write $\text{type}(\nu)$ when $\mathcal{D}$ is clear from the context.

These domains are called oligomorphic. Section 12.2 in Part II further discusses them.

[115]: Marcja and Toffalori (2003), ‘Quantifier Elimination’

As usual, we extend valuations to constants by letting $\nu(c) = c$ for all $c \in \mathbb{C}$.

The result is established for dense linear orders, but a formula over $(\mathbb{D}, =, \mathbb{C})$ can be seen as a formula over $(\mathbb{D}, =, <, \mathbb{C})$ that simply does not use $<$, where $<$ is a dense linear order over $\mathbb{D}$.

Definition 4.31 Let $\mathcal{D} = (\mathbb{Q}, <, 0)$, and let $\nu : X \rightarrow \mathbb{Q}$. We associate to it an equivalence relation $[\nu]_{\mathbb{Q}}$ whose classes are ordered (we again denote it $[\nu]$ when the data domain is clear from the context). More precisely, for all $x, y \in X \sqcup \{0\}$, $x \sim y$ whenever $\nu(x) = \nu(y)$ and $[x] < [y]$ (resp. $[x] > [y]$) whenever $\nu(x) < \nu(y)$ (resp. $\nu(x) > \nu(y)$), where we set $\nu(0) = 0$. The set of constraints over $(\mathbb{Q}, <, 0)$ with variables X is denoted $\text{Constr}_{\mathbb{Q}}(X)$. A constraint can be represented as a total preorder over R , which is in particular a binary relation over R , so the size of $\text{Constr}_{\mathbb{Q}}(R)$ is bounded by $2^{|R|^2}$.

Proposition 4.32 Let $\mathcal{D} = (\mathbb{Q}, <, 0)$. For all $\nu, \nu' : X \rightarrow \mathbb{Q}$, we have that $[\nu]_{\mathbb{Q}} = [\nu']_{\mathbb{Q}}$ if and only if $\text{type}_{\mathbb{Q}}(\nu) = \text{type}_{\mathbb{Q}}(\nu')$.

Proof. The proof relies on the same argument as for Proposition 4.31: if $\text{type}_{\mathbb{Q}}(\nu) = \text{type}_{\mathbb{Q}}(\nu')$, then we know that ν satisfies $\bigwedge_{x,y \in X \sqcup \{0\}} x \bowtie_{x,y} y$, where $\bowtie_{x,y} \in \{<, =, >\}$ characterises the relative order of x and y in ν . As a consequence, ν' satisfies the same formula, which exactly specifies $[\nu']_{\mathbb{Q}}$.

Conversely, $(\mathbb{Q}, <, 0)$ admits quantifier elimination [115, Section 2.3], so any first-order formula φ over X is equivalent to a quantifier-free formula ψ over X . By definition, $[\nu]_{\mathbb{Q}}$ determines which atomic formulas are satisfied by ν , so a straightforward structural induction yields that ν and ν' satisfy the same quantifier-free formulas. In particular, $\nu \models \psi$ if and only if $\nu' \models \psi$. $\square$

Example 4.6 Consider again the valuation ν of Example 4.2. Its type in $(\mathbb{N}, =)$ contains infinitely many formulas, but is characterised by the formulas $r_1 \neq r_2, r_1 \neq r_3, r_2 = r_3$, which correspond to the equivalence relation whose classes are $\{\{r_1\}, \{r_2, r_3\}\}$. In $(\mathbb{N}, =, 0)$, we add the information that $r_1 = 0$, so we summarise the type through the equivalence relation $\{\{0, r_1\}, \{r_2, r_3\}\}$.

In $(\mathbb{Q}, <)$, it is characterised by $r_1 < r_2, r_1 < r_3$ and $r_2 = r_3$, which corresponds to ordered equivalence classes, that we represent as an ordered tuple $(\{r_1\}, \{r_2, r_3\})$.

Note that in $(\mathbb{N}, =)$, ν has the same type as ν' defined as $\nu'(r_1) = 2, \nu'(r_2) = \nu'(r_3) = 1$, while their types are distinct in $(\mathbb{N}, =, 0)$ and $(\mathbb{Q}, <)$.

Finally, in $(\mathbb{N}, <, 0)$, all distinct valuations have a distinct type, already with a single register. Indeed, let $n \in \mathbb{N}$, and consider the first-order formula $\varphi_{\geq n}(x) := \exists y_0, \dots, \exists y_{n-1}, y_0 < y_1 \wedge y_1 < y_2 \cdots \wedge y_{n-2} < y_{n-1} \wedge y_{n-1} < x$. Then, for all $m \in \mathbb{N}$, $\varphi(m)$ holds if and only if $m \geq n$. Thus, if $\nu, \nu' : \{r_0\} \rightarrow \mathbb{N}$ are distinct, let $m = \nu(r_0)$ and $n = \nu'(r_0)$ and assume without loss of generality that $m < n$. $\text{type}_{\mathbb{N}}(\nu')$ contains $\varphi_n(x)$ while $\text{type}_{\mathbb{N}}(\nu)$ does not. More generally, a natural number $n \in \mathbb{N}$ is characterised by $\varphi_{=n} = \varphi_{\geq n} \wedge \neg \varphi_{\geq n+1}$.

This link between constraints and types allows to circumvent the ad-hoc construction of a data value that enables a transition, as was done in Lemma 4.12. We apply this to the case of $(\mathbb{Q}, <, 0)$.

Here, $[x]$ denotes the equivalence class of x for $\sim_{\nu}$.

A total preorder or weak order over X is a binary relation $\lesssim$ that models ordered sets with ties allowed. Formally, it is:

- transitive, i.e. for all $x, y, z \in X$, if $x \lesssim y$ and $y \lesssim z$, then $x \lesssim z$
- strongly connected, i.e. for all $x, y \in X$, $x \lesssim y$ or $y \lesssim x$ (note that this implies that $\lesssim$ is reflexive)

The number of weak orders over a set X of size n is the n -th Fubini number (also known as ordered Bell numbers). There is no simple closed form, which is why we use the coarse bound 2^{n^2} .

$r_1 \neq r_2$ is a short for $\neg(r_1 = r_2)$.

Recall that $=$ can be derived in $(\mathbb{Q}, <)$.

4.6. Non-Deterministic Register Automata over a Densely Ordered Domain

Let us extend our study to the case of a densely ordered data domain. For simplicity, we let $\mathcal{D}$ be $(\mathbb{Q}, <, 0)$, but the reader can check that the properties we exhibit hold for any domain $(\mathbb{D}, <, \mathcal{C})$ where $\mathbb{D}$ is infinite, $<$ is a dense order and $\mathcal{C}$ is a finite set of constants.

Definition 4.32 (Dense order) Let $\mathbb{D}$ be a set, and $<$ an order relation. We say that $<$ is *dense* when for all $x, y \in \mathbb{D}$ such that $x < y$, there exists $z \in \mathbb{D}$ such that $x < z < y$.

First, one can notice that the renaming property does not hold anymore over this data domain, even for deterministic register automata:

Property 4.33 Consider again $L_{<}^{\mathbb{Q}} = \{d_0 d_1 \dots \in \mathbb{Q}^\omega \mid \forall i < j, d_i < d_j\}$. It is recognised by the deterministic register automaton over $(\mathbb{Q}, <, 0)$ of Figure 4.1c on page 81. It is easy to see that $L_{<}^{\mathbb{Q}}$ contains no data word with only finitely many distinct data. In particular, one cannot hope to get an analogue of Proposition 4.15 for register automata $(\mathbb{Q}, <)$.

Over this data domain, we only study non-deterministic register automata, since universal ones do not behave as well, as explained in Section 4.8.

While non-deterministic register automata over $(\mathbb{Q}, <, 0)$ do not have the renaming property, we show that their projection over labels is ω -regular. To this end, we demonstrate that the set of feasible action sequences is ω -regular (when restricted to a finite set of actions).

Proposition 4.34 For any finite subset $T \subset_f \text{Tests}_{\mathbb{Q}}(R)$, there exists a non-deterministic ω -automaton A_T^R over the alphabet $T \times 2^R$ with states $\text{Constr}_{\mathbb{Q}}(R)$ and a trivial acceptance condition which accepts $\text{Feasible}_{\mathbb{Q}}(R) \cap (T2^R)^\omega$.

Proof. The ω -automaton starts in $[v_R^t]_{\mathbb{Q}}$. Then, observe that the successor relation over constraints can be expressed as a first-order formula: there is a transition $\tau \xrightarrow{\phi, \text{asgn}} \tau'$ if and only if

$$\begin{aligned} \psi_{\tau, \tau'}^{\phi, \text{asgn}}(r_1, \dots, r_k, r'_1, \dots, r'_k) := & \exists x_d \cdot \psi_{\tau}(r_1, \dots, r_k) \wedge \psi_{\tau'}(r'_1, \dots, r'_k) \wedge \\ & \phi(r_1, \dots, r_k, x_d) \wedge \\ & \bigwedge_{r \in \text{asgn}} r' = x_d \wedge \bigwedge_{r \notin \text{asgn}} r' = r \end{aligned}$$

is satisfiable.

Let $\alpha \in (TA)^\omega$ be an action sequence. Assume that there exists $w \in \text{Comp}(\alpha)$. Then, there exists a sequence of valuations $(v_i)_{i \in \mathbb{N}}$ such that $v_0 = v_R^t$ and for all $i \in \mathbb{N}$, $v_i \xrightarrow{w[i], \phi_i, \text{asgn}_i} v_{i+1}$. Then, for each $i \in \mathbb{N}$, $[v_i] \xrightarrow{\phi_i, \text{asgn}_i} [v_{i+1}]$, by instantiating x_d with $w[i]$, r_j by $v_i(r_j)$ and r'_j by $v_{i+1}(r_j)$ in the above formula $\psi_{\tau, \tau'}^{\phi, \text{asgn}}$, so this yields a run in the automaton, which is accepting since all runs are.

Conversely, let $\rho = \tau_0 t_0 \tau_1 t_1 \dots$ be a run of the automaton over some action sequence α . We can build by induction a sequence of valuations $(v_i)_{i \in \mathbb{N}}$ and a word $w \in \mathbb{D}^\omega$ such that $w \in \text{Comp}(\alpha)$ and for all $i \in$

An acceptance condition W is *trivial* when $W = Q^\omega$. A word is thus accepted as soon as it admits a run.

We intersect with the finite subset Φ to get a finite input alphabet.

$\mathbb{N}$, $[v_i] = \tau_i$. We take $v_0 = v^t$. Now, if v_i and $w[:i]$ have been built, we know that $\tau_i \xrightarrow{\phi_i, \text{asgn}_i} \tau_{i+1}$. Thus, there exists v'_i, v'_{i+1} such that $\psi_{\tau_i, \tau_{i+1}}^{\phi_i, \text{asgn}_i}(v'_i(r_1), \dots, v'_i(r_k), v'_{i+1}(r_1), \dots, v'_{i+1}(r_k))$ holds, by taking some valuation d' for x_d . By definition of $\psi_{\tau, \tau'}^{\phi, \text{asgn}}$, this implies that $[v'_i] = \tau_i = [v_i]$. By Proposition 4.32, this means that a first-order formula over v_i is satisfiable if and only if it is satisfiable over v'_i , so we know that $\psi_{\tau_i, \tau_{i+1}}^{\phi_i, \text{asgn}_i}(v_i(r_1), \dots, v_i(r_k), r'_1, \dots, r'_k)$ is satisfiable. Take a valuation v_{i+1} of $r'_1, \dots, r'_k$ that satisfy the formula, and take some d as a value of x_d that is existentially quantified. We then have that $v_i \xrightarrow{d, \phi, \text{asgn}} v_{i+1}$, and moreover $[v_{i+1}] = \tau_{i+1}$ since v_{i+1} satisfies the formula, which in particular determines the constraint.

Overall, we get that $w \in \text{Comp}(\alpha)$. $\square$

By applying the same construction as for Proposition 4.30, we get:

Proposition 4.35 *The projection over labels of a non-deterministic register automaton over $(\mathbb{Q}, <, 0)$ is ω -regular.*

More precisely, it is recognised by a non-deterministic ω -automaton A^{lab} with states $Q \times \text{Constr}_{\mathbb{Q}}(R)$ and acceptance condition $\pi_1^{-1}(Q)$.

In , we noted that the size of $\text{Constr}_{\mathbb{Q}}(R)$ is bounded by $2^{(|R|+1)^2}$, so we can decide emptiness in polynomial space (a result known from [39]).

Theorem 4.36 ([39, Theorem 21]) *The emptiness problem for deterministic and non-deterministic register automata over $(\mathbb{Q}, <, 0)$ is PSPACE-complete.*

Proof. Let A be a register automaton over $(\mathbb{Q}, <, 0)$. $L(A) \neq \emptyset$ if and only if $\text{lab}(L(A)) \neq \emptyset$. By Proposition 4.35, we can construct an ω -automaton with exponentially many states, that can be described in polynomial space, and that recognises $\text{lab}(L(A))$. Since emptiness for our ω -automata is in polynomial time (Theorem 3.5), we get a PSPACE decision procedure.

The lower bound is directly inherited from PSPACE-hardness of the emptiness problem for register automata over $(\mathbb{N}, =, 0)$, since a register automaton over $(\mathbb{Q}, <, 0)$ can easily simulate one over $(\mathbb{D}, =, \#)$, by writing $x \neq y$ as $x < y \vee x > y$ and $x = y$ as $\neg(x \neq y)$. $\square$

Finally, as in the case of $(\mathbb{D}, =, \mathbb{C})$, we can do the product with the ω -automaton $A_{\text{MCTests}_{\mathbb{Q}}(R)}^R$ that recognises feasible action sequences over maximally consistent tests to get a locally concretisable automaton (see Proposition 4.25).

Proposition 4.37 *Any non-deterministic register automaton over $(\mathbb{Q}, <, 0)$ can be turned into a locally concretisable one.*

More precisely, let A be a register automaton over $(\mathbb{Q}, <, 0)$ with states Q . There exists an equivalent register automaton A' with states $Q \times \text{Constr}_{\mathbb{Q}}(R)$ that is locally concretisable.

Proof. To lighten the notations, we denote $\Phi = \text{MCTests}_{\mathbb{Q}}(R)$ the set of maximally consistent tests in $(\mathbb{Q}, <, 0)$ over registers R . Let A_{Φ}^R be the Büchi automaton of Proposition 4.34 that recognises $\text{Feasible}(R) \cap (\Phi 2^R)^{\omega}$. Then, the product of A and A_{Φ}^R recognises the runs of A that are feasible,

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

Recall that $\text{MCTests}_{\mathbb{Q}}(R)$ denotes the set of maximally consistent tests in $(\mathbb{Q}, <, 0)$ over R .

labelled with maximally consistent tests. In the presence of maximally consistent tests, the successor constraint is unique, which enforces that along the run, the constraint stored in the state indeed correspond to that of the current valuation. Since the constraints determine whether a transition can be taken, we get that the resulting automaton is locally concretisable. $\square$

4.7. Register Automata over a Discretely Ordered Domain

4.7.1. Indefinite Growth in a Bounded Universe: Exhibiting Non-Regular Behaviours

Let us now examine the case of a discrete order, more precisely of the data domain $(\mathbb{N}, <, 0)$. This setting is more technical than the previous ones, for the following reason.

Property 4.38 *The set of feasible action sequences over $(\mathbb{N}, <, 0)$ and the projection over labels of register automata with data domain $(\mathbb{N}, <, 0)$ are not ω -regular.*

Proof. For instance, consider the language of data words in $(\mathbb{N}, <, 0)$ consisting in an initial bound M labelled b , and then of finite sequences of data labelled a that are strictly increasing, bounded from above by M , and separated by occurrences of $(b, 0)$. Formally,

$$L_{\rightarrow} = \left\{ \begin{array}{l} (b, M)(a, d_0^0) \dots (a, d_{n_0}^0) \\ (b, 0)(a, d_0^1) \dots (a, d_{n_1}^1)(b, 0) \dots \end{array} \mid \begin{array}{l} M \in \mathbb{N} \text{ and for all } i \in \mathbb{N} \\ 0 \leq d_0^i < \dots < d_{n_i}^i < M \end{array} \right\}$$

It is recognised by the deterministic register automaton of Figure 4.5. Then, its projection over labels is

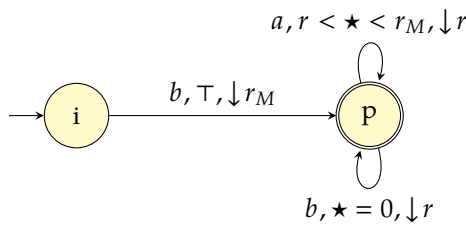


Figure 4.5: A deterministic register automaton over $(\mathbb{N}, <, 0)$ with two registers r and r_M that recognises $L_{\rightarrow}$. It initially reads the upper bound M labelled by b . Then, in state p , it reads a and checks that their data values strictly increase with its register r (recall that registers initially contain 0); it checks at the same time that they are below M . Everytime it reads a b , it resets r to 0.

$$\text{lab}(L_{\rightarrow}) = \{ba^{n_0}ba^{n_1}\dots \mid \exists M \in \mathbb{N}, \forall i \in \mathbb{N}, n_i \leq M\}$$

Note that this is exactly L_{aB} of Example 4.7.2. A simple pumping argument establishes that the above language is not ω -regular [59]. $\square$

As a consequence, one cannot hope to turn a register automaton over $(\mathbb{N}, <, 0)$ into a locally concretisable one.

Property 4.39 *There exists a register automaton over $(\mathbb{N}, <, 0)$ that has no equivalent locally concretisable register automaton.*

[59]: Bojańczyk and Colcombet (2006), ‘Bounds in ω -Regularity’

Proof. When a register automaton is locally concretisable, its syntactical automaton recognises $\text{lab}(A)$ when projecting away the actions (Observation 4.22). This implies that $\text{lab}(A)$ is ω -regular. Thus, the register automaton of Figure 4.5 on page 108 does not admit a locally concretisable equivalent. $\square$

4.7.2. Constraint Sequences in $\mathbb{N}$

Unlike in $(\mathbb{Q}, <, 0)$, it is not anymore sufficient to check that an action sequence induce constraints that are locally consistent with tests. For instance, consider the action sequence over a single register r that repeatedly asks to input a data value that is strictly below the content of r , and to store it in r , defined as $\alpha = (\top\{r\})((r > \star)\{r\})^\omega$. It is consistent but not feasible, as it would require the existence of an infinite descending chain in $\mathbb{N}$. One now need a global information over the action sequence. We model this information through the notion of constraint sequence, which relates successive values of the registers. This notion is equivalent to the 2-frame sequences of [116, Section 3], where constraints are called frames (they use k -frame sequences to model dependencies over several steps, to be able to handle LTL formulas like $X^k(\phi)$). Some results in this section rely on ideas similar to those in [116, Sections 6 and 7]; we provide standalone proofs as it yields a more unified presentation of our results and avoids the introduction of additional notations. Moreover, we need a finer characterisation to get recognisability by a deterministic max-automaton (Theorem 4.46).

[116]: Demri and D’Souza (2007), ‘An automata-theoretic approach to constraint LTL’

Definition 4.33 (Constraint Sequence) Let R be a set of registers, and $R' = \{r' \mid r \in R\}$ be a primed copy of R . We define $\text{unprime} : R' \rightarrow R$ in the natural way. We now consider constraints over $R \sqcup R'$, that correspond to maximally consistent conjunctions of atomic formulas of the form $r \bowtie s$ for $r, s \in R \sqcup R'$ and $\bowtie \in \{<, >, =\}$. To lighten the notations, we often write them as sets of atomic formulas instead of conjunctions.

A *constraint sequence* is then a sequence $\tau_0 \tau_1 \dots$ of constraints over $R \sqcup R'$. It is *consistent* if for all $i \in \mathbb{N}$, $\tau_{i+1}|_R = \text{unprime}(\tau_i|_{R'})$, where unprime is extended to constraints in the expected way.

$\tau_{i+1}|_R$ denotes the restriction of τ_{i+1} to the set of registers R .

A valuation $v : R \sqcup R' \rightarrow \mathbb{N}$ is *compatible* with a constraint τ , written $v \models \tau$, if every atomic formula holds when we replace every $r \in R \sqcup R'$ by $v(r)$. A constraint sequence $\tau_0 \tau_1 \dots$ is *feasible* in $\mathbb{N}$ if there exists a sequence of valuations $v_0 v_1 \dots \in (\mathbb{N}^R)^\omega$ such that $v_i \cup v_{i+1}' \models \tau_i$ for all $i \geq 0$. If, additionally, $v_0 = v_R' : r \mapsto 0$, then it is *0-feasible*. Notice that feasibility implies consistency. Feasibility and 0-feasibility in $\mathbb{Q}$ are defined analogously. Given a set R of registers, we denote $\text{FeasibleCS}_{\mathbb{D}}(R)$ (respectively, $\text{FeasibleCS}_{\mathbb{D}}^0(R)$) the set of feasible constraint sequences (resp., 0-feasible constraint sequences) over R , over data domain $\mathbb{D}$. When the data domain is not specified, we are considering feasibility in $\mathbb{N}$.

In this section, we distinguish between feasibility and 0-feasibility to separate the difficulties linked with chains and those related to 0.

The mention of $\mathbb{D}$ is omitted when clear from the context.

Example 4.7 Let $R = \{r_1, r_2, r_3, r_4\}$. consider a consistent constraint sequence $\tau_0 \tau_1 \dots$ that starts with

$$\{r_1 < r_2 < r_3 < r_4, r_4 = r_3', r_3 = r_4', r_1 = r_1', r_1 > r_2'\} \{r_2 < r_1 < r_4 < r_3, r_4 = r_3', r_3 = r_4', r_1 = r_1', r_2 > r_1'\}$$

Note that we omit some atomic formulas in τ_0 and τ_1 for readability: although they are not maximal (e.g. τ_0 does not contain $r_2' < r_1' < r_4' < r_3'$),

they can be uniquely completed to maximally consistent sets. Figure 4.6 (ignore the colored paths for now) depicts $\tau_0\tau_1$ plus a bit more constraints. The black lines represent the evolution of a single register. The constraint

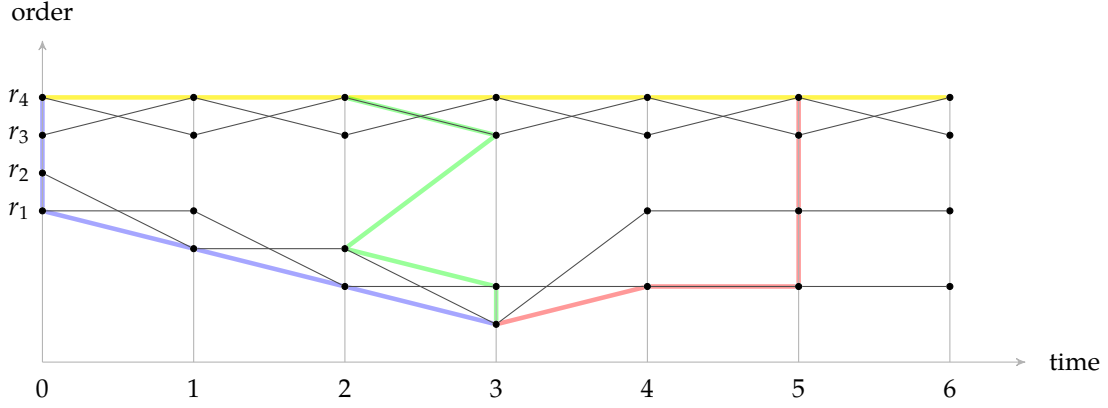


Figure 4.6.: Graphical depiction of a constraint sequence. Register values are depicted as black dots, and dots are connected by black lines when they are related to the same register. Coloured paths depict chains (cf infra).

τ_0 describes the transition from moment 0 to 1, and C_1 , from 1 to 2. This finite constraint sequence is feasible both in $\mathbb{Q}$ and $\mathbb{N}$. For instance, the valuation sequence can start with $v_0 : r_4 \mapsto 6, r_3 \mapsto 5, r_2 \mapsto 4, r_1 \mapsto 3$. Note however that no valuation sequence starting with $v_0(r_3) < 5$ can satisfy the sequence in $\mathbb{N}$. Also, the constraint τ_0 requires all registers in R to differ, hence the sequence is not 0-feasible in $\mathbb{Q}$ nor in $\mathbb{N}$.

Another example is given by the constraint sequence $(\{r > r'\})^\omega$ with $R = \{r\}$, which results from the action sequence $(\top\{r\})((r > \star)\{r\})^\omega$ that we described earlier: it is feasible in $\mathbb{Q}$ but not in $\mathbb{N}$.

Feasibility of constraint sequences in $\mathbb{N}$

In this section, we study how to check the feasibility of constraint sequences in $(\mathbb{N}, <, 0)$. In [39, Appendix C], Segoufin and Torunczyk show that feasible constraint sequences in $\mathbb{N}$ are recognisable by non-deterministic ω B-automata [59, Section 3.2]. We provide a characterisation of feasible constraint sequences that allows to refine this result to *deterministic max-automata* [117, Section 2]. As a corollary, we get that they are expressible in weak MSO with the unbounding quantifier, since it recognises the same languages as deterministic max-automata [117, Theorem 5].

The characterisation is based on the study of chains of registers, i.e. of sequences of registers with constraints over their relative order given by the constraint sequence. The discrete structure of $(\mathbb{N}, <, 0)$ imposes some conditions on such chains, namely that they cannot be infinitely decreasing nor unboundedly increasing if they are bounded from above. In Lemma 4.41, we show that those conditions characterise feasibility. We later on refine this result by showing that it suffices to consider one-way chains, that can only go from left to right in the constraint sequence (Lemma 4.42).

This result was obtained during the quest for decidability of unbounded synthesis for deterministic one-sided specification automata over $(\mathbb{N}, <, 0)$

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

[59]: Bojańczyk and Colcombet (2006), ‘Bounds in ω -Regularity’

[117]: Bojańczyk (2011), ‘Weak MSO with the unbounding quantifier’

(cf Section 7.3.4): while games with a winning condition expressed by a non-deterministic ω B-automaton are not known to be decidable, those over deterministic max-automata can be shown decidable (cf Remark 7.10), which implies decidability of the synthesis problem. In the end, we use a different approach, that yields a better complexity. However, this result is interesting in its own right, since deterministic automata model are in general easier to handle than non-deterministic ones, as illustrated in the context of games: most game resolution techniques rely on some determinisation procedure, except for approaches based on good-for-games automata [97].

Definition 4.34 Fix R and a constraint sequence $\tau_0\tau_1\dots$ over R . A *two-way chain* is a finite or infinite sequence

$$\gamma = (r_0, m_0) \bowtie_0 (r_1, m_1) \bowtie_1 \dots \in ((R \times \mathbb{N}) \cdot \{<, =, >\})^{*,\omega}$$

satisfying the following (note that m_0 can be distinct from 0).

- $m_{i+1} = m_i$, or $m_{i+1} = m_i + 1$ (time flows forward), or $m_{i+1} = m_i - 1$ (time goes backwards).
- If $m_{i+1} = m_i$ then $(r_i \bowtie_i r_{i+1}) \in C_{m_i}$.
- If $m_{i+1} = m_i + 1$ then $(r_i \bowtie_i r'_{i+1}) \in C_{m_i}$.
- If $m_{i+1} = m_i - 1$ then $(r_{i+1} \bowtie_i r'_i) \in C_{m_i-1}$.

It is *decreasing* (respectively, *increasing*) if for all $0 \leq i < |\gamma|$, $\bowtie_i \in \{>, =\}$ (resp., $\bowtie_i \in \{<, =\}$). In the following, we only consider decreasing two-way chains. For such a chain, its *depth* is the number of $>$. When there are infinitely many, we say that the chain is *infinitely decreasing*. Figure 4.6 on page 110 shows four two-way decreasing chains, respectively depicted in blue, red, green and yellow. For instance, the green-colored chain $(r_4, 2) > (r_3, 3) > (r_2, 2) > (r_1, 3) > (r_2, 3)$ has depth 4.

A chain is *one-way* when time flows forward ($m_{i+1} = m_i + 1$) or stays the same ($m_{i+1} = m_i$), i.e. we do not allow $m_{i+1} = m_i - 1$. In Figure 4.6, the blue chain is one-way decreasing, the red chain is one-way increasing.

A *stable chain* is an infinite one-way chain

$$(r_0, m)' = (r_1, m + 1)' = (r_2, m + 2)' = \dots$$

i.e. for all $i \geq 0$, $m_{i+1} = m_i + 1$ and $\bowtie_i$ is $=$. Then, we also write $(m, r_0 r_1 r_2 \dots)$.

Given a stable chain $\sigma = (m, r_0 r_1 \dots)$ and a (finite or infinite) chain $\gamma = (s_0, n_0) \bowtie_0 (s_1, n_1) \bowtie_1 \dots$ such that for all $0 \leq i < |\gamma|$, $n_i \geq m$, we say that the chain σ is *non-strictly above* γ if for all n_i , the constraint τ_{n_i} contains $r_{n_i-m} > s_{n_i}$ or $r_{n_i-m} = s_{n_i}$. We also say that γ is *non-strictly below* σ .

A stable chain $(m, r_0 r_1 \dots)$ is *maximal* if it is non-strictly above all other stable chains that start after m . In Figure 4.6, the yellow chain $(0, (r_4 r_3)^\omega)$ is stable and non-strictly above all other chains, hence maximal.

A chain is *ceiled* if it is below a maximal stable chain.

First, let us show that we can avoid repetitions in chains that appear in feasible constraint sequences:

[97]: Henzinger and Piterman (2006), ‘Solving Games Without Determinization’

We put equality between quotes to highlight the fact that it is a *constraint symbol*: we do not mean that all $(r_i, m + i)$ are equal.

Lemma 4.40 *Let γ be chain. We say that it is looping if there exists $0 \leq i < j < |\gamma|$ such that $(r_i, m_i) = (r_j, m_j)$.*

Let κ be a feasible constraint sequence. If κ has an infinite (respectively, finite) chain of depth d starting from $(r, m) \in R \times \mathbb{N}$ (respectively, starting from $(r, m) \in R \times \mathbb{N}$ and ending in $(s, n) \in R \times \mathbb{N}$), then it admits a chain with the same properties that is non-looping.

Proof. Let κ be a feasible constraint sequence, and let $\gamma = (r_0, m_0) \bowtie_0 (r_1, m_1) \bowtie_1 \dots$ be a (finite or infinite) chain of depth d starting from $(r, m) \in R \times \mathbb{N}$ (and ending in $(s, n) \in R \times \mathbb{N}$ if it is finite) that appears in κ . We show how to remove a loop of γ ; the result is then obtained by induction on the number of remaining loops.

Assume that γ has a loop, and let $i, j \in \mathbb{N}$ be such that $(r_i, m_i) = (r_j, m_j)$. If there is at least an occurrence of $<$ or $>$ between i and j , then κ is not feasible, because it means that $v_{m_i}(r_i) < v_{m_j}(r_j)$ (respectively, $v_{m_i}(r_i) > v_{m_j}(r_j)$), we have that for all $i \leq k \leq j$, $\bowtie_k$ is $'='$. Thus, the chain $\gamma' = (r_0, m_0) \bowtie_0 \dots (r_i, m_i)(r_{j+1}, m_{j+1}) \dots$ obtained by removing the loop has the same depth and the same endpoint(s), which concludes the proof. $\square$

The result can equivalently be shown by assuming by contradiction that the chain with the minimal number of loops has at least one loop.

Now, we show that the structure of the chains of a constraint sequence characterise its feasibility.

Lemma 4.41 *A consistent constraint sequence κ is feasible if and only if*

- (i') κ has no infinitely decreasing two-way chain, and
- (ii') Ceiled chains have a uniformly bounded depth, i.e. there exists a bound $B \in \mathbb{N}$ such that increasing two-way chains of κ that are ceiled have depth at most B .

Items are primed as we later on refine this characterisation, see Lemma 4.42

Proof idea. The left-to-right direction is easy: if item (i') is not satisfied, then one needs infinitely many values below the maximal initial value of a register to make the constraint sequence feasible, which is impossible in $\mathbb{N}$. Likewise, if item (ii') is not satisfied, then one also needs infinitely many values below the value of a maximal stable chain, which is again impossible.

For the other direction, we show that if both items hold, then one can construct a sequence of valuations $v_0 v_1 \dots$ satisfying the constraint sequence such that for all $r \in R$, $v_i(r)$ is the largest depth of a (decreasing) two-way chain starting in r at moment i . $\square$

Proof. Direction $\Rightarrow$: first, assume that a constraint sequence $\kappa = \tau_0 \tau_1 \dots$ is feasible by a valuation sequence $v_0 v_1 \dots$. Aiming for a contradiction, suppose that item (i') does not hold, i.e. there is an infinitely decreasing two-way chain $\gamma = (r_0, m_0) \bowtie_0 (r_1, m_1) \dots$. Then, for all $i \geq 0$, we have that $v_i(r_i) \bowtie_i v_{i+1}(r_{i+1})$. Since there are infinitely many $\bowtie_i$ that are equal to $'>'$, this yields an infinite descending sequence in $\mathbb{N}$, which yields a contradiction since $\mathbb{N}$ is well-founded. Now, again towards a contradiction, assume the negation of item (ii'), i.e. that there is a sequence of two-way ceiled chains of unbounded depth. By definition of a ceiled chain, the constraint sequence κ has a maximal stable chain. Let M be the value of the registers in the maximal stable chain (all such registers have the same value, by definition of a stable chain). Take a ceiled chain of depth

We say 'infinite descending sequence' for 'infinite descending chain', which is the usual terminology in the context of well-founded orders, to avoid a clash with our notion of chain, that concern registers. As witnessed by this proof, both notions are closely related.

at least $M + 1$. Necessarily, one of its registers has a value that exceeds M , which yields a contradiction.

Direction $\Leftarrow$: let $\kappa = \tau_0 \text{constraint}_1 \dots$ be a consistent constraint sequence that satisfies items (i') and (ii'). Let us build a sequence of register valuations $v_0 v_1 \dots$ such that for all $i \geq 0$, $v_i \sqcup v'_{i+1} \models \text{constraint}_i$. Let $r \in R$ be a register, and $i \geq 0$. Let $\delta(r, i)$ be the largest depth of two-way chains that start from (r, i) . Note that $\delta(r, i) < \infty$ by item (i'). Then, define $v_i(r) = \delta(r, i)$.

Recall that v' is defined as $v' : r' \mapsto v(r)$ for all $r \in R$.

Let us show that for all $i \geq 0$, we have that $v_i \sqcup v'_{i+1} \models \text{constraint}_i$ holds, i.e. that all atomic formulas of τ_i are satisfied. Pick an arbitrary atom $r_1 \bowtie r_2$ of τ_i , where $r_1, r_2 \in R \sqcup R'$. Define $m_{r_1} = i + 1$ if r_1 is a primed register, else $m_{r_1} = i$; similarly define m_{r_2} . There are two cases.

- $\bowtie$ is '='. Then, the deepest chains from (r_1, m_{r_1}) and (r_2, m_{r_2}) have the same depth, $\delta(r_1, m_{r_1}) = \delta(r_2, m_{r_2})$, which means that $v_i \sqcup v'_{i+1}$ satisfies the atomic formula.
- $\bowtie$ is '>'. Then, any chain $(r_2, m_{r_2}) \dots$ of depth d starting from (r_2, m_{r_2}) can be prefixed by (r_1, m_{r_1}) to create the chain $(t_1, m_{t_1}) > (t_2, m_{t_2}) \dots$ of depth $d + 1$. Hence $\delta(r_1, m_{r_1}) > \delta(r_2, m_{r_2})$, therefore $v_i \sqcup v'_{i+1}$ satisfies the atomic formula.

This concludes the proof. $\square$

The above lemma characterises feasibility in terms of two-way chains. As we demonstrate, it actually suffices to consider one-way chains, thanks to a Ramsey argument.

Lemma 4.42 *A consistent constraint sequence κ is feasible in $\mathbb{N}$ if and only if*

- (i) *It has no infinitely decreasing one-way chains, and*
- (ii) *Ceiled one-way chains have a uniformly bounded depth, i.e. there exists a bound $B \in \mathbb{N}$ such that increasing one-way chains of κ that are ceiled have depth at most B .*

Proof idea. The left-to-right direction is a direct consequence of Lemma 4.41, since one-way chains form a particular case of two-way chains. For the other direction, we actually show that each item implies its primed version (see Lemma 4.41). First, from an infinitely decreasing two-way chain, we can always extract an infinitely decreasing one-way chain. Indeed, two-way chains can be infinite to the right but not to the left, so for any moment i , there exists a moment $j > i$ such that one register of the chain is smaller at step j than a register of the chain at step i .

Now, given a sequence of ceiled two-way chains of unbounded depth, we need to construct a sequence of one-way chains of unbounded depth. This construction is more difficult than in the case: even though there are deeper and deeper ceiled two-way chains, they may start at later and later moments in the constraint sequence and go to the left, thus one cannot just take an arbitrarily deep two-way chain and extract from it an arbitrarily deep one-way chain. However, we show, using a Ramsey argument, that it is possible to extract arbitrarily deep one-way chains as the two-way chains are not completely independent. $\square$

Proof. Let κ be a consistent constraint sequence. If it is feasible, by Lemma 4.41, we know that it satisfies items (i') and (ii'), which implies that both (i) and (ii), as they respectively weaken their primed version.

Now, assume that κ is not feasible. By Lemma 4.41, it means that either (i') or (ii') is violated.

Extracting an infinitely decreasing one-way chain from a two-way one

First, assume that κ has an infinitely decreasing two-way chain $\gamma = (r_a, i) \dots$, we construct an infinitely decreasing one-way chain γ' . The construction is illustrated in Figure 4.7. Our one-way chain γ' starts in

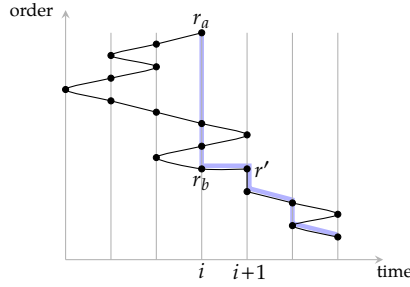


Figure 4.7.: Proving that $\neg(i') \Rightarrow \neg(i)$ to get Lemma 4.42. The two-way chain γ is in black, the constructed one-way chain γ' is in blue.

(r_a, i) . The area on the left from i -timeline contains $i \cdot |R|$ points, but γ has an infinite depth, so at some point it must go to the right from i . Let r_b be the smallest register visited at moment i by γ . We first assume that r_b is different from r_a (the other case is treated later). We have $(r_b, i) \triangleright (r', i+1)$. We append this to γ' and get $\gamma' = (r_a, i) \triangleright (r_b, i) \triangleright (r', i+1)$. If r_a and r_b are equal, the chain γ moved $(r_a, i) \triangleright (r', i+1)$, in which case we only append $(r_a, i) \triangleright (r', i+1)$. By repeating the argument from the point $(r', i+1)$, we add one link to the chain γ' . Thus, by induction on $i \geq 0$, we can construct the infinitely decreasing one-way chain γ' , which means that item (i) is violated.

Extracting arbitrarily deep one-way ceiled chains from two-way ones

Now, assume that item (ii') does not hold. Given a set of ceiled two-way chains of unbounded depth, we need to construct a set of ceiled one-way chains of unbounded depth. This is done with a Ramsey argument: we represent a two-way chain as a clique with colored edges, and whose monochromatic subcliques represent all one-way chains. With Ramsey's theorem [118, Theorem B], we know that a monochromatic subclique of a required size always exists if a clique is large enough, which yields the sought one-way chain. Let $n \geq 0$; we need to exhibit a one-way chain of depth at least n . Let $l = R(3, n)$ be the Ramsey number for 3 colors and target size n . Let γ be a two-way chain of depth l ; it exists since item (ii') fails. From it we construct the following colored clique (the construction is illustrated in Figure 4.8 on the following page).

- Remove stuttering elements from γ : whenever $(r_i, m_i) = (r_{i+1}, m_{i+1})$ appears in γ , remove (r_{i+1}, m_{i+1}) . We repeat this process until no stuttering elements appear. Let $\gamma_{>} = (r_1, m_1) > \dots > (r_l, m_l)$ be the resulting sequence; it is strictly decreasing, and contains l pairs (the same as the depth of the original γ). Note the following property ($\dagger$): for every not necessarily adjacent $(r_i, m_i) > (r_j, m_j)$, there

[118]: Ramsey (1930), 'On a Problem of Formal Logic'

Given a number c of colors and a size n , the Ramsey number $R(c, n)$ denotes the size of a clique needed to ensure the existence of a monochromatic subclique when c colors are used for colouring. It corresponds to the m_0 of Theorem B in [118].

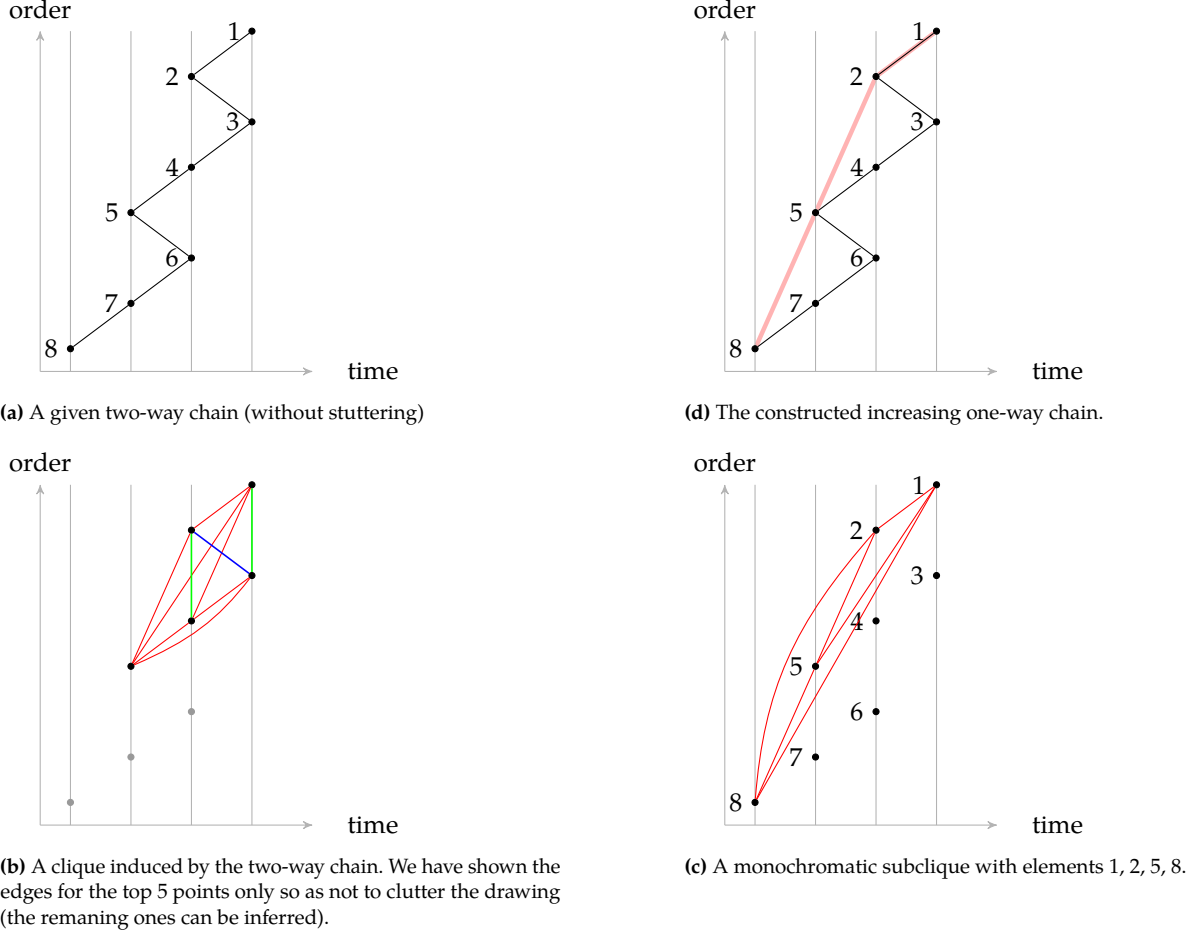


Figure 4.8.: Proving that $\neg(ii') \Rightarrow \neg(ii)$ to get Lemma 4.42.

is a one-way chain $(r_i, m_i) \dots (r_j, m_j)$; it is decreasing if $m_i < m_j$, and increasing otherwise; its depth is at least 1.

- The elements (r, m) of $\gamma_>$ serve as the vertices of the colored clique. The edge-coloring function is: for every $(r_a, m_a) > (r_b, m_b)$ in $\gamma_>$, let $\text{color}((r_a, m_a), (r_b, m_b))$ be $\nearrow$ if $m_a < m_b$, $\searrow$ if $m_a > m_b$, $\downarrow$ if $m_a = m_b$ (see Figure 4.8b).

By applying Ramsey's theorem [118, Theorem B], we get a monochromatic subclique of size n with vertices $V \subseteq \{(r_1, m_1), \dots, (r_l, m_l)\}$. Its color cannot be $\downarrow$ when $n > |R|$, because a time line has at most $|R|$ points. Suppose the subclique color is $\nearrow$ (the case of $\searrow$ is similar). We build the increasing sequence $s = (r_1^*, m_1^*) < \dots < (r_n^*, m_n^*)$, where $m_i^* < m_{i+1}^*$ and $(r_i^*, m_i^*) \in V$, for every plausible i . The sequence s may not be a one-way chain, because the removal of stuttering elements that we performed at the beginning can cause time jumps $m_{i+1} > m_i + 1$. However, relying on property (†), one can construct a one-way chain γ^* of depth n from s by inserting the necessary elements between (r_i, m_i) and (r_{i+1}, m_{i+1}) . Finally, when the subclique has color $\searrow$, the resulting chain is decreasing.

Thus, for every given n , we constructed either a decreasing or increasing ceiled one-way chain of depth n , which means that item (ii) does not hold.

Overall, if κ is a consistent constraint sequence that is not feasible, it

[118]: Ramsey (1930), 'On a Problem of Formal Logic'

either breaks item (i) or (ii), which concludes the proof. $\square$

The above characterisation is then easily lifted to 0-feasibility.

Lemma 4.43 *A consistent constraint sequence $\kappa = \tau_0\tau_1 \dots$ is 0-feasible in $\mathbb{N}$ if and only if:*

- (i) *It has no infinitely decreasing one-way chains,*
- (ii) *Ceiled one-way chains have a uniformly bounded depth, and moreover*
- (iii) *Initially, all registers contain 0, i.e. $\tau_{0|R} = \{r=s \mid r, s \in R\} \cup \{r=0 \mid r \in R\}$, and*
- (iv) *It has no decreasing one-way chain of depth ≥ 1 from any (r_i, m_i) such that τ_i implies $r_i = 0$.*

More precisely, there exists a bound $B \in \mathbb{N}$ such that increasing *one-way* chains of κ that are ceiled have depth at most B .

We now have all the tools in hand to prove the main result of this section, namely that feasible action sequences over $(\mathbb{N}, <, 0)$ are recognised by *max-automata* [117, Section 2] (Theorem 4.46). They consist in an ω -automaton equipped with a finite set of counters $c_1, \dots, c_n$ which can be incremented, reset to 0, or updated by taking the maximal value of two counters. As for ωB -automata, these counters cannot be tested along the run (otherwise we get an undecidable model, since it can simulate Minsky machines). The acceptance condition is given as a Boolean combination of conditions ‘counter c_i is bounded along the run’, i.e. there exists a bound $B \in \mathbb{N}$ such that counter x_i has value at most B during the whole execution. By using negation, one can also expression conditions like ‘ x_i is unbounded along the run’, i.e. it takes arbitrarily large values.

[117]: Bojańczyk (2011), ‘Weak MSO with the unbounding quantifier’

This model strictly lies between ω -automata and non-deterministic ωB -automata. On the one hand, they can recognise the language L_{aB} of , defined as

$$L_{aB} = \{ba^{k_0}ba^{k_1} \dots ba^{k_i}b \dots \mid \exists B \in \mathbb{N}, \forall i \in \mathbb{N}, k_i \leq B\}$$

On the other hand, they cannot recognise [117, Lemma 27]

$$L_{\parallel < \infty} = \{a^{n_1}ba^{n_2}ba^{n_3}b \dots \mid \liminf n_i < \infty\}$$

First, let us show the result for *constraint* sequences.

Proposition 4.44 *For every R , there is a deterministic max-automaton A^R recognising the language of constraint sequences over R that are 0-feasible $\mathbb{N}$.*

Its number of states is exponential in $|R|$, and it has $\mathcal{O}(|R|^2)$ counters.

Proof. We design a deterministic max-automaton that checks conditions (i) to (iv) of Lemma 4.43.

Condition (i), namely the absence of infinitely decreasing one-way chains, is an ω -regular condition: an ω -automaton can guess a chain and verify that it is infinitely decreasing, i.e. that it sees $>$ infinitely often, using a Büchi condition; determinising and complementing provides a deterministic parity automaton for this property.

Checking condition (i) (the absence of ceiled one-way chains of unbounded depth) is more involved. We design a master automaton that tracks every chain γ that currently exhibits a stable behaviour. To every such chain γ , the master automaton assigns a tracer automaton whose task is to ensure the absence of unbounded depth ceiled chains below

γ . To that end, it uses $2|R|$ counters and requires them to be bounded. When two chains arrive in the same point in a constraint, the automaton takes the maximum of their depth (hence the need for a max-automaton, and not just an ω -automaton). The overall acceptance condition ensures that if the chain γ is stable, then there are no ceiled chains below γ of unbounded depth. Since the master automaton tracks *every* such potential chain, we are done.

Condition (iii) (all registers contain 0) is easily shown ω -regular, as well as condition (iv) (no strictly decreasing chains from 0), which simply consists in checking that no ' $>$ ' is seen from a register r such that τ implies $r = 0$.

Since max-automata are able to recognise ω -regular languages and are closed under Boolean operations [117], we can build a deterministic max-automaton that checks all those conditions. $\square$

[117]: Bojańczyk (2011), 'Weak MSO with the unbounding quantifier'

We can now relate action sequences and constraint sequences: an action sequence α over $(\mathbb{N}, <, 0)$ naturally induces a constraint sequence κ_α over $(\mathbb{N}, <, 0)$ which is 0-feasible if and only if α is feasible. For instance, for registers $R = \{r, s\}$, an action sequence starting with $(r < \star \wedge s < \star, \{s\})$ (test whether the input data value d is above the values of r and s and store it in s) induces a constraint sequence starting with $\{r = s, r = 0, r = r', s < s', r' < s'\}$ (the atomic formulas $r = s$ and $r = 0$ result from the fact that all registers are initially equal to 0; we omitted some formulas than can be deducted).

Recall that to lighten the notations, we represent constraints as sets of atomic formulas.

Lemma 4.45 *Let R be a set of registers. There exists a mapping constr from action sequences over R (over data domain $(\mathbb{N}, <, 0)$) to constraint sequences over R (again, over data domain $(\mathbb{N}, <, 0)$) such that for all action sequences α $\text{constr} \alpha$ is 0-feasible if and only if α is feasible.*

Proof. The general construction is a straightforward generalisation of the above example and is consequently omitted. $\square$

With this correspondence, we get that the set of feasible action sequences over $(\mathbb{N}, <, 0)$ is also recognisable by a deterministic max-automaton. Recall that an action sequence is feasible if there exists a data word that is compatible with it. Indeed, to any action sequence α , one can associate a constraint sequence κ_α which is 0-feasible if and only if α is feasible (we do not distinguish between feasibility and 0-feasibility for action sequences since we are only interested in the former):

Theorem 4.46 *For every R , there is a deterministic max-automaton A^R recognising the language $\text{Feasible}_{(\mathbb{N}, <, 0)}(R)$ of action sequences over R that are feasible in $\mathbb{N}$.*

Its number of states is exponential in $|R|$, and it has $\mathcal{O}(|R|^2)$ counters.

By [117, Theorem 5], we know that deterministic max-automata are equivalent to weak MSO with the unbounding quantifier, which consists in Monadic Second Order Logic where quantification over sets is limited to finite sets, equipped with a quantifier U such that $UX \cdot \varphi(X)$ holds whenever $\varphi(X)$ holds for sets X of arbitrary size. Thus, we obtain the following corollary:

[117]: Bojańczyk (2011), 'Weak MSO with the unbounding quantifier'

Corollary 4.47 *For every R , there is a weak MSO+U formula recognising the language $\text{Feasible}_{(\mathbb{N}, <, 0)}(R)$ of action sequences over R that are feasible in $\mathbb{N}$.*

Both results provide a way to abstract the behaviour of register automata over $(\mathbb{N}, <, 0)$. As mentioned earlier, they yield decidability of the synthesis game over specifications given by such automata (see also Remark 7.10).

In [39], the authors establish decidability of the emptiness problem for register automata over $(\mathbb{N}, <, 0)$, by reducing to the emptiness problem of a non-deterministic ω B-automaton. Of course, the argument again works in the case of deterministic max-automaton, so we can apply these results to show:

Theorem 4.48 ([39, Theorem 14]) *The emptiness problem for register automata over $(\mathbb{N}, <, 0)$ is decidable.*

Proof. Let A be a register automaton over $(\mathbb{N}, <, 0)$. As in the proof of Proposition 4.30, we make the product of A with the deterministic max-automaton A^R of Theorem 4.46. Then, such an automaton is empty if and only if A is empty. Emptiness is decidable for deterministic max-automata, which concludes the proof [117, Theorem 3]. $\square$

The reader who is mainly interested in the unbounded synthesis problem over $(\mathbb{N}, <, 0)$ can now jump to the next Chapter 5 before going to Chapter 7, where this study is detailed in Section 7.3.4.

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

[117]: Bojańczyk (2011), ‘Weak MSO with the unbounding quantifier’

4.8. Universal Register Automata

Already over data domains with equality only, the emptiness problem of universal register automata is undecidable, by a direct reduction from the universality problem of non-deterministic register automata (undecidable by [101, Theorem 18]). For this reason, we do not provide a detailed study of the model; the only properties that we use are obtained by duality.

Note that universal register automata over $(\mathbb{D}, =, \mathbb{C})$ can recognise the language of words whose data values are pairwise distinct, $L_{\forall \neq} = \{d_0 d_1 \dots \mid \forall i \neq j, d_i \neq d_j\}$, as witnessed by the register automaton of Figure 4.1b on page 81. Since it does not contain any data word with finitely many distinct data values, it means that they do not have the renaming property (Proposition 4.15).

To get an idea of the expressiveness of this automaton class, let us have a look at its projection over labels. We start by pointing out that it is not ω -regular; as we see next, the situation is much worse.

Example 4.8 Consider the following data language over finite data words:

$$L_{brg} = \left\{ (\text{req}, d_1) \dots (\text{req}, d_n)(\text{grt}, d'_1) \dots (\text{grt}, d'_m) \mid \begin{array}{l} \text{for all } i \neq j, d_i \neq d_j \\ \text{and all } 1 \leq i \leq n, \\ \text{there exists } j \\ \text{such that } d'_j = d_i \end{array} \right\}$$

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

L is defined over finite data words so as not to clutter the argument, but it is directly generalised to data ω -words by appending $\$^\omega$ at the end. The b stands for ‘batch’, as req are sent in a batch.

A data word in L_{brg} consists in a word $w \in \text{req}^n$ with pairwise distinct data followed by a word $w' \in \text{grt}^m$ which contains at least all the data of w . This language can be interpreted as the request-grant specification, restricted to the case where all requests are made first, and are all made by pairwise distinct clients.

L_{brg} is recognised by a universal register automaton which checks that once a label grt is read, only grt are read (this is an ω -regular property) and, on reading (req, d_i) , universally triggers a run which checks that

1. (req, d_i) does not appear again
2. (grt, d_i) appears at least once.

Now, $\text{lab}(L_{brg}) = \{\text{req}^n \text{grt}^m \mid m \geq n\}$, which is not regular. Padding with $\$$ yields a data language that is not ω -regular.

As a consequence, we have:

Property 4.49 *The projection over labels of universal register automata is not always ω -regular.*

Actually, one can define a universal register automaton which recognises encoding of runs of a Turing machine. The following result is trivially obtained from the undecidability of the universality problem for non-deterministic register automata [101, Theorem 18], but what we are interested in is the construction of the automaton. This construction is inspired by the proof of Theorem 1 in [44].

Theorem 4.50 ([101]) *The emptiness problem for universal register automata over $(\mathbb{D}, =, \#)$ is undecidable.*

Proof. Let M be a deterministic Turing machine with Q states. Recall that a configuration of M is a triple $c = (q, w, r) \in Q \times \{0, 1\}^n \times \{0, \dots, n-1\}$ for some $n \in \mathbb{N}$, where q is the current state of M , w is the content of its tape, and r is the position of the head. A configuration c can be encoded as w , that we decorate with q and whose r -th position is marked with a special symbol. Formally, c is encoded as the word x_c of length n over alphabet $Q \times \{0, 1\} \times \{\nabla, \cdot\}$. For all $0 \leq i < n$, we let $x_c[i] = (q, w[i], p)$, where $p = \nabla$ if $i = r$, and $p = \cdot$ otherwise. In the following, it eases the proof to assume that when the length of the tape never exceeds $m \in \mathbb{N}$, configurations all are encoded with words of fixed length m . This is doable for all configurations with tape length $n \leq m$ by allowing a third value ϵ for elements of the tape, and instead encoding $w' = w\epsilon^{m-n}$.

Now, there exists a URA A_M with labels alphabet $Q \times \{0, 1, \epsilon\} \times \{\nabla, \cdot\}$ which recognises a data language L_M whose projection on labels is

$$\text{lab}(L_M) = \left\{ x_{c_0} \$ x_{c_1} \$ \dots \mid \begin{array}{l} \text{there exists } m \in \mathbb{N}, \text{ all } x_{c_i} \text{ have length } m \text{ and} \\ c_0 c_1 \dots \text{ is the sequence of configurations of } M \end{array} \right\}$$

The main ingredient of the proof is to show that there exists a universal register automaton (with two registers) which recognises the data language consisting in infinite repetitions of a finite sequence of data values which are pairwise distinct, separated by $\$$:

$$L_{\$} = \{(d_0 d_1 \dots d_m \$)^\omega \mid m \in \mathbb{N} \wedge \forall 0 \leq i < j \leq m, d_i \neq d_j\}$$

To recognise this language, the URA:

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

[44]: Ehlers, Seshia, and Kress-Gazit (2014), ‘Synthesis with Identifiers’

Note that asking that all c_i have length m implies that the length of the tape never exceeds m along the computation. As we briefly explain at the end of the proof, it is actually not necessary to restrict to computations of bounded length, yet it makes the argument much simpler.

As usual, the $\$$ separator can either be simulated with a label, or by reading an additional data value at the beginning.

- Stores d_0 and triggers a run which checks that d_0 appears immediately after each occurrence of \$.
- Before reading the \$, when it reads a data value d_i , it triggers a run which stores the next data value d_{i+1} and checks that on every occurrence of d_i , the next data value is d_{i+1} . When the next symbol is a \$ (i.e. $i = m$), it instead checks that after every occurrence of d_m , the next symbol is a \$.

Recall that languages recognised by URA are closed under intersection (Proposition 4.3), so we can assume that the automaton already restricts to labelled data words w whose data component $\text{dt}(w)$ is of the form $d_0 \dots d_m \$^\omega$ for some $m \in \mathbb{N}$, where the d_i are pairwise distinct. We further ask that the label component $\text{lab}(w)$ is of the form $x_0 \$ x_1 \$ \dots$, where each x_i has length m ; this is done thanks to the \$ separator. We moreover ask that x_i is a valid encoding: all labels encode the same state q ; the reading head is at exactly one position (i.e. ∇ appears exactly once), and $w \in \{0, 1\}^* \mathcal{E}^*$ (i.e. once a padding symbol $\mathcal{E}$ is read, the automaton only reads padding symbols). This is a regular property.

Now that the input is well-formed, it remains to check that the x_i indeed encode the sequence of configurations of M . First, A_M checks that x_0 encodes the initial configuration $(q^i, \mathcal{E}, 0)$. It thus checks that $x_0 \in (q^i, \mathcal{E}, \nabla)(q^i, \mathcal{E}, \cdot)^*$ (recall that we already know that all x_i have the same length, so we need not evaluate the length of x_0). Then, the automaton has to verify that successive x_i indeed encode successive configurations.

Let $i \in \mathbb{N}$, and assume the configuration encoded by x_i is $c = (p, w, r)$, and that $\delta(p, w[r]) = (q, a, m)$ (we drop the index i for readability). We treat the case $a = 0$ (the machine writes 0 at the reading head position) and $m = \rightarrow$ (the reading head moves to the right); the other cases are similar. Checking that the labels of x_{i+1} all encode the same state q is a regular property. Verifying that the content of the tape is unchanged, except for $w[r]$, is done as follows: the automaton stores the associated data value and memorises the content of the current cell in its state. It then checks that label of the next occurrence of data value indeed corresponds to the same bit. Note that here, it is crucial that the data values associated with each position are pairwise distinct, so that they can be used as a unique identifier. Finally, checking that the machine writes 0 in its current cell and moves one cell to the right is done in a similar way, by locating the position of r using the data value associated with ∇ . If the reading head is already at the rightmost position, the automaton rejects.

As a consequence, $\text{lab}(L_M)$ indeed consists in encodings of the run of M (this run is unique as we assume that M is deterministic), if this run only uses a bounded amount of tape (otherwise $L_M = \emptyset$). By additionally asking that some configuration is accepting, we reduce the emptiness problem of universal register automata to the halting problem for Turing machines. Indeed, if a run is accepting, then in particular it only uses a bounded amount of tape.

Let us conclude the proof by noting that we can actually construct an automaton that recognises the unique encoding without padding of the run of M , whether it uses a bounded amount of tape or not. In our setting, the machine cannot erase the content of its tape, i.e. it cannot write the

symbol $\mathcal{E}$ (this is without loss of generality). As a consequence, the length of the x_i is increasing. We thus need to slightly modify the automaton recognising $L_{\$}$. It proceeds as before, by checking that the presence of d_i implies the presence of d_{i+1} in the next step, but the case of d_m is treated differently when the reading head is at the rightmost position of the tape and moves to the right. Instead of asking that the next symbol is a $\$$, we ask that it is a new data value, distinct from the previous ones, and that the symbol after it is a $\$$. The freshness condition can be enforced by checking that between two $\$$, any data value appears at most once. If the reading head is not at the rightmost position or does not move to the right, the automaton behaves as before. The above construction is correspondingly adapted. $\square$

4.9. Register Automata and Arithmetic

4.9.1. Register Automata over $(\mathbb{N}, +1, 0)$

As hinted at by the register automaton of Figure 4.1d on page 81 in Example 4.3, register automata over $(\mathbb{N}, +1, 0)$ are equivalent to Minsky machines, so they can compute any partial recursive function (modulo a Gödel-like encoding that is complicated but effective) [102, Theorem I], and are thus Turing-complete. This is almost by definition, since the registers play the same role in both models. One simply needs to observe that addition, subtraction and zero-tests can be implemented (addition is $\star = c_i + 1$, subtraction is $\star + 1 = c_i$, and zero-test is $c_i = 0$). As a consequence, we get:

Property 4.51 *Let M be a (deterministic) Minsky machine with counters $\{c_1, \dots, c_k\}$ for some $k \geq 1$. There exists a deterministic register automaton over $(\mathbb{N}, +1, 0)$ of size linear in $|M|$ whose runs are exactly those of M .*

As a consequence, deterministic register automata over $(\mathbb{N}, +1, 0)$ are equivalent to Turing machines when they have at least two registers (again, modulo some encoding). Since the halting problem is undecidable [1, 119, 120], so is the emptiness problem, and we get:

Theorem 4.52 *The emptiness problem for deterministic register automata over $(\mathbb{N}, +1, 0)$ with at least two registers is undecidable.*

4.9.2. Register Automata over $(\mathbb{N}, \times, 2, 3)$

One can implement in a similar fashion multiplication and division by 2 and 3 if the predicate $\times$ (with its expected interpretation) is given, e.g. division by 3 is $\star \times 3 = c_i$, where c_i is some register. By [102, Theorem II], we get that, even with one register, one can compute any partial recursive function. As a consequence,

Theorem 4.53 *The emptiness problem for deterministic register automata over $(\mathbb{N}, \times, 2, 3)$ is undecidable, already with one register.*

Remark 4.21 We picked 2 and 3 as they are minimal, but this also works with any two distinct prime numbers.

[102]: Minsky (1961), 'Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines'

[1, 119]: Turing, Turing (1936, 1938), 'On Computable Numbers, with an Application to the Entscheidungsproblem', 'On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction'. See [120]: Sipser (2013), *Introduction to the Theory of Computation*, and in particular Section 4.2 and the Theorem 5.2 in Section 5.1, for a more detailed presentation.

[102]: Minsky (1961), 'Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines'

4.10. Extensions of the Model

In this section, we describe two (somewhat orthogonal) extensions of the model. The first one is non-deterministic reassignment [38], also known as *guessing*. We make it part of the definition of the transducers of Part II, but it is not included in this part to keep the argument focused. The second one is fresh-register automata [121], and is not extensively studied for the same reason. However, when this is relevant, we make a few remarks to point out if the results still hold when allowing these extensions.

[121]: Tzevelekos (2011), ‘Fresh-register automata’

4.10.1. Non-Deterministic Reassignment, Also Known as Guessing

In [38], the authors introduce *non-deterministic reassignment*, a feature that aims at improving the symmetry of the model. It is also known as *guessing*, and we favour the latter terminology since it also makes sense in the universal semantics. At each step, on top of assigning the input data value to some registers, the automaton can *guess* data values and assign them to some other registers.

Definition 4.35 (Non-Deterministic Reassignment, a.k.a. Guessing) A *register automaton with non-deterministic reassignment*, or *register automaton with guessing*, is the same as a register automaton, except that its transitions are now of the form $t = p \xrightarrow{\sigma, \phi, \text{asgn}, \text{ndasgn}} q$, where $\text{ndasgn} \subseteq R$ and $\text{asgn} \cap \text{ndasgn} = \emptyset$, thus Δ is of type $\Delta \subset_f Q \times \Sigma \times \text{Tests}_{\mathbb{D}}(R) \times 2^R \times 2^R \times Q$. The successor relation over configurations is modified as follows. Given two configurations (p, ν) and (q, λ) , a transition t as above and a data value $d \in \mathbb{D}$, we say that C' is a *successor configuration* of C whenever:

In the context of register automata with guessing, we call *non-guessing register automata* those that do not have this feature, as defined in Definition 4.13.

- $\nu, d \models \phi$ (as for non-guessing register automata)
- Registers are assigned as follows. For all $r \in R$:
 - if $r \in \text{asgn}$, $\lambda(r) = d$ (as for non-guessing register automata)
 - if $r \in \text{ndasgn}$, then $\lambda(r)$ is not constrained and can take any data value (the register automaton ‘guesses’ the value of the register)
 - otherwise, $\lambda(r) = \nu(r)$ (as for non-guessing register automata)

The notions of (partial, initial, final, accepting) run are defined in the expected way. Register automata with guessing can also be defined for both the non-deterministic and universal semantics. Again, a data word is accepted by a non-deterministic register automaton with guessing if there exists an accepting run over it. It is accepted by a universal register automaton with guessing if all runs of the automaton are accepting (for all possible guesses; cf Example 4.10).

Guessing is, in essence, not deterministic, and the deterministic restriction of register automata with guessing is defined as (non-guessing) deterministic register automata.

Remark 4.22 The transitions can equivalently be represented as $p \xrightarrow{\sigma, \phi, \text{regOp}} q$, where $\text{regOp} : R \rightarrow \{\text{keep}, \text{set}, \text{guess}\}$. Intuitively, *keep* means that the content of the register is left unchanged, *set* means that it is assigned,

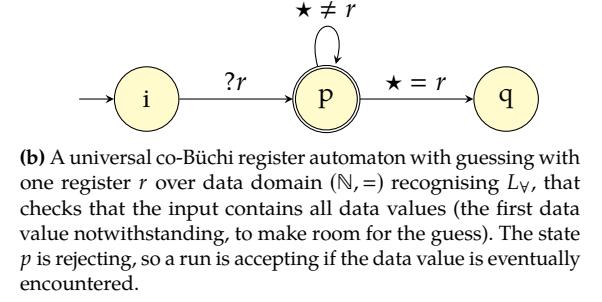
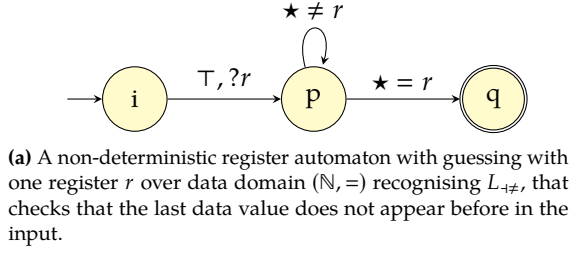


Figure 4.9.: Two register automata with guessing, that respectively recognise $L_{\neq}$ and $L_{\forall}$. $?r$ graphically denotes $\text{ndasgn} = \{r\}$, and the asgn are omitted as they are empty. Recall that T is a test which is satisfied by any data value.

and guess that it is guessed. We adopt the latter formalism in Part II. In this part, we prefer to stay as close as possible to Definition 4.13.

This attribute strictly increases expressiveness, already over data domains with equality only.

Example 4.9 ([38, Example 4]) Consider the language of finite data words whose last symbol is distinct from all the others. Formally,

$$L_{\neq} = \{d_0 \dots d_n \mid \forall 0 \leq i < n, d_i \neq d_n\}$$

This language cannot be recognised by any non-deterministic register automaton without guessing [38, Example 4]. However, it is recognised the non-deterministic register automaton with guessing of Figure 4.9a. It initially guesses the last data value, and then waits in p until it sees it again, then transitions to q . If it was indeed the last data value, it accepts, otherwise it rejects on the next step since there is no transition from q . In [21, Example 4], it is shown that $L_{\neq}$ cannot be recognised by a register automaton. Intuitively, since the automaton does not know what will be its last data value, it would have to store all previous data values to check that they are distinct from this last one.

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

Example 4.10 For universal register automata, a data word is accepted if it is accepted by all runs, for all possible guesses. As witnessed by the automaton of Figure 4.9b, this allows to recognise data words which contain all data values

$$L_{\forall} = \{d_0 d_1 \dots \mid \forall d \in \mathbb{D}, \exists i \geq 0, d_i = d\}$$

For simplicity, we considered the variant that excludes the first data value, $\{d_0 d_1 \dots \mid \forall d \in \mathbb{D}, \exists i \geq 1, d_i = d\}$, as in our syntax, the guess can only be verified a posteriori. By using an additional register that contains the first data value d_0 to check that the guessed data value is distinct from d_0 , the automaton can be tweaked to recognise $L_{\forall}$.

Closure Properties

Let us describe a few properties of this extension, that help understand how it behaves.

Union and Intersection As pointed out in [38, Theorem 16], the constructions of Proposition 4.2 also work in the presence of guessing (see

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

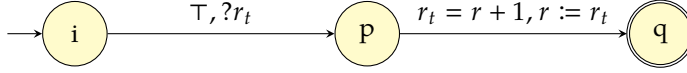


Figure 4.10.: A non-deterministic register automaton with guessing with two registers r and r_t over data domain $(\mathbb{N}, +1, 0)$ that silently increments r . $r := r_t$ consists in copying the content of r_t into r .

also [122, Section 2.5]). The choice of the data domain actually plays no role in the proof, so the result again holds for arbitrary data domains:

Proposition 4.54 (Union and Intersection of non-deterministic register automata with guessing) *Let A_1, A_2 be two non-deterministic register automata with guessing over the same data domain $\mathcal{D}$, with respectively n_1 and n_2 states and k_1 and k_2 registers, and with acceptance condition Ω_1 and Ω_2 . Then:*

- $L(A_1) \cup L(A_2)$ is recognised by a non-deterministic register automaton with guessing with $n_1 + n_2$ states and $\max(k_1, k_2)$ registers, and with acceptance condition $\Omega_1 \cup \Omega_2$.
- $L(A_1) \cap L(A_2)$ is recognised by a non-deterministic register automaton with guessing with $n_1 \cdot n_2$ states and $k_1 + k_2$ registers, and with acceptance condition $\pi_1^{-1}(\Omega_1) \cap \pi_2^{-1}(\Omega_2)$.

As for (non-guessing) register automata, the dual result holds in the universal semantics (see Proposition 4.3).

Closure under automorphisms Closure under automorphisms again holds in the presence of guessing.

Proposition 4.55 *Let A be a non-deterministic register automaton with guessing over data domain $\mathcal{D}$ recognising a data language $L \subseteq (\Sigma \times \mathbb{D})^\omega$, and let $\mu \in \text{Aut}(\mathcal{D})$. Then, $\mu(L) = L$.*

In other words, for all labelled data words $w \in (\Sigma \times \mathbb{D})^\omega$, $w \in L \Leftrightarrow \mu(w) \in L$.

Proof. We redo the proof in Part II, as it is at the center of the study (see Proposition 12.13), but the argument is essentially the same as for non-guessing register automata, and relies on observing that the successor relation over configurations is closed under automorphisms. $\square$

Restriction to a finite set of data values Contrary to non-guessing register automata (Proposition 4.1), it can happen that the restriction to a finite set of data values is not ω -regular, for some data domains. In particular, over $(\mathbb{N}, +1, 0)$, one can construct a register automaton that simply ignores its input, and simulates a Minsky machine using guessing. We explain how to implement increment of register r : the automaton guesses some data value, stores it in a temporary register and checks on the next step that it is indeed $v(r) + 1$. This is graphically depicted in Figure 4.10. Note that this register automaton uses reassignments, but this can be removed by adding information to the states, as in Proposition 4.6. As a consequence, we get that:

Property 4.56 *There exists a register automaton with guessing A (with unary label alphabet) such that $L(A) \cap \{0, 1\}^\omega$ is not ω -regular.*

Proof. Using the construction of Figure 4.10, one can build a register automaton with guessing over finite data words with data domain

[122]: Zeitlin (2006), ‘Look-ahead finite-memory automata’. Register automata with guessing are called ‘look-ahead finite-memory automata’ in this work.

$(\mathbb{N}, +1, 0)$ such that $L(A) \cap \{0, 1\}^* = \{0^n 1^n \mid n \geq 0\}$. It can then be padded to get $\{0^n 1^n 0^\omega \mid n \geq 0\}$, which is not ω -regular.

Such an automaton has two registers r and r_t . First, note that it can check whether it reads 0 or 1 with respective tests $\varphi_0 := \star = 0$ and $\varphi_1 := \star = 0 + 1$. In the following, we write a for 0 and b for 1 when talking about input data values so as not to confuse them with the valuations of the registers, even though internally the machine only manipulates integers. Initially, r and r_t contain 0. Then, on reading an a , it increments its register r as is done in Figure 4.10. At some point, it starts reading some b . It then decrements r , exactly in the same way. When r reaches 0 again (this can be tested by asking whether $r = 0$ on the transitions), it asks to read only b next, and rejects otherwise.

One can generalise the construction to simulate any Minsky machine. We do not formalise it as this property is only mentioned to illustrate the difficulties that might arise when adding guessing. $\square$

Remark 4.23 Of course, if we additionally restrict the register valuations to take their values in a finite set of data values, the above construction breaks and we recover the property that register automata with guessing behave in a regular way. Indeed, then, the number of possible configurations is finite, so the construction of Proposition 4.1 can be generalised.

Fortunately, over data domain $(\mathbb{D}, =, \mathbb{C})$, register automata with guessing again behave like finite-state automata when restricted to a finite set of data values:

Proposition 4.57 ([38, Proposition 9]) *Let A be a register automaton with guessing over data domain $(\mathbb{D}, =)$ with label alphabet Σ . For all finite subsets $A \subset_f \mathbb{D}$, $L(A) \cap (\Sigma \times A)^\omega$ is ω -regular.*

The result is initially for finite data words, but it is easily extended to data ω -words. We also added labels; this poses no difficulty.

Projection over Labels Already without guessing, projection over labels is not always ω -regular, e.g. for data domain $(\mathbb{N}, <, 0)$ (see Section 4.4.2). However, this is the case over data domain $(\mathbb{D}, =, \mathbb{C})$ (Proposition 4.58) and $(\mathbb{Q}, <, 0)$ (Proposition 4.59). The first result is a consequence of the fact that register automata with guessing over data domains with equality enjoy the renaming property [38, Proposition 14]. As they behave like ω -automata when executed over a finite set of data values, the same reasoning as in the non-guessing case (see Proposition 4.19) applies, and we get:

Proposition 4.58 *Let A be a non-deterministic register automaton with guessing. Then, $\text{lab}(L(A))$ is ω -regular.*

More precisely, it is recognised by a non-deterministic ω -automaton A^{lab} with $n(k + \gamma + 1)^k$ states, where n is the number of states of A , k its number of registers, and γ the number of constants. Its acceptance condition is $\pi_1^{-1}(\Omega)$, where Ω is the acceptance condition of A .

Over $(\mathbb{Q}, <, 0)$, constraints actually allow to abstract non-deterministic register automata with guessing, so we can generalise Proposition 4.35 to:

Proposition 4.59 *The projection over labels of a non-deterministic register automaton with guessing over $(\mathbb{Q}, <, 0)$ is ω -regular.*

More precisely, it is recognised by a non-deterministic ω -automaton A^{lab} with states $Q \times \text{Constr}_{\mathbb{Q}}(R)$ and acceptance condition $\pi_1^{-1}(Q)$.

Proof. The same construction as for Proposition 4.35 works for non-deterministic register automata with guessing, since the successor relation for configurations can again be expressed by a first-order formula (cf Proposition 4.34) in the presence of guessing.

We do not redo the proof as the result is not central to this part, and can also be obtained from Part II which natively includes guessing in the definitions. Indeed, it is a consequence of Theorem 12.24, as $(\mathbb{Q}, <, 0)$ is oligomorphic and polynomially decidable (see Corollary 12.25). $\square$

Remark 4.24 (Closure under reversal) One of the motivations for the extension of register automata with a guessing mechanism is to get a model that is closed under reversal, as is the case for finite-state automata. For instance, $L_{\neq}$ is the reversal (see Proposition 4.60) of the data language $L_{\neq} = \{d_0 \dots d_n \mid \forall 1 \leq i \leq n, d_i \neq d_0\}$, which is recognisable by a deterministic register automaton. Indeed, it is the complement of L_{first} of Example 4.1 (restricted to finite data words), which is recognised by the deterministic register automaton of Figure 4.1a on page 81. This is not by accident:

Proposition 4.60 (Closure under reversal [38, Corollary 31]) *Given a finite word $w = a_0 a_1 \dots a_{n-1} a_n \in A^*$ (for some possibly infinite set A) its reversal is the word $\tilde{w} = a_n a_{n-1} \dots a_1 a_0 \in A^*$. The notion is extended to languages by letting, for any $L \subseteq A^*$, $\tilde{L} = \{\tilde{w} \mid w \in L\}$.*

If L is recognised by a register automaton with non-deterministic reassignment, so is $\tilde{L}$.

We do not use this result in the following, but it highlights the gain of symmetry induced by non-deterministic reassignment.

Correspondence with other formalisms Languages recognised by register automata with non-deterministic reassignment also correspond to those generated by regular grammars over infinite alphabets [122, Theorem 8], and have an equivalent description by regular expressions [38, Theorem 30]. Finally, note that these automata are equivalent to G -automata over $(\mathbb{D}, =)$ that consists in an extension of finite-state automata to nominal sets [25, Theorem 6.4]. For these reasons, we allow it in our definition of asynchronous transducers in Part II, that has a more theoretical aim.

However, in this part, we do not want to make it central to the study and favour the model *without* non-deterministic reassignment, whose behaviour is arguably more intuitive. Still, we make a few remarks along the document to point out that our results still hold over $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$ when non-deterministic reassignment is allowed, resorting to properties of the model that are established in [38]. The original model is defined over finite data words, but the properties easily extend to infinite data words.

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

[25]: Bojańczyk, Klin, and Lasota (2014), ‘Automata theory in nominal sets’. They are simply called ‘finite memory automata’ in the paper and the presence of guessing is not made explicit, but the semantics indeed correspond to register automata *with* non-deterministic reassignment.

4.10.2. Fresh-Register Automata

With the non-deterministic semantics, register automata cannot distinguish between a data value that is locally fresh, i.e. that is not present in the current register valuation, and one that is globally fresh, i.e. that never appeared previously in the input (cf Example 4.9). In [121], Tzevelekos extend finite-memory automaton to fresh-register automata, that have the ability to determine whether a data value is globally fresh.

[121]: Tzevelekos (2011), ‘Fresh-register automata’

The predicate $\otimes$ Formally, the notion of test is extended with a special unary predicate $\otimes$ that holds whenever its argument is a globally fresh data value. A *fresh-register automaton* is then a register automaton whose transition relation is instead of type $\Delta \subset_f Q \times \Sigma \times \text{Tests}_{\mathcal{D}}^{\otimes}(R) \times 2^R \times Q$, where $\text{Tests}_{\mathcal{D}}^{\otimes}(R)$ consist in tests over $\mathcal{D}$ where the special predicate $\otimes$ is allowed.

We use the notation $\otimes$ of [121].

Extending configurations with histories Since global freshness is not a local property, one cannot define its semantics from a configuration (q, ν) only. Thus, the notion of configuration is extended to contain an *history*, which is meant to be the set of data values that already appeared in the input. In other words, a *configuration* is now a triple (q, ν, H) , where (q, ν) consists in a state and a register valuation, as before, and $H \subseteq \mathbb{D}$. Correspondingly, satisfiability is defined for a test and a *pair* (ν, H) , where, for all atomic formulas φ that are not $\otimes$, we say that $(\nu, H) \models \varphi$ whenever $\nu \models \varphi$ in the sense of Definition 4.7. Then, for a valuation $\nu : X \rightarrow \mathbb{D}$ (for some finite set X of variables) and a variable $x \in X$ we say that $(\nu, H) \models \otimes(x)$ whenever $\nu(x) \notin H$. We write $\nu, H, d \models \phi$ for $(\nu\{\star \leftarrow d\}, H) \models \phi$.

The successor relation over extended configurations The successor relation for configurations is defined as follows: given two configurations (p, ν, H) and (q, λ, H') , a transition $p \xrightarrow{\sigma, \phi, \text{asgn}} q$ and a data value $d \in \mathbb{D}$, we say that (q, λ, H') is a *successor* of (p, ν, H) on reading the labelled data (σ, d) and following transition t when:

- ▶ d satisfies the test in the current configuration: $\nu, h, d \models \phi$
- ▶ λ is updated according to asgn , i.e. $\lambda = \nu\{\text{asgn} \leftarrow d\}$ (as for usual register automata)
- ▶ The history is extended with d , i.e. $H' = H \cup \{d\}$

Note that we implicitly require that labels match, as for (non-fresh)register automata.

The notions of (partial, initial, final, accepting) run are defined in the expected way. Once more, fresh-register automata can be endowed with the non-deterministic and universal semantics; we leave out the definitions. A fresh-register automaton A is *deterministic* if for all reachable configurations (p, ν, H) and all labelled data values $(\sigma, d) \in \Sigma \times \mathbb{D}$, there exists at most one transition t such that $(p, \nu, H) \xrightarrow[t]{d} (q, \nu', H')$.

Example 4.11 Consider again the data language $L_{\forall \neq}$ of data words that contain pairwise distinct data values, defined as $L_{\forall \neq} = \{d_0 d_1 \cdots \mid \forall i \neq j, d_i \neq d_j\}$ in Example 4.1, and recognised by the *universal* register automaton of Figure 4.1b on page 81. We know that it is not recognisable by

any non-deterministic register automaton (Property 4.18). It is recognised by the deterministic fresh-register automaton of Figure 4.11.

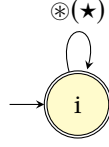


Figure 4.11. A fresh-register automaton with one register r that recognises the language $L_{\forall \neq}$ of data words with pairwise distinct data values.

$\otimes(\star)$ holds whenever $\star$ is globally fresh.

Remark 4.25 Note that the fresh predicate can easily be simulated by the universal register automaton with one register of Figure 4.12, by adapting that of Figure 4.1b on page 81. For this reason, we do not study the fresh extension of universal register automata.

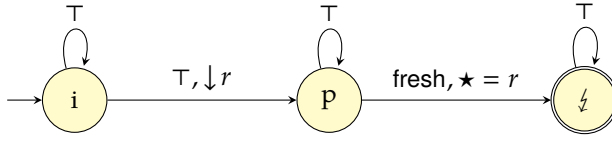


Figure 4.12. A universal register automaton with one register r that simulates the $\otimes$ predicate.

Here, $\top$ means that no test is conducted, and additionally that the label does not matter. We wrote *fresh* for $\otimes$, to make it clear that we do not use the predicate: this is only a label. ζ is a sink rejecting state.

Let us explain how our definition indeed corresponds to the model of [121, Definition 1]. Using the construction of Section 4.4.4 that consists in writing tests in full disjunctive normal form, one can see that over $(\mathbb{D}, =)$, with a unary alphabet of labels, it is possible to restrict the tests to the following syntax:

- $p \xrightarrow{\phi_E(\star), \text{asgn}} q$ (E is exactly the set of registers that contain $\star$)
- $p \xrightarrow{\text{locFresh}(\star), \text{asgn}} q$ (the input data value is locally fresh)
- $p \xrightarrow{\otimes(\star), \text{asgn}} q$ (the input data value is globally fresh)

This corresponds to the syntax of transitions of Fresh Register Automata with multiple assignment as defined in [121, Definition 18], that are equivalent to fresh-register automata (in their sense, i.e. with single assignment) by using Lemma 19, 20 and 21.

Closure Properties The constructions of Proposition 4.2 can be adapted to the case of fresh-register automata to show that they are closed under union and intersection. The proof is spelled out in the case of $(\mathbb{D}, =)$ in [121, Proposition 22], and can be generalised.

Closure under automorphisms again holds, since the successor relation of configurations is closed under automorphisms (see [121, Proposition 6] for the case of $(\mathbb{D}, =)$).

Restriction to a finite set of data values First, it is straightforward to extend the construction of Proposition 4.1 to show that fresh-register automata behave like ω -automata when restricted to a finite alphabet, over any data domain. Again, there is a finite set of configurations; it suffices to additionally keep in memory the history of letters that already appeared in the input. Thus, we get

Recall that ϕ_E is defined in Notation 4.23 as $\bigwedge_{r \in E} \star = r \wedge \bigwedge_{r \notin E} \star \neq r$.

locFresh is defined in Notation 4.17 as $\bigwedge_{r \in R} \star \neq r$. Note that it is a particular case of ϕ_E , by taking $E = \emptyset$. We keep them separate as they play a different role.

This property contrasts with the non-deterministic reassignment extension, where some register automata do not behave like ω -automata when restricted to a finite input alphabet see Property 4.56

Proposition 4.61 *Let A be a fresh-register automaton over data domain $\mathcal{D}$. For any finite subset $X \subseteq \mathcal{D}$, $L(A) \cap (\Sigma \times X)^\omega$ is ω -regular.*

More precisely, $L(A) \cap (\Sigma \times X)^\omega$ is recognised by an ω -automaton A_X with $n(s+1)^k 2^k$ states, where n is the number of states of A , k its number of registers, and $s = |X|$ is the size of X . Moreover, if A is deterministic, so is A_X .

The 2^k factor comes from the fact that we additionally need to remember the history, which can be any subset of X .

Projection over labels In the proof of Proposition 24, the authors observe that a fresh-register automaton (in their sense) is empty if and only if it is empty, when interpreted syntactically. As a consequence, we get that their projection over labels of fresh-register automata with the equality predicate is ω -regular.

Proposition 4.62 *Let A be a fresh-register automaton over $(\mathcal{D}, =, \mathcal{C})$. $\text{lab}(A)$ is ω -regular.*

Synthesis Problems over Data Words

5.

The goal of this thesis is to generalise the study of reactive synthesis problems to the case of infinite alphabets. In this chapter, we describe how to extend existing notions in the finite alphabet case to this setting. First, the notion of synchronous program has to be adapted to the case of programs that manipulate data words, which implies to work modulo an encoding, since the tape alphabet of a Turing machine is necessarily finite. Second, we explain how to represent data word specifications with register automata. This consists in transferring the notion of specification automaton: a specification register automaton alternately reads an input and an output labelled data value, and accepts if and only if the sequence of outputs form an acceptable behaviour for the corresponding sequence of inputs. Then, we consider register transducers as the analogue of transducers in the data word case: they generalise transducers in the same way as register automata extend finite-state automata. Finally, we define the Church synthesis problem over data words as the problem of synthesising a register transducer from a data word specification. We can naturally associate a game with it, that straightforwardly extends the Church game that models the finite alphabet case (cf Section 3.5.3).

Summary

5.1. Reactive Synthesis over Data Words	130
5.2. Data Automatic Specifications	131
5.3. Register Transducers	132
5.3.1. I/O Action Sequences	134
5.4. The Church Synthesis Problem	136

5.1. Reactive Synthesis over Data Words

Recall the reactive synthesis problem (Problem 3.2):

Problem 3.2: REACTIVE SYNTHESIS PROBLEM

- Input:** A specification $S \subseteq I^\omega \times O^\omega$
Output: A synchronous program Q which uniformises S
(i.e. $f_Q \subseteq S$) if it exists
No otherwise

We need to extend the notion of synchronous program to the case of a program which operates over data values. Intuitively, this is the same notion, and we simply assume that data can be encoded in $\mathbb{O}(1)$. Formally, let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathbb{C})$ be a data domain. An *encoding* of $\mathcal{D}$ is an injective function $\text{enc} : \mathbb{D} \rightarrow \{0, 1\}^*$, along with, for all $R \in \mathcal{R}$, a program V_R such that for all $(t_1, \dots, t_k) \in \mathbb{D}^k$ (where k is the arity of R), V_R outputs 1 on input $\text{enc}(t_1)\# \dots \# \text{enc}(t_k)$ whenever $R(t_1, \dots, t_k)$ holds in $\mathcal{D}$ (and 0

Notions related to encoding are explored in more detail in Section 11.1.1 in Part II.

otherwise). Note that the existence of such an encoding implies that $\mathbb{D}$ is countable. Then, a function $f : \mathbb{D}^* \rightarrow \mathbb{D}^*$ is computable when there exists a program $P_f : (\{0, 1\}^* \#)^* \rightarrow \{0, 1\}^*$ such that for all $w = d_0 \dots d_n \in \mathbb{D}^*$, $P_f(\text{enc}(d_0)\# \dots \# \text{enc}(d_n)) = \text{enc}(f(w))$.

Finally, a synchronous program over data words is simply the ω -behaviour of some program over data words.

Over data words, the reactive synthesis problem can be stated as follows:

Problem 5.1: REACTIVE SYNTHESIS PROBLEM OVER DATA WORDS

- Input:** A specification $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$
Output: A synchronous program over data words P computing
 a (causal) function $f_P : (\Sigma \times \mathbb{D})^\omega \rightarrow (\Gamma \times \mathbb{D})^\omega$
 which realises S (i.e. $f_P \subseteq S$) if it exists
 no otherwise

The Church synthesis problem is a specialisation of the reactive synthesis problem to the case of MSO specifications, where the target implementations are computable by finite-state machines. Both the specification formalism and the notion of finite-state computability need to be generalised to the setting of infinite alphabets. As explained in [103], in this setting, compromises have to be made if one wants to preserve decidability.

Over data words, satisfiability of Monadic Second-Order logic is undecidable, even with the equality predicate only. Actually, this is already the case of First-Order logic, as soon as three variables are allowed [123, Proposition 28]. More generally, there does not yet exist a logic which is suitable for expressing specifications, either by lack of expressive power (in particular, closure under negation is often missing) or because the satisfiability problem is already undecidable, which implies that there is no hope for synthesis. It is not yet clear whether the search for a suitable logic will ever be bountiful [124, 125].

Unlike the finite alphabet case, there is no known correspondence between logics and automata which can be used for synthesis purposes. As a first step, we study the Church synthesis problem for specifications expressed as register automata, that we call data automatic. They are the analogue of automatic specifications, as defined over finite alphabets (cf Section 3.4.3). Moreover, already over $(\mathbb{D}, =, \#)$, most register automata models are inequivalent [26, Section 1.4]. In this work, we focus on one-way models, since two-way models have undecidable emptiness and universality problems, already in the deterministic case [101, Theorem 5.3]. This implies that synthesis is undecidable as well. We thus study the Church synthesis problem for specifications expressed as non-deterministic, universal and deterministic register automata.

5.2. Data Automatic Specifications

As in the finite alphabet case (see Section 3.4.3), relations over data words can be encoded as languages, a pair $(w_i, w_o) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ being encoded as $w_i[0]w_o[0]w_i[1]w_o[1] \dots \in ((\Sigma \times \mathbb{D})(\Gamma \times \mathbb{D}))^\omega$.

[103]: Björklund and Schwentick (2010), ‘On notions of regularity for data languages’

[123]: Bojanczyk et al. (2011), ‘Two-variable logic on data words’

[124]: Benedikt, Ley, and Puppis (2010), ‘Automata vs. Logics on Data Words’

[125]: D’Antoni (2012), ‘In the Maze of Data Languages’

In [108]: Bojańczyk and Stefanski (2020), ‘Single-Use Automata and Transducers for Infinite Alphabets’, the authors established that rigidly guarded MSO logic is equivalent with single-use register automata, which form a strict subclass of non-deterministic register automata. However, this class is not expressive enough for our synthesis purpose. Conversely, pebble automata lie between FO and MSO, but they are too expressive since emptiness and universality are undecidable, as shown in [101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’ (Theorem 5.4).

[26]: Bojańczyk (2019), *Atom Book*

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

Correspondingly, a register automaton which alternately reads labelled data from $(\Sigma \times \mathbb{D})$ and $(\Gamma \times \mathbb{D})$ can be seen as recognising the relation

$$\{(w_i, w_o) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \mid w_i[0]w_o[0]w_i[1]w_o[1] \in L(A)\}$$

We can assume without loss of generality that such an automaton is partitioned according to inputs and outputs.

Definition 5.1 Formally, a *specification register automaton* is a register automaton $A = (Q, I, \Sigma, R, \Delta, \Omega)$ where

- ▶ $Q = Q^i \sqcup Q^o$
- ▶ $I \subseteq Q^i$
- ▶ $\Sigma = \Sigma^i \cup \Sigma^o$
- ▶ $\Delta = \Delta^i \sqcup \Delta^o$, with $\Delta^\sigma \subseteq Q^\sigma \times \Sigma^\sigma \times \text{Actions}_{\mathcal{D}}(R) \times Q^{\bar{\sigma}}$
- ▶ $\Omega \subseteq (Q^i Q^o)^\omega$

Where, for $\sigma \in \{i, o\}$, $\bar{i} = o$ and $\bar{o} = i$.

Relation recognised by specification register automata are called *data-automatic relations*, as they are an analogue of automatic relations over data words.

Example 5.1 (Request-Grant) Consider, once again, the setting of a server, granting requests from a set C of clients. We now want to distinguish between the clients; e.g., when a client $id \in C$ makes a requests, the server has to specifically grant this request, by marking it with the client identifier. Here, we do not assume a priori a bound on the number of clients, so the specification can be expressed as $S_{rg} \subseteq I^\omega \times O^\omega$, where $I = \{\text{req}, \text{idle}\} \times \mathbb{D}$ and $O = \{\text{grt}, \text{idle}\} \times \mathbb{D}$, where $\mathbb{D}$ is some countably infinite set (e.g., $\mathbb{N}$).

If you come back to the coffee machine example (Example 3.4), you could also think of the set C as a set of commands; it then allows to model the fact that if one ordered an espresso, they will be disappointed if they receive a long coffee instead.

$$S_{rg} = \left\{ (i, o) \in I^\omega \times O^\omega \mid \text{for all } j \in \mathbb{N}, \begin{array}{l} \text{if } \text{lab}(i[j]) = \text{req} \text{ then} \\ \text{there exists } k \geq j, \\ \text{lab}(o[k]) = \text{grt} \text{ and} \\ \text{dt}(o[k]) = \text{dt}(i[j]) \end{array} \right\}$$

The ω -automaton of Figure 3.8 in Example 3.19 can be equipped with one register to recognise the above specification. One can check that a

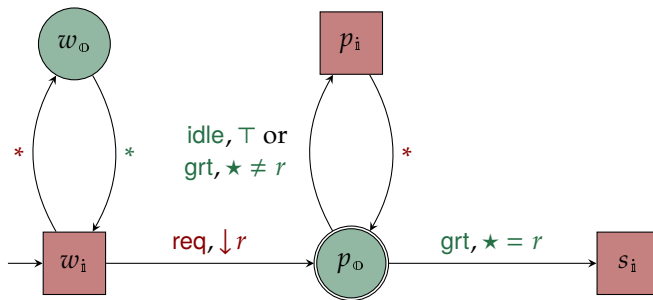


Figure 5.1: A universal co-Büchi register automaton recognising S_{rg} over data words. Recall that $\downarrow r$ stands for $\text{asgn} = \{r\}$, $\star$ is the data value and $*$ accepts any input (or output). The initial state is w_i , and p_o is rejecting.

run is rejecting whenever at some point there is some request from client $id \in C$ but no corresponding grant with identifier id is met afterwards.

5.3. Register Transducers

Correspondingly, for target implementations, we equip synchronous sequential transducers with registers. As for the automaton model, it can use its registers to store data values and conduct tests with regard to

the predicates of the data domain. It is additionally equipped with the ability to output the content of one of its registers, along with an output label.

Definition 5.2 (Synchronous Sequential Register Transducer) A *synchronous sequential register transducer* over data domain $\mathbb{D}$ is a tuple $T = (Q, q^i, \Sigma, \Gamma, R, \delta)$, where:

- Q is a finite set of *states*
- $q^i \in Q$ is the *initial state*
- Σ (respectively Γ) is the *input* (respectively *output*) label alphabet
- R is a finite set of *registers*
- $\delta : Q \times \Sigma \times \text{FTests}_{\mathbb{D}}(R) \rightarrow 2^R \times \Gamma \times R \times Q$ is the (total) transition function, where $\text{FTests}_{\mathbb{D}}(R) \subset_f \text{Tests}_{\mathbb{D}}(R)$ is a finite subset of tests that is:
 - *deterministic*: for all $\phi \neq \psi \in \text{FTests}_{\mathbb{D}}(R)$ $\phi \wedge \psi$ is not satisfiable
 - *complete*: $\bigvee \text{FTests}_{\mathbb{D}}(R)$ is valid.

Note that this transducer does not have a distinguished set of accepting states: implicitly, all its states are accepting.

A transition $\delta(p, a, \phi) = (\text{asgn}, b, r, q)$ is written $p \xrightarrow[a, \phi | \text{asgn}, b, r]{T} q$.

As usual, T can be omitted when clear from the context.

In this chapter, we simply write *register transducer* to denote synchronous sequential register transducers as no ambiguity arises.

We now define the semantics of these transducers, that derives from that of register automata. Let $T = (Q, q^i, \Sigma, \Gamma, R, \delta)$ be a register transducer. A *configuration* of T is a pair $(p, v) \in Q \times \text{Val}_{\mathbb{D}}(R)$, and we again denote $\text{Confs}(T) = Q \times \text{Val}_{\mathbb{D}}(R)$. Given a configuration (p, v) and a labelled data value $(\sigma, d) \in \Sigma \times \mathbb{D}$, the transition that is *enabled* by (σ, d) is the unique $p \xrightarrow[\sigma, \phi | \text{asgn}, \gamma, r]{\sigma, \phi | \text{asgn}, \gamma, r} q$ such that $v, d \models \phi$. The successor configuration of (p, v) on reading (σ, d) is then (q, λ) , where $\lambda = v \{\text{asgn} \leftarrow d\}$. When transitioning from (p, v) to (q, λ) , the transducer *produces* the output $(\gamma, e) \in \Gamma \times \mathbb{D}$, where $e = \lambda(r)$. Overall, we write $(p, v) \xrightarrow[\gamma, e]{\sigma, d | \gamma, e}_T (q, \lambda)$ whenever T transitions from (p, v) to (q, λ) by taking transition t on reading (σ, d) while producing (γ, e) .

The enabled transition exists and is unique since we syntactically ensured that the transition relation is deterministic and complete.

Then, given an input data word $u \in (\Sigma \times \mathbb{D})^\omega$, a *run* of T on input u is an infinite sequence $\rho = C_0 t_0 C_1 t_1 \dots \in (\text{Confs}(T) \delta)^\omega$ such that for all $i \in \mathbb{N}$, $C_i \xrightarrow[\gamma_i, e_i]{\sigma_i, d_i | \gamma_i, e_i}_T C_{i+1}$, where $(\sigma_i, d_i) = u[i]$ for some $(\gamma_i, e_i) \in \Gamma \times \mathbb{D}$. We say that ρ *produces* the output $v = (\gamma_0, e_0)(\gamma_1, e_1) \dots \in (\Gamma \times \mathbb{D})^\omega$. Observe that, by letting $\text{valuations}(\rho) = v_0 v_1 \dots$ the sequence of valuations of ρ , we have, for all $i \in \mathbb{N}$, $v[i] = (\gamma_i, v_{i+1}(r_i))$.

Note that for any $C \in \text{Confs}(T)$, there exists a unique run of T on u that starts in C . In particular, there exists a unique run of T over u that is *initial*, i.e. which starts in (q^i, v_R^i) . It produces some output $v \in (\Gamma \times \mathbb{D})^\omega$. Then, T *computes* the total function $f_T : (\Sigma \times \mathbb{D})^\omega \rightarrow (\Gamma \times \mathbb{D})^\omega$ defined, for all $u \in (\Sigma \times \mathbb{D})^\omega$, by letting $f(u) = v$, where v is the output of the unique initial run of T on u .

Finally, a *partial run* is a finite prefix of a run; we say that a configuration C is *reachable* if there exists a partial run that is initial and ends in C .

Remark 5.1 It is an important property that f_T computes a total function. Indeed, endowing register transducer with a non-trivial acceptance

condition so that they can have a partial domain makes us cross the decidability border (Theorem 8.4).

Remark 5.2 To ease the definition, we picked a syntactical restriction that enforces an analogue of sequentiality (i.e., input-determinism) over data words. Note that the notion can be made a bit more flexible, by imposing a semantical restriction on δ that is more liberal than the syntactical one: instead of asking for a deterministic and complete set of tests, we require that for any reachable configuration (p, v) and any labelled data value $(\sigma, d) \in \Sigma \times \mathbb{D}$, there exists a unique transition $p \xrightarrow[\tau]{\sigma, \phi | \text{asgn}, \gamma, r} q$ which is *enabled* by d from (p, v) , i.e. such that $v, d \models \phi$. In this definition, we again have that any input data word admits an initial run and that it is unique, which is sufficient to get that a transducer computes a function (i.e. the image of each data word is unique) which is total (the image exists). The fact that both definitions are equivalent result from the equivalence between syntactical and semantical determinism for register automata (Section 4.13). We use this fact in Chapter 6, to reduce the exploration space of our algorithms.

Remark 5.3 Note that in our model, the register transducer *first* assigns the input data value to its registers, and *then* produces the corresponding output data value. This choice is consistent with the term of reactive system, and with the fact that in the finite alphabet case, the output letter can depend on the last input letter. It also allows to model natural functions, among which the identity function $\text{id}_{\mathbb{D}} : u \in (\Sigma \times \mathbb{D})^\omega \mapsto u$ (cf Figure 5.2), which is not possible if we adopt the convention that the output is produced *before* the assignment.

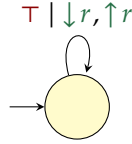


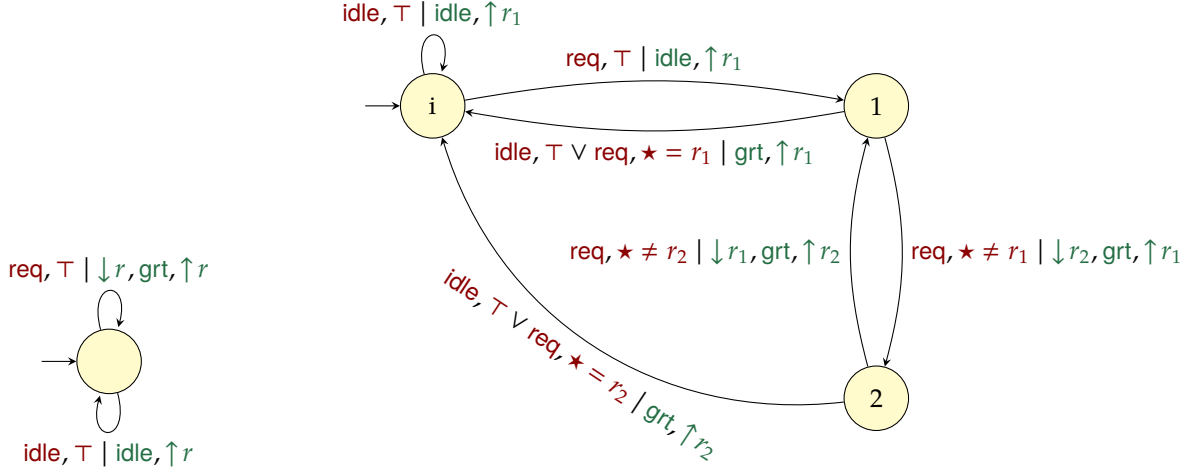
Figure 5.2.: A synchronous sequential register transducer that computes the identity function. Its label alphabets are unary and omitted.

Example 5.2 If you come back to the previous example (Example 5.1), it can be implemented by the register transducer of Figure 5.3a on the following page, which grants a request as soon as it receives it. Now, assume that the specification cannot grant a request immediately, e.g. to model the fact that the server has a buffer which receives the requests and that only transmits them to its control unit on the next clock tick. To prevent spamming, the specification also ignores a request if the same client did one on the previous time unit. The register transducer of Figure 5.3b satisfies this specification.

5.3.1. I/O Action Sequences

As for register automata, the notion of I/O action sequence allows to connect the syntax and semantics of the model (see Section 4.5.4). Given a set of registers R , an *input action* is a test $\phi \in \text{Tests}_{\mathbb{D}}(R)$. Correspondingly, an *output action* is a pair $(\text{asgn}, r) \in 2^R \times R$.

Remark 5.4 One has to be a bit careful here: input and output actions respectively correspond to the left- and right-hand-side of the type of the transition function δ (we introduce labels later on). As a consequence, tests belong to the input, while assignments belong to the output actions,



(a) A register transducer computing the function g_{rg} which, to each $i \in I^\omega$, associates the output $o \in O^\omega$ such that for all $j \in \mathbb{N}$, $i[j] = (req, id) \Leftrightarrow o[j] = (grt, id)$.

(b) A synchronous sequential register transducer waiting for one time unit before granting a request. It has two registers r_1 and r_2 , in which it alternately store the pending request. In state i , it has no pending request. When it receives one, it stores the corresponding identifier in r_1 and transitions to state 1. If it receives another, it grants the pending one and stores the new one in r_2 , switching to state 2. The $\vee$ in the transitions is a short for when there are two transitions with distinct tests but same output actions.

Figure 5.3. Two synchronous sequential register transducers that realise the request-grant specification over data words. The one on the right additionally complies with the requirement to wait for one time unit before granting a request.

even though what is assigned is the input data value. This is because when we model synthesis problems of register transducers, assignments are chosen by Eve (a.k.a. the Output player).

Definition 5.3 Let $\beta = \phi_0(\text{asgn}_0, r_0)\phi_1(\text{asgn}_1, r_1) \in (\text{Tests}(R)(2^R \times R))^\omega$ be an I/O action sequence. A pair $(u, v) \in \mathbb{D}^\omega \times \mathbb{D}^\omega$ is *compatible* with β whenever there exists an infinite sequence of valuations $(v_i)_{i \in \mathbb{N}} \in (\text{Val}_{\mathcal{D}}(R))^\omega$ such that:

- $v_0 = v_R^i$
- for all $i \in \mathbb{N}$, $v_i\{\star \leftarrow u[i]\} \models \phi_i$ and $v_{i+1} = v_i\{\text{asgn}_i \leftarrow u[i]\}$
- for all $i \in \mathbb{N}$, $v[i] = v_{i+1}(r_i)$.

In other words, (u, v) is compatible with β whenever u is compatible with $(\phi_0, \text{asgn}_0)(\phi_1, \text{asgn}_1) \dots$ in the sense of register automata and, additionally, $v = v_1(r_0)v_2(r_1)v_3(r_2) \dots$, where the v_i are defined in the compatibility relation for data words in the context of register automata (cf Section 4.5.4).

Given an I/O action sequence β , we define the set of compatible pairs of data words as

$$\text{CompIO}(\beta) = \{(u, v) \in \mathbb{D}^\omega \times \mathbb{D}^\omega \mid (u, v) \text{ is compatible with } \beta\}$$

It follows from the definitions that

Proposition 5.1 For all $(u, v) \in \mathbb{D}^\omega \times \mathbb{D}^\omega$ and all action sequences β , $(u, v) \in \text{CompIO}(\beta)$ whenever there exists a register transducer (with no labels) which admits a run ρ on input u which yields output v and such that $\text{actions}(\rho) = \beta$.

We denote β instead of α the action sequences of register transducers to distinguish them from the analogous concept in the case of register automata.

Note the shift of indices between valuations and output registers in w_o (cf Remark 5.3).

5.4. The Church Synthesis Problem over Data Words

In our study, the Church synthesis problem over data domain $\mathcal{D}$ is now stated as follows:

Problem 5.2: CHURCH SYNTHESIS PROBLEM OVER DATA WORDS

Input: A data-automatic specification $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$
 given as a register automaton A_S

Output: A register transducer T computing
 a function $f_T : (\Sigma \times \mathbb{D})^\omega \rightarrow (\Gamma \times \mathbb{D})^\omega$
 which realises S (i.e. $f_T \subseteq S$) if it exists
 No otherwise

Of course, we do not claim that this is the only possible generalisation of the Church synthesis problem to the setting of infinite alphabets.

In the finite alphabet case, the Church synthesis problem can be solved by solving a two-player game that we introduced as the Church game, as explained in Section 3.5.3. This game can be generalised to the case of data words. Winning strategies of Eve again correspond to implementations of the specification, and winning strategies for Adam witness that the specification is not realisable. The game is now defined over an infinite set of actions, so it cannot be solved directly, and, as we show, it is undecidable for most specification formalisms (see Theorems 6.20, 7.1 and 7.19). However, many of our proofs rely on reductions to games, and correctness of those reductions is proven by showing that winning strategies in them induce strategies that are winning in the Church data game.

Definition 5.4 (Church data game) Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Sigma \times \mathbb{D})^\omega$ be a data word specification. The Church data game G_S over S is defined as a two-player game with two vertices $v_\forall$ and $v_\exists$, in which Adam picks inputs in $(\Sigma \times \mathbb{D})$, while Eve picks outputs in $(\Gamma \times \mathbb{D})$. Thus, edges of Adam are $v_\forall \xrightarrow{(\sigma, d)} v_\exists$ for all $(\sigma, d) \in \Sigma \times \mathbb{D}$, and $v_\exists \xrightarrow{(\sigma, d)} v_\forall$ for Eve. A play is thus an infinite sequence $\rho = v_\forall i_0 v_\exists o_0 v_\forall i_1 v_\exists o_1 \dots$; it is winning if $\text{actions}(\rho) = i_0 o_0 i_1 o_1 \dots \in \langle S \rangle$, i.e. $(i_0 i_1 \dots, o_0 o_1 \dots) \in S$.

As for finite alphabets, it follows from the definitions that (cf Proposition 3.13):

Proposition 5.2 *There exists a synchronous program which implements S if and only if Eve has a winning strategy in G_S that is computable.*

However, the notion of finite-state computable programs and of finite-memory strategy are obsolete in the data case, as the elements that they manipulate take their values in an infinite set. We propose to generalise these notion through register transducers: a program is register computable whenever it can be computed by a register transducer. Correspondingly, a register strategy is a strategy that can be computed using a register machine. It is immediate to show that:

Proposition 5.3 *There exists a register computable program which implements S if and only if Eve has a register winning strategy in G_S .*

Register-Bounded Synthesis of Register Transducers

6.

Already over data domains with equality only, the Church synthesis problem is undecidable for specifications expressed by non-deterministic (Theorem 6.20) and universal register automata (Theorem 7.1), as universality (respectively emptiness) is undecidable for those models. The latter result is detailed in the next Chapter 7 (Unbounded Synthesis of Register Transducers). Intuitively, it results from the fact that the number of registers of the implementation is not bounded a priori, which makes the exploration space infinite. As explained in [36], one can recover decidability by additionally taking as input a bound k , and only consider implementations with at most k registers. Formally, the *register-bounded synthesis problem over data words* is defined as:

Problem 6.1: REGISTER-BOUNDED SYNTHESIS PROBLEM OVER DATA WORDS

- Input:** A data word specification $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ and an integer bound $k \geq 1$
- Output:** A register transducer T with k registers computing a function $f_T : (\Sigma \times \mathbb{D})^\omega \rightarrow (\Gamma \times \mathbb{D})^\omega$ which realises S (i.e. $f_T \subseteq S$) if it exists
No otherwise

For the complexity analysis, we assume that k is given in unary. Indeed, describing a register automaton with k registers in general requires $O(k)$ bits, and not $O(\log k)$ bits.

Remark 6.1 (Register-bounded synthesis over finite alphabets) This problem is reminiscent to the *bounded synthesis* approach of [3] that consists in setting a bound on the number of states of the implementation. While it relies on the same idea of bounding the exploration space, here, not all parameters are bounded as the state space is left unconstrained. This is the reason why we chose the more precise terminology of ‘register-bounded synthesis’, inspired by [35].

And, as it happens, the register-bounded synthesis problem over data words generalises the Church synthesis problem that we defined over finite alphabets (Problem 3.4). Indeed, recall that a register automaton (respectively a register transducer) with no registers is simply an ω -automaton (resp., an ω -transducer). Thus, when the specification given as input has no registers and in the limit case where $k = 0$, the register-bounded synthesis problem over data words reduces to finding a transducer implementation with arbitrarily many states.

In this chapter, we develop a generic method to solve the slightly more general template-guided synthesis problem, which takes as input a template that constrains the actions of the target register transducer implementation (in particular, it bounds its number of registers). The method is generic in the sense that it is agnostic with regard to the specification formalism. We then apply it to the resolution of the register-bounded synthesis problem for specifications expressed as universal register automata (Theorems 6.14 and 6.16). In contrast, we demonstrate that the register-bounded synthesis problem is undecidable for specifications expressed as non-deterministic register automata, already for a

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

[3]: Finkbeiner and Schewe (2013), ‘Bounded synthesis’

[35]: Khalimov and Kupferman (2019), ‘Register-Bounded Synthesis’

bound $k = 1$ (Theorem 6.19), and exhibit a subclass of specifications that allows to recover decidability (Theorem 6.25).

Remark 6.2 (Syntax-guided synthesis) The terminology of template-guided synthesis is loosely related to syntax-guided synthesis [2]: the idea is again to receive as input a template for the implementation, but the objectives differ. In our setting, a template simply consists in an ω -regular constraint on the action sequences of the target transducer, and the focus is set on bounding the number of registers. In contrast, syntax-guided synthesis targets programs in a given programming language and is based on a context-free grammar that describes their syntax. The exploration space is bounded, so as to allow a SAT-based exploration, and recursion is forbidden to get decidability.

Related Publications A preliminary version of this work has been published in the conference paper [37]. It is focused on the register-bounded synthesis problem in the case of $(\mathbb{D}, =, \#)$, and considers register automata with maximally consistent tests. An extended version appeared as [41], that features a more flexible formalism for register automata allowing implicit tests that are given as logical formulas, as well as simplified proofs. This chapter generalises the study to the template-guided synthesis problem and correspondingly introduces the notion of template. This is done to allow a finer-grained synthesis algorithm, that targets single-assignment register transducers. The applications are extended to the case of $(\mathbb{Q}, <, 0)$, and the constructions also yield decidability for universal register automata with guessing.

Summary

6.1. A Generic Approach to Template-Guided Synthesis with Registers

In this section, we provide a generic approach (Section 6.1) to register-bounded synthesis, and use it to explore the problem over different data domains. This first allows us to reprove decidability of the problem for specifications expressed as universal register automata over data domains with equality only (Section 6.2.1 and Theorem 6.14, first proven in [36, Corollary 3]). We then extend the result to $(\mathbb{Q}, <, 0)$ and we further allow data guessing (Section 6.2.3). Unfortunately, register-bounded synthesis is undecidable for specifications expressed by non-deterministic register automata, already over data domains with equality only (Theorem 6.19). However, we present a restriction of non-deterministic register automata, namely test-freeness, that allows to recover decidability in the non-deterministic case (Section 6.3.2).

The notion of compatible relational data word connects the syntax and semantics of register transducers. As we show, this connection yields a generic method to tackle the register-bounded synthesis problem. In the rest of this section, we fix the data domain $\mathcal{D}$, input alphabet Σ and output alphabet Γ .

[2]: Alur et al. (2013), ‘Syntax-guided synthesis’

Actually, the constraint is even simpler as it restricts the alphabet of actions, but the approach can be extended to any ω -regular language of action sequences (Remark 6.10).

[37]: Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Ed. by Wan Fokkink and Rob van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019

[41]: Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *Logical Methods in Computer Science* 17.1 (2021)

These works also contain material that is related to Chapters 7 and 8. This will be detailed in due course.

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

Recall that given a set of registers R , we defined labelled input actions as elements of $A_R^i = \Sigma \times \text{Tests}_{\mathcal{D}}(R)$ and labelled output actions as elements of $A_R^o = 2^R \times \Gamma \times R$. In the following, we omit the term ‘labelled’ as it is clear from the context. Given a sequence $\beta \in \text{ActSeqIO}(R)$ that alternates between input and output actions, the set of compatible relational data words is given as

$$\text{ComplO}(\beta) = \left\{ (u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \mid \begin{array}{l} \text{there exists a run } \rho \text{ s.t.} \\ \text{actions}(\rho) = \beta, \\ \rho \text{ is an initial run} \\ \text{over } u \text{ yielding } v \end{array} \right\}$$

Remark 6.3 Note that the notion of run depends on the machine we consider. However, the existence of a run over a given data word actually only depends on the sequence of actions, as in the case of register automata when we introduced action sequences (Section 4.5.4). For instance, for a finite data word $1 \cdot 1 \cdot 2$ over $(\mathbb{D}, =)$, it does not matter whether the run is $p \xrightarrow{\star \neq r, \downarrow} q \xrightarrow{\star = r} r \xrightarrow{\star \neq r} s$ or $p \xrightarrow{\star \neq r, \downarrow} p \xrightarrow{\star = r} p \xrightarrow{\star \neq r} p$: the compatibility relation does not depend on the states.

6.1.1. Templates

Bounding the number of registers of the target implementation already limits the exploration space, which is still infinite in general. We additionally restrict the space of tests, as there are infinitely many distinct formulas over $R_k \sqcup \{\star\}$, and doubly exponentially many that are not equivalent. To that end, we introduce the notion of template.

Definition 6.1 Let $\mathcal{D}$ be a data domain and R be a finite set of registers. A *template* for R is a finite set of pairs of input and output actions $\mathfrak{T} \subset_f A_R^i \times A_R^o$. We say that a sequential register transducer $T = (Q, \Sigma, \Gamma, q^t, \delta)$ *complies* with $\mathfrak{T}$ when $\delta \subseteq Q \times \mathfrak{T} \times Q$.

To a template $\mathfrak{T}$, we associate its *input* and *output alphabets*, respectively defined as $\mathfrak{T}_i = \pi_1(\mathfrak{T})$ and $\mathfrak{T}_o = \pi_2(\mathfrak{T})$. Then, we say that a (register-free) sequential transducer $T = (Q, \mathfrak{T}_i, \mathfrak{T}_o, q^t, \delta)$ *complies* with $\mathfrak{T}$ when $\delta \subseteq Q \times \mathfrak{T} \times Q$.

Example 6.1 For any set of registers R and any finite set $\Phi \subset_f \text{Tests}_{\mathcal{D}}(R)$, $(\Sigma \times \Phi) \times A_R^o$ is a template. Any register transducer T with tests Φ complies with the template $(\Sigma \times \Phi) \times A_R^o$.

Among the above, $\mathfrak{T}_R^{\text{MC}} = (\Sigma \times \text{MCTests}_{\mathcal{D}}(R)) \times A_R^o$ is a template. As we see later, not all templates are equal, and the reader might feel that since any test can be turned into maximally consistent ones (Proposition 4.5), the latter will play a special role. But let us not put the cart before the horse.

Remark 6.4 (Why templates?) The notion of template makes a link between tests and assignments. This allows for finer-grained synthesis algorithms. For instance, we know that single-assignment register transducers are expressively equivalent to multiple-assignment ones (Proposition 6.5), which allows to target those implementations without loss of generality; such machines can be expressed as a template (see Proposition 6.6). Knowing that the machine is single-assignment then

When we write $\delta \subseteq Q \times \mathfrak{T} \times Q$, we implicitly interpret δ as a relation. Thus, we mean that for all $(p, \sigma, \phi) \in \text{dom}(\delta)$, $(p, \sigma, \phi, \text{asgn}, \gamma, q) \in Q \times \mathfrak{T} \times Q$, where $(\text{asgn}, \gamma, q) = \delta(p, \sigma, \phi)$.

One may say that some templates are more equal than others.

permits to prune tests that are not consistent with single-assignment valuations (e.g. those that ask that $r_1 \neq \# \wedge r_1 = r_2$, see Proposition 4.7), reducing the set of tests by an exponential factor.

Template-Guided Synthesis

A template serves as constraining the shape of the implementation. Correspondingly, we define two synthesis problems, respectively targeting register transducers and ω -transducers that comply with $\mathfrak{T}$:

Problem 6.2: $\mathfrak{T}$ DATA GUIDED SYNTHESIS PROBLEM

- Input:** A data word specification $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$
Output: A register transducer T that complies with $\mathfrak{T}$ and which realises S (i.e. $\llbracket T \rrbracket \subseteq S$) if it exists
 No otherwise

One can then define the analogous problem for words over a finite alphabet:

Problem 6.3: $\mathfrak{T}$ GUIDED SYNTHESIS PROBLEM

- Input:** A specification $W \subseteq \mathfrak{T}_i^\omega \times \mathfrak{T}_o^\omega$
Output: A (register-free) transducer T that complies with $\mathfrak{T}$ and which realises W (i.e. $\llbracket T \rrbracket \subseteq W$) if it exists
 No otherwise

If W is ω -regular, the latter problem is decidable: restricting to transducers that comply with $\mathfrak{T}$ can be done by including the requirement in the specification, and we know that the transducer synthesis problem is decidable for automatic specifications (Theorem 3.15). Thus, our aim is to reduce the $\mathfrak{T}$ -data guided synthesis problem to its finite alphabet homologue.

Definition 6.2 Let R be a set of registers, and $\mathfrak{T}$ be an a template over R . We associate to it the set of action sequences over $\mathfrak{T}$

$$\text{ActSeqIO}(\mathfrak{T}) = \{a_0 b_0 a_1 b_1 \cdots \in (\mathfrak{T}_i \mathfrak{T}_o)^\omega \mid \forall i \geq 0, (a_i, b_i) \in \mathfrak{T}\}$$

In particular, $a_0 b_0 a_1 b_1$ is an I/O action sequence, i.e. $a_0 b_0 a_1 b_1 \cdots \in \text{ActSeqIO}(R)$.

From Data Words to Words: the Associated Specification

Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a data word specification and R be a set of registers. At this point, we make no assumptions on the formalism used to express S . Given a template $\mathfrak{T}$ over R , we define a finite alphabet specification $W_S^\mathfrak{T}$ that accepts action sequences of $\mathfrak{T}$ whose compatible relational data words all belong to S . We call it the *associated specification*. Formally,

$$W_S^\mathfrak{T} = \{\beta \in \text{ActSeqIO}(\mathfrak{T}) \mid \text{CompIO}(\beta) \subseteq S\}$$

Soundness

The above specification allows to reduce the reactive synthesis problem of S to a synthesis problem over finite alphabets, provided we restrict to well-behaved templates. First, one should consider templates that indeed yield register automata. For instance, there is no hope that the singleton template $\{(\perp, \emptyset)\}$ gives rise to a register transducer, as it

induces a transition function that is not complete. One should also exclude templates whose tests are not mutually exclusive, e.g. $\{(\top, \emptyset), (\star = r, \emptyset)\}$ (for some arbitrary choice of $r \in R$), as the transition function that one obtains is not deterministic.

Definition 6.3 Let $\mathfrak{T}$ be a template. It is *sound* if any (register-free) ω -transducer $T = (Q, A, B, q^i, \delta)$ for $A \subset_f A_R^i$ and $B \subset_f A_R^o$ that complies with $\mathfrak{T}$ is such that $(Q, \Sigma, \Gamma, q^i, \delta)$ is a register transducer with registers R , i.e. δ is deterministic and complete when interpreted over data values.

Observation 6.1 Consider again the template $\mathfrak{T}_R^{\text{MC}}$ of Example 6.1. This template is sound: let $T = (Q, \Sigma \times \text{MCTests}(R), A_R^o, q^i, \delta)$ be a (register-free) transducer that complies with $\mathfrak{T}_R^{\text{MC}}$. Then, for any reachable configuration (q, ν) and any data value $d \in \mathbb{D}$, there exists a unique maximally consistent test ϕ such that $(\nu, d) \models \phi$, so δ is deterministic and complete over data values.

Soundness indeed yields a sound reduction:

Lemma 6.2 Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a data word specification, and let $\mathfrak{T}$ be a sound template.

If the finite alphabet specification $W_S^{\mathfrak{T}}$ is realisable by a synchronous sequential transducer (with no registers) that complies with $\mathfrak{T}$, then the data word specification S is realisable by a synchronous sequential register transducer that complies with $\mathfrak{T}$.

Proof. Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a data word specification, and let $\mathfrak{T}$ be a sound template, and assume that $W_S^{\mathfrak{T}}$ is realisable by a (register-free) synchronous sequential transducer $T = (Q, A, B, q^i, \delta)$ that complies with $\mathfrak{T}$. Define the machine $T' = (Q, q^i, \Sigma, \Gamma, R, \delta)$. Since $\mathfrak{T}$ is sound, T' is a synchronous sequential register transducer.

Remark 6.5 The notion of template is a bit abstract, so let us instantiate it to $\mathfrak{T}_R^{\text{MC}}$ to illustrate the reasoning. Then, $T = (Q, A, B, q^i, \delta)$, where $A = \Sigma \times \text{MCTests}_{\mathbb{D}}(R)$ and $B = 2^R \times \Gamma \times R$. Thus, we have $\delta : Q \times (\Sigma \times \text{MCTests}_{\mathbb{D}}(R)) \rightarrow 2^R \times \Gamma \times R$. This is exactly the type of the transition function of a register transducer whose tests are maximally consistent. And we indeed get a sequential machine, since those tests are mutually exclusive and their disjunction is valid, which implies that for any reachable configuration and any labelled data value $(\sigma, d) \in \Sigma \times \mathbb{D}$, there exists a unique transition that is enabled by (σ, d) .

We need to show that T' realises S . Let $u \in (\Sigma \times \mathbb{D})^\omega$, and let ρ' be the run of T' over u ; it yields some output $v = f_{T'}(u)$. Denote $\beta = \text{actions}(\rho')$. By construction, $\beta = \text{actions}(\rho)$ for some run ρ of the finite alphabet transducer T , so $\text{CompIO}(\beta) \subseteq S$. Since, by definition of the semantics of register transducers, $(u, v) \in \text{CompIO}(\beta)$, we get that $(u, v) \in S$. Overall, this means that $f_{T'} \subseteq S$; in other words, T' realises S . $\square$

Completeness

In general, we target implementations that belong to a specific class of register transducers, for instance those with a number k of registers. To that end, we need to be able to express any register transducer of the class with the template. Consider e.g. the template $\{(\top, \emptyset)\}$. It is sound, as any ω -transducer that complies with it indeed corresponds to a register

transducer. However, it can only represent those that conduct no tests over their input nor assignments, so it only represents a (very restricted) subclass of register transducers.

Definition 6.4 Let $\mathcal{T}$ be a class of register transducers with registers R . We say that a template $\mathfrak{T}$ over R is *complete* for $\mathcal{T}$ if any transducer in this class admits an equivalent transducer that complies with $\mathfrak{T}$.

Remark 6.6 We could generalise the notion of template by allowing different sets of registers in the same template, to get a notion of completeness for classes of transducers over different sets of registers. However, since a template is anyway finite, it fixes a bound k on the number of registers of the transducers in the corresponding class. Then, any transducer in the class can be simulated by a transducer with registers R_k , where R_k is a set of registers of size k .

The template $\{(\top, \emptyset)\}$ is tautologically complete for the (mildly interesting) class of register transducers with actions $(\top, \emptyset)$, but it is not complete for the class of transducers with one (or more) registers. In contrast, we know that maximally consistent tests allow to express any test. In combination with Observation 6.1), we get:

Proposition 6.3 For any set of registers R , $\mathfrak{T}_R^{MC}$ is a sound and complete template for register transducers with registers R .

Proof. The proof of Proposition 4.5 is directly adapted to the case of transducers, as it only relies on writing tests in full disjunctive normal form, so we know that any register transducer $T = (Q, q^i, \Sigma, \Gamma, R, \delta)$ can be turned into an equivalent $T'' = (Q, q^i, \Sigma, \Gamma, R, \delta'')$ which has the same states and such that δ'' is a partial function $\delta'' : Q \times \Sigma \times \text{MCTests}_{\mathcal{D}}(R) \rightarrow 2^R \times \Gamma \times R \times Q$. $\square$

Using reassignments, we can further restrict to singleton assignments.

Notation 6.5 In the following, we denote $\text{SAsgn}_R = \{\{r\} \mid r \in R\} \sqcup \{\emptyset\}$.

Proposition 6.4 For any set of registers R , $\mathfrak{T}_R^{MCSA} = (\Sigma \times \text{MCTests}_{\mathcal{D}}(R)) \times (\text{SAsgn}_R \times \Gamma \times R)$ is a sound and complete template for register transducers with registers R .

Proof. Soundness is established as above: since maximally consistent tests are mutually exclusive, they induce a deterministic transition function. Moreover, their disjunction is valid, so they yield a complete transition function.

Completeness is obtained as follows: take a register transducer T . Any transition $p \xrightarrow{\sigma, \phi | \text{asgn}, \gamma, r} q$ with $\text{asgn} \neq \emptyset$ is equivalent to $p \xrightarrow{\sigma, \phi | \{s\}, \text{asgn} := s, \gamma, r} q$, where $s \in \text{asgn}$ is chosen arbitrarily and $\text{asgn} := s$ corresponds to the reassignment function $f : t \in \text{asgn} \mapsto s$ and $t \notin \text{asgn} \mapsto t$. Removing reassignments is done as in Proposition 4.6. One just has to be careful in the case where the output register r belongs to asgn , but this poses no major difficulty. Then, one can turn this transducer into one with maximally consistent tests as was done in Proposition 4.5, and make the transition function total as above. $\square$

Further, the proof of Proposition 4.8 again works for transducers, since the construction preserves determinism. Thus, we get:

Proposition 6.5 Let $T = (Q, q^i, \Sigma, \Gamma, R, \delta)$ be a register transducer. There exists an equivalent single-assignment register transducer $T' = (Q \times R^R, q^i \times \{\text{id}_R\}, \Sigma, \Gamma, R, \delta')$.

Proof. The proof of Proposition 4.8 directly extends to transducers. As above, one just has to be careful in the case where the output register belongs to asgn . $\square$

As a consequence, we can target implementations that consist in single-assignment register transducers.

Proposition 6.6 For any set of registers R , the following set is a sound and complete template for register transducers with registers R :

$$\mathfrak{T}_R^{\text{SA}} = \left\{ ((\sigma, \phi), (\text{asgn}, \gamma, r)) \in \mathfrak{T}_R^{\text{MCSA}} \left| \begin{array}{l} \text{if there exists } s \in R \text{ such that} \\ \phi \wedge (\star = s) \text{ is satisfiable} \\ \text{then } \text{asgn} = \emptyset \end{array} \right. \right\}$$

Conversely, if a template is complete for a class, we get the other direction:

Lemma 6.7 Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a data word specification, and let $\mathfrak{T}$ be a template that is complete for a class $\mathcal{F}$.

If the data word specification S is realisable by a synchronous sequential register transducer in $\mathcal{F}$, then the finite alphabet specification $W_S^{\mathfrak{T}}$ is realisable by a synchronous sequential transducer (with no registers).

Proof. Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a data word specification, let $\mathfrak{T}$ be a template that is complete for some class $\mathcal{F}$, and assume that S is realisable by a synchronous sequential register transducer $T = (Q, q^i, \Sigma, \Gamma, R, \delta)$ that belongs to $\mathcal{F}$. Since $\mathfrak{T}$ is complete for $\mathcal{F}$, we know that there exists an equivalent register transducer $T' = (Q', q'^i, \Sigma, \Gamma, R, \delta')$ that complies with $\mathfrak{T}$. By definition, this implies that δ' is total, so T' can be seen as a (register-free) synchronous sequential transducer $T'' = (Q', \mathfrak{T}_i, \mathfrak{T}_o, q'^i, \delta')$. It computes some (total) function $f_{T''} : \mathfrak{T}_i^\omega \rightarrow \mathfrak{T}_o^\omega$.

Remark 6.7 As in Remark 6.5, let us illustrate the argument with $\mathfrak{T}_R^{\text{MC}}$, which is complete for the class of register transducers with registers R . If S is realised by a register transducer T with registers R , then one can turn it into an equivalent transducer with maximally consistent tests. Since its transition function is total, it can be interpreted as a register-free transducer.

Let us show that T'' realises $W_S^{\mathfrak{T}}$. Let $\beta_i \in \mathfrak{T}_i^\omega$, $\beta_o = f_{T''}(\beta_i)$ and $\beta = \langle \beta_i, \beta_o \rangle$. Correspondingly, let ρ be the run of T'' over β_i yielding output β_o . We need to show that $\text{CompIO}(\beta) \subseteq S$. There are two cases:

- $\text{CompIO}(\beta) = \emptyset$. This is the case when β is not feasible. Then $\text{CompIO}(\beta) \subseteq S$ trivially holds.
- Otherwise, let $(u, v) \in \text{CompIO}(\beta)$. By construction, $\text{actions}(\rho)$ is the sequence of actions of a run ρ' of T' which, on input u , yields output $f_{T'}(u) = v$. Since T realises S , we have $f_T \subseteq S$, so $f_{T'} = f_T \subseteq S$, hence $(u, v) \in S$. This implies that $\text{CompIO}(\beta) \subseteq S$.

Overall, we get that T'' realises $W_S^{\mathfrak{T}}$. $\square$

The notion of single-assignment is naturally extended to transducers as follows: a transducer is *single-assignment* if for all transitions $p \xrightarrow{\sigma, \phi | \text{asgn}, \gamma, r} q$, if $\phi \wedge (\star = s)$ is satisfiable for some $s \in R$ then $\text{asgn} = \emptyset$, and otherwise asgn is a singleton or the empty set.

Recall that $\langle a_0 a_1 \dots, b_0 b_1 \dots \rangle$ is defined as $a_0 b_0 a_1 b_1 \dots$.

From Data Words to Words: the Transfer Theorem

Combining Lemma 6.2 (soundness) with Lemma 6.7 (completeness) yields:

Theorem 6.8 (Transfer, [41, Theorem 4.1]) *Let $\mathfrak{T}$ be a sound and complete template for some class $\mathcal{F}$, and let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a data word specification. The following are equivalent:*

1. *The data word specification S is realisable by a synchronous sequential register transducer in $\mathcal{F}$.*
2. *The finite alphabet specification $W_S^{\mathfrak{T}}$ is realisable by a synchronous sequential transducer (with no registers).*

Remark 6.8 This transfer theorem also transfers objects, as witnessed by the proofs of the lemma: a register transducer implementation of S can be converted to an ω -transducer implementation of $W_S^{\mathfrak{T}}$, and conversely.

This transfer theorem has the following corollary.

Corollary 6.9 *Let $\mathcal{S}$ be a class of data word specification, and let R be a set of registers of size $|R| = k$.*

If $W_S^{\mathfrak{T}^{\text{SA}}}$ is effectively automatic for all $S \in \mathcal{S}$, then the synthesis problem of implementations defined by register transducers with k registers is decidable for $\mathcal{S}$.

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

Recall that a finite alphabet specification is automatic if it can be recognised by a specification automaton.

Proof. Let $k \geq 1$, and let R_k be some arbitrary set of k registers (its choice is irrelevant). By Proposition 6.6, we have that $\mathfrak{T}^{\text{SA}}$ is a sound and complete template for register transducers with registers R_k . Let S be a specification over data words. By Theorem 6.8, S is realisable by a transducer with k registers if and only if $W_S^{\mathfrak{T}^{\text{SA}}}$ is realisable by an ω -transducer, so the k -register-bounded synthesis problem for S reduces to the synthesis problem for $W_S^{\mathfrak{T}^{\text{SA}}}$. If the latter is effectively automatic, i.e. if it can be represented by an ω -automaton $A_{S,k}$ that can be computed from S , solving the synthesis problem for $W_S^{\mathfrak{T}^{\text{SA}}}$ reduces to solving a game with $A_{S,k}$ as an observer. This is decidable by Theorem 3.14. $\square$

Recall that any ω -automaton is equivalent with a parity automaton. The proof of Theorem 3.14 then relies on determining the observer, which is always doable (at an exponential price), cf Fact 3.1.

Now, we establish decidability of the register-bounded synthesis problem for different specification formalisms and for data domains $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$. This is done by leveraging the above corollary, which amounts to show that the associated specification is effectively automatic, i.e. to construct an ω -automaton that recognises it.

Expressing the Associated Specification with Language Operations

In the following, we fix a template $\mathfrak{T}$ over registers R . Our goal is to express the associated specification $W_S^{\mathfrak{T}}$ using operations on data languages, to further reduce the problem to establishing properties of the specification model. More precisely, we show that Equation 6.2 on page 146 holds, where $L_{\mathfrak{T}}$ is defined in Equation 6.1 on page 146. This formula looks barbarian, and it indeed is. However, the point is not so much to align funny-looking characters as to highlight the fact that $W_S^{\mathfrak{T}}$ can be expressed from the specification formalism using the following operations:

- Complement
- Product of a data relation with an automatic relation, more precisely the relation $\text{ActSeqIO}(\mathfrak{T}) = \{a_0a_1 \dots, b_0b_1 \dots \in \mathfrak{T}_i^\omega \times \mathfrak{T}_o^\omega \mid a_0b_0a_1b_1 \dots \in \text{ActSeqIO}(\mathfrak{T})\}$
- Intersection
- Projection over labels

Moreover, $L_{\mathfrak{T}}$ is definable by a non-deterministic register automaton, and even a deterministic one, although with more states (Lemma 6.11). Together with Corollary 6.9, this yields decidability of the register-bounded synthesis problem for specifications expressed as universal register automata over $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, 0)$ (respectively Theorems 6.14 and 6.16).

Among the above constructs, product and intersection will pose no major difficulty, as register automata are closed under those operations: Propositions 4.2 and 4.3 for intersection; the construction for the product is easy. On the contrary, they are not closed under complement, and it is actually the case of most models over infinite alphabets whose emptiness is decidable (see [103]). That is why we get positive results for universal register automata, but need to consider a subclass in the case of non-deterministic register automata, while the latter enjoy better properties (contrast Section 4.5 with Section 4.8).

[103]: Björklund and Schwentick (2010), 'On notions of regularity for data languages'

Before writing the dreadful Equation 6.2, we show the following lemma, which has a more semantical flavour. For simplicity, we define a variant of the word product that implicitly amalgamates labels in Σ and Γ (or, equivalently, consider unlabelled input and output actions). For an I/O action sequence $\beta = (\sigma_0, \phi_0)(\text{asgn}_0, \gamma_0, r_0)(\sigma_1, \phi_1) \dots \in \text{ActSeqIO}(\mathfrak{T})$ and for data words $u = (\sigma'_0, d_0)(\sigma'_1, d_1) \dots \in (\Sigma \times \mathbb{D})^\omega$, $v = (\gamma'_0, e_0)(\gamma'_1, e_1) \dots \in (\Gamma \times \mathbb{D})^\omega$, $\beta \otimes (u, v)$ is only defined when for all $i \in \mathbb{N}$, $\sigma_i = \sigma'_i$ and $\gamma_i = \gamma'_i$. We then let $\beta \otimes (u, v) \in (A_{\mathfrak{T}}^i \times \mathbb{D})^\omega \times (A_{\mathfrak{T}}^o \times \mathbb{D})^\omega$ be

$$\beta \otimes (u, v) = (((\sigma_0, \phi_0), d_0)((\sigma_1, \phi_1), d_1) \dots, ((\gamma_0, \text{asgn}_0, r_0), e_0)((\gamma_1, \text{asgn}_1, r_1), e_1) \dots)$$

Lemma 6.10 Consider the set $L_{S, \mathfrak{T}}$ of relational data words $w \in S$ labelled by action sequences $\beta \in \text{ActSeqIO}(\mathfrak{T})$ such that $w \in \text{CompIO}(\beta)$. Formally,

$$L_{S, \mathfrak{T}} = \left\{ \beta \otimes (u, v) \mid \begin{array}{l} \beta \in \text{ActSeqIO}(\mathfrak{T}), (u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \\ (u, v) \in \text{CompIO}(\beta) \text{ and } (u, v) \in S \end{array} \right\}$$

Then $W_S^{\mathfrak{T}}$ can be obtained from the complement of S by projecting on labels and complementing back:

$$W_S^{\mathfrak{T}} = \left(\text{lab} \left(L_{S^c, \mathfrak{T}} \right) \right)^c$$

Proof. Let $\beta \in \text{ActSeqIO}(\mathfrak{T})$. Then,

$$\begin{aligned} & \beta \notin W_S^{\mathfrak{T}} \\ \Leftrightarrow & \text{CompIO}(\beta) \not\subseteq S \\ \Leftrightarrow & \exists (u, v) \in \text{CompIO}(\beta) \cap S^c \\ \Leftrightarrow & \exists (u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \text{ such that } \beta \otimes (u, v) \in L_{S^c, \mathfrak{T}} \\ \Leftrightarrow & \beta \in \text{lab}(L_{S^c, \mathfrak{T}}) \quad \square \end{aligned}$$

Now, $L_{S, \mathfrak{T}}$ can be decomposed further. Define

$$L_{\mathfrak{T}} = \left\{ \beta \otimes (u, v) \mid \begin{array}{l} \beta \in \text{ActSeqIO}(\mathfrak{T}), \\ (u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \\ (u, v) \in \text{ComplO}(\beta) \end{array} \right\} \quad (6.1)$$

Observe that

$$L_{S, \mathfrak{T}} = (\text{ActSeqIO}(\mathfrak{T}) \otimes S) \cap L_{\mathfrak{T}}$$

Together with Lemma 6.10, we get:

$$W_S^{\mathfrak{T}} = (\text{lab}((\text{ActSeqIO}(\mathfrak{T}) \otimes S^c) \cap L_{\mathfrak{T}}))^c \quad (6.2)$$

We conclude the section by showing that $L_{\mathfrak{T}}$ is recognisable by a register automaton.

Lemma 6.11 *The data relation $L_{\mathfrak{T}}$ defined in Equation 6.1 is recognised by a non-deterministic register automaton with states $2^R \sqcup \{i\}$ and registers R .*

Additionally, $L_{\mathfrak{T}}$ is recognised by a deterministic register automaton with reassignments with states $\{i, o\}$ and registers $R^t = R \sqcup \{r_{\text{tmp}}\}$. Equivalently, a deterministic register automaton with states $\{i, o\} \times (R^t)^{R^t}$ and registers R^t .

They have a trivial acceptance condition, i.e. all states are accepting, and operate over input alphabet $\mathfrak{T}_i$ and output alphabet $\mathfrak{T}_o$.

Proof. We first define a non-deterministic register automaton recognising $L_{\mathfrak{T}}$. It has states $2^R \sqcup \{i\}$, where i is its sole initial state. It is also its only input state. From state i , it reads a labelled data value $((\sigma, (\lambda, \phi)), d) \in \mathfrak{T}_i \times \mathbb{D}$. Then, it:

- (i) Checks that labels coincide, i.e. $\sigma = \lambda$
- (ii) Checks that d satisfies test ϕ in its current register configuration
- (iii) Guesses an assignment asgn and performs it, i.e. assigns d to registers in asgn
- (iv) Transitions to the corresponding state asgn .

Now, in any output state $\text{asgn} \in 2^R$, the automaton, on reading labelled data value $((\gamma, (\text{asgn}', \zeta, r)), e) \in (\Gamma \times A_k^\circ) \times \mathbb{D}$,

- (i) Checks that labels coincide, i.e. $\gamma = \zeta$
- (ii) Checks that its guess was correct, i.e. $\text{asgn}' = \text{asgn}$
- (iii) Verifies that the e is indeed the content of r , i.e. conducts test $\phi_r := \star = r$.
- (iv) Transitions back to i . Note that it conducts no assignment.

Finally, we equip the automaton with a trivial acceptance condition. The fact that it recognises $L_{\mathfrak{T}}$ follows from the definitions.

In the above construction, the only source of non-determinism is the guessing of asgn . This is because assignments are output actions. However, the automaton can store the data value d in some additional register r_{tmp} , and then reassign it to the proper registers once it reads asgn . Moreover, there is no need for storing asgn in the states anymore. Doing so, we get a deterministic automaton. Finally, one can get rid of reassignments, as was shown in Section 4.4.5: the automaton keeps track of the reassignments in its states, through an equality relation between its registers and those of the simulated automaton which conducts reassignments. This comes

at the price of an exponential blowup, as the relations between registers are recorded through a function from R^t to R^t (Proposition 4.6). $\square$

Remark 6.9 Note that the state space of the non-deterministic register automaton can be restricted to assignments that actually belong to $\mathfrak{T}$. For instance, for the single-assignment ones (Propositions 6.4 and 6.6), one can restrict to SAsgn_R .

Remark 6.10 The above construction hints at a possible generalisation of the notion of template by assimilating with its set of compatible action sequences that would be given as an ω -regular language $L \subseteq (A_{R_k}^i A_{R_k}^o)^\omega$ of I/O action sequences (here, it is simply recognised by an ω -automaton with two states and a trivial acceptance condition). We did not include it so as not to obfuscate the argument.

In the following, we consider separately the non-deterministic and universal semantics, as there is a sharp contrast between them. Indeed, the register-bounded synthesis problem is undecidable for the former, even when the data domain only allows equality (Theorem 6.19). On the contrary, it is decidable over $(\mathbb{D}, =, \mathbb{C})$ [36, Corollary 3] (see also Section 6.2.1) and even $(\mathbb{Q}, <, 0)$ for the latter, as we show in Section 6.2.2. We start with universal automata, as they offer more positive results. We then show the above undecidability result, and introduce a subclass of non-deterministic register automata whose register-bounded synthesis problem is decidable.

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

6.2. Specifications Expressed as Universal Register Automata

6.2.1. The Case of Equality

We start with the simplest data domain: an infinite set with equality only. The above reasoning allows us to reprove the main result of [36], namely that register-bounded synthesis is decidable, and more precisely in 2-ExpTime , for specifications over data domain $(\mathbb{D}, =, \mathbb{C})$ expressed by universal register automata (Theorem 6.14). The transfer theorem allows to exploit known properties of register automata, yielding a more compositional proof.

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

First, let us define the template we use. It essentially consists in the single-assignment one (Proposition 6.6), where we prune tests that are not satisfiable. Indeed, by Proposition 4.7, we know that we cannot have $\star = r_1 \wedge \star = r_2$ for $r_1 \neq r_2$, except when $\star = \#$. This allows to reduce the size of the template by an exponential factor (recall that there are exponentially maximally consistent tests by Proposition 4.5). Note that for simplicity, in our definition of a sequential transducer, we asked that the transition function is syntactically deterministic and complete. However, it is readily seen that we can instead ask this on a semantical level, by only requiring that it is deterministic and complete for reachable configurations (cf Remark 5.2).

The skeptical reader can observe that the study again works if we take the more liberal template $\mathfrak{T}^{\text{SA}}$, which does not prune unrealisable tests.

Definition 6.6 (Single-assignment template for equality) Let R be a set of registers. A *single-assignment action* is one of the following:

- A test $\star = c$ for some $c \in \mathbb{C}$, along with $\text{asgn} = \emptyset$
- $\text{nonCst}(\star) \wedge \star = r$, where $\text{nonCst}(\star) = \bigwedge_{c \in \mathbb{C}} \star \neq c$, along with $\text{asgn} = \emptyset$
- $\text{nonCst}(\star) \wedge \text{locFresh}_R(\star)$ along with $\text{asgn} = \emptyset$ or $\text{asgn} = \{r\}$ for some $r \in R$

Recall that locFresh_R is defined as $\text{locFresh}_R = \bigwedge_{r \in R} \star \neq r$ (Notation 4.17).

Then, the *single-assignment template for equality* is defined as

$$\mathcal{T}_R^{\text{SA}=} = \left\{ ((\sigma, \phi), (\text{asgn}, \gamma, r)) \mid \begin{array}{l} \sigma \in \Sigma, \gamma \in \Gamma, r \in R \text{ and} \\ (\phi, \text{asgn}) \text{ is a single-assignment action} \end{array} \right\}$$

Proposition 6.12 *For any set of registers R , $\mathcal{T}_R^{\text{SA}=}$ is a sound and complete template for register transducers with registers R .*

Proof. First, let $T = (Q, A_R^i, A_R^o, q^i, \delta)$ be a (register-free) transducer. Let (q, v) be a reachable configuration. By Proposition 4.7, which is easily adapted to register transducers, we know that for all $r, r' \in R$ such that $v(r) = v(r')$, either $r = r'$ or $v(r) = v(r') = \#$. Thus, let $d \in \mathbb{D}$. There are three cases:

- $d = c$ for some $c \in \mathbb{C}$. Then $v, d \models \star = c$, and does not satisfy any other test of δ
- $d = v(r)$ for some $r \in R$. If $d = c$ for some $c \in \mathbb{C}$, we are back to the first case. Otherwise, $v, d \models \text{nonCst}(\star) \wedge \star = r$ and does not satisfy any other tests, since we cannot have $v(r) = v(r')$ for $r \neq r'$
- $d \notin v(R)$. If d is equal to some constant, we are back to the first case. Otherwise, $v, d \models \text{nonCst}(\star) \wedge \text{locFresh}_R(\star)$ at the exclusion of any other test.

This means that δ is indeed deterministic and complete, so T induces a sequential register transducer: $\mathcal{T}_R^{\text{SA}=}$ is sound.

Now, let T be some sequential transducer with registers R . Using Proposition 6.5, turn it into an equivalent single-assignment register transducer T'' . We have seen above that the single-assignment tests are mutually exclusive in the presence of single-assignment valuations, and that they are complete. As a consequence, any test ϕ of T'' can be written as $\bigvee_{i \in I} \psi_i$ for some finite set I of indices, where each ψ_i is a single-assignment test. Thus, T'' can be turned into a register transducer T' that complies with $\mathcal{T}_R^{\text{SA}=}$, so this template is complete. $\square$

Remark 6.11 Single-assignment register transducers can be exponentially less succinct than their multiple-assignment homologues (see Remark 4.14). However, it is difficult to harness this fact in an automatic procedure, as it corresponds to some kind of minimisation. In particular, the way we build our finite alphabet specification in Lemma 6.13 imposes to store the equality relations between registers in the states of the specification automaton, which cancels the succinctness gain of allowing duplicate values: the implementation is computed by solving a parity game played on the specification automaton, which yields a positional strategy that has as many states as the specification automaton, and there are exponentially many such states if equality relations are stored. On the contrary, the single-assignment template restricts to equality relations where all classes but that of $\#$ contain at most one register of the implementation. Thus, this template reduces the exploration space, on top of reducing the alphabet of input actions.

In the following, we fix a bound $k \geq 1$ and a set $R_k = \{r_1, \dots, r_k\}$ of registers. We only use the above template in the whole section, so we denote $W_S^k = W_S^{\mathcal{T}_{R_k}^{\text{SA}=\text{}}}$ the specification associated to S with regard to $\mathcal{T}_{R_k}^{\text{SA}=\text{}}$, and $L_k = L_{\mathcal{T}_{R_k}^{\text{SA}=\text{}}}$ (see Lemma 6.11). We also let A_k^i be its input alphabet, and $A_k^o = \text{SAsgn}_{R_k}$ be its output alphabet. Finally, we denote $A_k = \text{ActSeqIO}(\mathcal{T}_{R_k}^{\text{SA}=\text{}})$ the set of compatible action sequences.

Given a specification S expressed by a universal register automaton, we need to show that the associated finite alphabet specification W_S^k is ω -regular:

Lemma 6.13 *Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a specification expressed by a universal register automaton A_S over $(\mathbb{D}, =, \mathbb{C})$ and $k \geq 1$. Then, W_S^k is ω -regular.*

More precisely, if A_S has n states and r registers equipped with a parity condition Ω of index d , it is recognised by a deterministic parity automaton with $2^{\mathbb{O}(n \cdot 2^{(r+k)^2})}$ states and $\mathbb{O}(d \cdot n \cdot 2^{(r+k)^2})$ colours.

Proof. Let $A_S = (Q, I, \Sigma \cup \Gamma, R, \Delta, \Omega)$ be a universal specification register automaton recognising a specification $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$. We let $n = |Q|$ be the number of states of A_S , and $r = |R|$ its number of registers. For the complexity analysis, we treat the case where Ω is a parity condition; other ω -regular conditions are treated similarly.

By Equation 6.2, we know that $W_S^k = \left(\text{lab} \left((A_k \otimes S^c) \cap L_k \right) \right)^c$. We build step by step an automaton for W_S^k .

Complementation of URA By duality of the non-deterministic and universal semantics, S^c is recognised by $(A_S)^c$, a non-deterministic register automaton with same states and registers as A_S , and with acceptance condition Ω^c , which is a parity condition of index $d + 1$.

Word product As a consequence, $A_k \otimes S^c$ is recognised by a non-deterministic register automaton with n states and r registers: simply add labels A_k^i (respectively A_k^o) to input (respectively output) transitions.

Intersection with L_k By Lemma 6.11, we know that L_k is recognised by a non-deterministic register automaton with $2^k + 1$ states and k registers (with trivial acceptance condition). Thus, $(A_k \otimes S^c) \cap L_k$ is recognised by a non-deterministic register automaton with $n \cdot (2^k + 1)$ states and $r + k$ registers, with acceptance condition $\pi_Q^{-1}(\Omega^c)$ (Proposition 4.2).

Projection over labels By Proposition 4.19 in Section 4.5, we have that $\text{lab} \left((A_k \otimes S^c) \cap L_k \right)$ is ω -regular. It is more precisely recognised by a non-deterministic ω -automaton with $(n \cdot (2^k + 1)) \cdot (r + k + 1)^{r+k} = \mathbb{O}(n \cdot 2^{(r+k)^2})$ states and acceptance condition $\pi_Q^{-1}(\Omega^c)$.

Complement of ω -automaton Finally, by duality, we get that W_S^k is recognised by a universal ω -automaton with acceptance condition $\pi_Q^{-1}(\Omega^c)$. This automaton can be converted into a deterministic parity automaton. If Ω is a parity condition of index d , this yields a deterministic ω -automaton with $2^{\mathbb{O}(n \cdot 2^{(r+k)^2})}$ states and $\mathbb{O}((d + 1) \cdot n \cdot 2^{(r+k)^2})$ colours [95]. $\square$

π_Q is the projection over the Q component. Depending on the state space, the inverse π_Q^{-1} is taken on a different set, but we do not need to be more precise as it suffices to know that the acceptance condition is easily computed from Ω .

[95]: Schewe and Varghese (2014), ‘Determinising Parity Automata’

We are now able to reprove the following result, known from [36] and [35]:

Theorem 6.14 ([36, Corollary 3], [35, Theorem 8], [41, Theorem 4.6]) *The register-bounded synthesis problem for specifications expressed as universal register automata over $(\mathbb{D}, =, \mathbb{C})$ is in 2-EXPTIME.*

Proof. Let S be a specification expressed by a universal register automaton over $(\mathbb{D}, =)$. By the previous Lemma 6.13, we construct a deterministic parity automaton $P_{S,k}$ for W_S^k . According to the transfer theorem (Theorem 6.8), S is realisable by a transducer with k registers if and only if W_S^k is realisable by a transducer. The way to decide it is to see W_S^k as a parity game and check whether Eve has a winning strategy. Parity games can be solved in time $\mathcal{O}(m^{\log d})$, where m is the number of vertices of the game and d is the parity index [89]. Here, we use Zielonka's algorithm [88] which runs in time $\mathcal{O}(m^d)$ as it does not change the complexity class. Overall, checking whether S is realisable by a transducer with k registers takes time $2^{\text{poly}(d,n)} \cdot 2^{\text{poly}(r,k)}$. $\square$

Remark 6.12 In [35, Theorem 8], the authors provide a finer complexity analysis, which establishes that their procedure is only simply exponential in the bound k . A similar reasoning applies here, as we can take the register automaton L_k to be deterministic, which implies that constraints behave deterministically over registers that belong to the implementation.

6.2.2. The Case of a Dense Order

The construction of Lemma 6.13 again works in the case of $(\mathbb{Q}, <, 0)$, as the different operations can again be conducted over this richer data domain. For simplicity, we use the template $\mathfrak{T}^{\text{SA}}$ as such (which is sound and complete by Proposition 6.6), even though some tests can be pruned due to the single-assignment property (Proposition 4.7).

In the following, we fix a bound $k \geq 1$ and a set $R_k = \{r_1, \dots, r_k\}$ of registers. As above, we denote $W_S^k = W_S^{\mathfrak{T}_{R_k}^{\text{SA}}}$ the specification associated to S with regard to $\mathfrak{T}_{R_k}^{\text{SA}}$, and $L_k = L_{\mathfrak{T}_{R_k}^{\text{SA}}}$ (see Lemma 6.11). We also let A_k^i be its input alphabet, and $A_k^o = \text{SAsgn}_{R_k}$ be its output alphabet. Finally, we denote $A_k = \text{ActSeqIO}(\mathfrak{T}_{R_k}^{\text{SA}})$ the set of compatible action sequences.

Lemma 6.15 *Let $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ be a specification expressed by a universal register automaton A_S over $(\mathbb{Q}, <, 0)$ and let $k \geq 1$. Then, W_S^k is ω -regular.*

More precisely, if A_S has n states and r registers, and if Ω is a parity condition of index d , it is recognised by a deterministic parity automaton with $2^{\mathcal{O}(n \cdot 2^{(r+k)^2})}$ states and $\mathcal{O}(d \cdot n \cdot 2^{(r+k)^2})$ colours.

Proof. By Equation 6.2, we know that $W_S^k = \left(\text{lab} \left((A_k \otimes S^c) \cap L_k \right) \right)^c$. The construction is the same as Lemma 6.15, we only need to redo the complexity analysis. Complementation is done in the same way, word product as well, and idem for the intersection operation, so we get a non-deterministic register automaton with states Q' of size $|Q'| = n \cdot (2^k + 1)$

[36]: Khalimov, Maderbacher, and Bloem (2018), 'Bounded Synthesis of Register Transducers'

[35]: Khalimov and Kupferman (2019), 'Register-Bounded Synthesis'

[41]: Exibard, Filiot, and Reynier (2021), 'Synthesis of Data Word Transducers'

[89]: Calude et al. (2017), 'Deciding parity games in quasipolynomial time'

[88]: Zielonka (1998), 'Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees', recalled in Theorem 3.12

and registers R' of size $|R'| = r + k$ that recognises $(A_k \otimes S^c) \cap L_k$. What changes is the projection over labels. By Proposition 4.35 in Section 4.6, we know that $\text{lab} \left((A_k \otimes S^c) \cap L_k \right)$ is recognised by a non-deterministic ω -automaton A^{lab} with states $P \times \text{Constr}_{\mathbb{Q}}(R')$ and acceptance condition $\pi_1^{-1}(Q')$. By Definition 4.6, we know that the size of $[R']_{\mathbb{Q}}$ is bounded by $2^{(r+k+1)^2}$, so the ω -automaton that we have built so far has $n \cdot (2^k + 1) \cdot 2^{(r+k+1)^2} = \mathcal{O}(n \cdot 2^{\text{poly}(r,k)})$ states and $d + 1$ priorities. It remains to complement it; to that end, we need to determinise it. Overall, this yields a parity automaton with $2^{\mathcal{O}(n^2 2^{\text{poly}(r,k)})}$ states and $\mathcal{O}((d + 1) \cdot n \cdot 2^{\text{poly}(r,k)})$ priorities. $\square$

The complexity analysis of Theorem 6.14 again applies to the case of $(\mathbb{Q}, <, 0)$, so we get:

Theorem 6.16 *The register-bounded synthesis problem for specifications expressed as universal register automata over $(\mathbb{Q}, <, 0)$ is in 2-ExpTime .*

The study can be extended to any dense order without endpoints, since it is isomorphic to $(\mathbb{Q}, <, 0)$ [126, Theorem 27] (the result is originally due to Cantor).

6.2.3. Allowing Non-Deterministic Reassignment

In Section 4.10.1, we pointed out that the projection over labels of register automata with guessing over $(\mathbb{D}, =, \mathbb{C})$ is effectively ω -regular, since they also enjoy the renaming property [38, Proposition 14] (see also Proposition 4.58). Moreover, the construction to recognise the projection is actually the same as without guessing. As a consequence, we can adapt the proof of Theorem 6.14 to allow non-deterministic reassignment:

Theorem 6.17 *The register-bounded synthesis problem for specifications expressed as universal register automata with guessing over $(\mathbb{D}, =, \mathbb{C})$ is in 2-ExpTime .*

In Part II, we demonstrate that this is also the case for $(\mathbb{Q}, <, 0)$ (Corollary 12.25, see also Proposition 4.59). Since the construction is again the same as without guessing, we obtain:

Theorem 6.18 *The register-bounded synthesis problem for specifications expressed as universal register automata with guessing over $(\mathbb{Q}, <, 0)$ is in 2-ExpTime .*

Remark 6.13 In Section 4.10.2, we also note that the projection over labels is again effectively ω -regular for fresh-register automata, so we could get a similar result for universal fresh-register automata. This is however less interesting, since the fresh predicate can be simulated with a universal register automaton with one register (see Remark 4.25).

This is a coarse upper bound, but it does not affect the complexity class.

We implicitly use the fact that $n \log n \leq n^2$, which yields a simpler expression. Note that $\text{poly}(r, k)$ means that there exists a polynomial $P(r, k)$ such that the bound holds, but it does not necessarily denote a fixed polynomial.

Constraints are bigger over $(\mathbb{Q}, <, 0)$, but this does not change the complexity class.

In a densely ordered set $(\mathbb{D}, <)$, an *endpoint* is either a lower or an upper bound, i.e. an element such that $\forall e \in \mathbb{D}, e \leq d \Rightarrow e = d$ (respectively $e \geq d \Rightarrow e = d$).

[126]: Roitman (1990), *Introduction to Modern Set Theory*

[38]: Kaminski and Zeitlin (2010), 'Finite-Memory Automata with Non-Deterministic Reassignment'

6.3. Specifications Expressed as Non-Deterministic Register Automata

6.3.1. The Case of Equality

Unfortunately, bounding the number of registers of the target implementation does not allow to recover decidability for specifications expressed as non-deterministic register automata, as the problem of whether a non-deterministic register automaton accepts all finite data words is undecidable. This is the case already over data domains with equality only.

Theorem 6.19 ([41, Theorem 3.2]) *The register-bounded synthesis problem from specifications expressed as non-deterministic register automata over $(\mathbb{D}, =, \#)$ is undecidable, already if the bound $k = 1$.*

This is the case even if we know that the specification given as input has total domain.

Proof. We reduce the problem from the universality problem of non-deterministic register automata over finite data words, with data domain $(\mathbb{D}, =, \#)$. This problem is undecidable [101, Theorem 18]. Let A be such an automaton. If we do not assume that the domain of the specification is total, then the result is directly obtained by asking whether $S = \text{id}_{L(A)}$ is realisable by a transducer with one register: if $L(A) = (\Sigma \times \mathbb{D})^\omega$, then $\text{id}_{L(A)}$ can trivially be implemented (even with a single register), otherwise $\text{dom}(S) \subsetneq (\Sigma \times \mathbb{D})^\omega$ so S is not realisable, as transducers have total domain by definition.

Now, let us show the stronger result, namely that the above problem is undecidable even if we know that the specification given as input has total domain (which is an undecidable property, cf Theorem 8.1). For simplicity, we assume that A has a label alphabet Σ with at least two letters. We define a specification S_A which starts by reading a finite data word w , then a separator $\$$ (its associated data value is arbitrary and not represented). Afterwards, it allows for two behaviours:

- Swapping the labels of the first and second labelled data that are read, then accept any behaviour. Note that this does not depend on w (cf specification fragment S_{swap}).
- Behaving like the identity function, but *only* whenever $w \in L(A)$ (cf specification fragment $S_{L(A)}$).

Finally, we complete the domain of S_A by allowing any behaviour on input data words which do not contain a $\$$ (cf specification fragment T). Formally, $S_A = S_{\text{swap}} \cup S_{L(A)} \cup T$, where:

$$S_{\text{swap}} = \left\{ \left(\begin{array}{l} w\$(\sigma_1, d_1)(\sigma_2, d_2)u, \\ w\$(\sigma_2, d_1)(\sigma_1, d_2)v \end{array} \right) \mid \begin{array}{l} w \in (\Sigma \times \mathbb{D})^* \\ \sigma_1, \sigma_2 \in \Sigma, d_1, d_2 \in \mathbb{D} \\ u, v \in (\Sigma \times \mathbb{D})^\omega \end{array} \right\}$$

$$S_{L(A)} = \left\{ (w\$u, w\$v) \mid \begin{array}{l} w \in L(A) \\ u, v \in (\Sigma \times \mathbb{D})^\omega \end{array} \right\}$$

$$T = \{(w, w) \mid w \notin (\Sigma \times \mathbb{D})^*\$ (\Sigma \times \mathbb{D})^\omega\}$$

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

We explain at the end of the proof how to lift the assumption that $|\Sigma| \geq 2$.

We say *specification fragment* to highlight the fact that the corresponding relation does not have a total domain, and only consists in a part of the sought specification, which is later on obtained by taking the union of the specification fragments.

Recall that register transducers necessarily implement total functions, so a specification whose domain is not total is never realisable.

We now show that S is definable by a non-deterministic register automaton. As those automata are closed under union (Proposition 4.2), it suffices to show that each fragment is NRA-definable. First, recognising the interversion of the first two labels σ_1 and σ_2 is easily done using non-determinism, and the behaviour on the data part is definable as well (the identity, then allow any output), so S_{swap} is NRA-definable. Second, simulating A then allowing any behaviour ($S_{L(A)}$) is also NRA-definable. Finally, T is an ω -regular property, so it is NRA-definable.

Now, if A accepts all inputs, i.e. $L(A) = (\Sigma \times \mathbb{D})^\omega$, then the identity function $\text{id}_{(\Sigma \times \mathbb{D})^\omega}$ realises S , since in that case $\text{id}_{(\Sigma \times \mathbb{D})^\omega} \subseteq S_{L(A)} \cup T \subseteq S_A$. This function can be implemented with a register transducer with one state and one register.

Conversely, if $L(A) \subsetneq (\Sigma \times \mathbb{D})^\omega$, assume by contradiction that S is realisable by a register transducer I . Let $w \in (\Sigma \times \mathbb{D})^\omega \setminus L(A)$. For any $(\sigma_1, d_1)(\sigma_2, d_2)u \in (\Sigma \times \mathbb{D})^\omega$, we must have $\llbracket I \rrbracket(w\$(\sigma_1, d_1)(\sigma_2, d_2)u) = w\$(\sigma_2, d_1)(\sigma_1, d_2)v$ for some $v \in (\Sigma \times \mathbb{D})^\omega$. This implies guessing the second label while having only read the first one, which is not doable by any sequential transducer as soon as $\sigma_1 \neq \sigma_2$.

Finally, we can actually lift the assumption that the labels alphabet contains at least two letters. In the above, the crux of the proof relies on the fact that S_{swap} is not realisable. We picked the classical swapping of letters due to its simplicity, but this can be done using any specification that is not realisable. One can for instance take the data specification of Figure 6.1, and adapt the above proof. $\square$

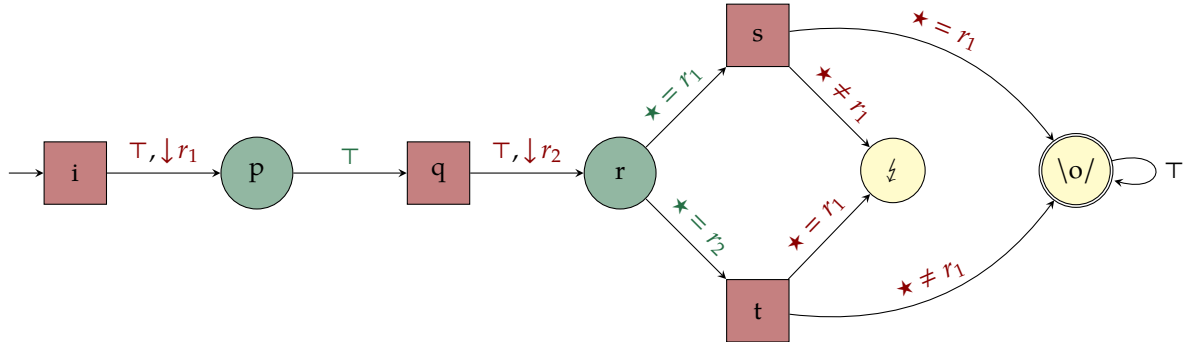


Figure 6.1: A (deterministic) specification register automaton whose specification is not realisable by any register transducer, nor any synchronous program. Initially, the environment inputs two data values, that are respectively stored in r_1 and r_2 (the output of the system in between, from p to q is irrelevant). Then, the system must guess whether the next input data value is equal to r_1 or not, and output r_1 (transition to s) or r_2 (transition to t) accordingly. If she guessed wrong, the specification automaton transitions to state ⚡ , which is an output state with no outgoing transitions, so it is rejecting. If the guess was right, the automaton transitions to the state ||o|| , which is a short for a pair of sink input and output accepting states. No synchronous program can make this guess correctly without knowing the third input data value, so this specification is not realisable by any synchronous program.

In the proof, we construct, given a non-deterministic register automaton over finite data words, a specification which is realisable by a transducer with one register if and only if A accepts all finite data words. However, one can notice that it is realisable by a transducer with one register if and only if it is realisable by any register transducer.

Theorem 6.20 ([41, Theorem 3.1]) *The (unbounded) Church synthesis problem for specifications given as non-deterministic register automata over $(\mathbb{D}, =, \#)$ is undecidable, even if we restrict to specifications with total domain.*

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

Note that the above reasoning actually even works for any synchronous program, as finite memory and the use of registers does not play a role (what matters is that the implementation is required to know what comes next, which is not doable synchronously). As a consequence, we get

Theorem 6.21 *The reactive synthesis problem for specifications given as non-deterministic register automata over $(\mathbb{D}, =, \#)$ is undecidable.*

As any data domain contains at least equality, the three above theorems can also be generalised to richer data domains.

Theorem 6.22 *The (unbounded) Church synthesis problem for specifications given as non-deterministic register automata over some data domain $\mathcal{D}$ is undecidable, even if we restrict to specifications with total domain.*

Remark 6.14 Note that this ‘monotonicity’ is not automatic: sometimes, enriching the implementation formalism (or both the specification and implementation formalisms, as when we consider richer data domains) allows to recover decidability. This is the case for our study in Part II, where, for asynchronous functional specifications, it is undecidable whether they can be implemented by an asynchronous sequential transducer, but one can decide whether it can be implemented by some asynchronous program.

Why does the method for URA not allow to conclude? Let us examine more precisely where the reasoning that we developed for universal register automata fails for non-deterministic ones. Recall that by Lemma 6.10, we have that

$$W_S^k = \left(\text{lab}(L_{S^c, k}) \right)^c$$

where

$$L_{S, k} = \left\{ \alpha \otimes w \mid \begin{array}{l} \alpha \in A_k^\omega, w \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \\ w \in \text{CompIO}(\alpha) \text{ and } w \in S \end{array} \right\}$$

By mimicking the reasoning of Lemma 6.13, we get that $L_{S, k}$ is definable by a universal register automaton (remember that there is a complement operation in the process). However, this does not allow to conclude, because the projection over labels of the language of a URA is not always ω -regular (Property 4.49). Even more so, these automata can simulate Turing machines, as shown in [101, Theorem 5.1] (see also Theorem 4.50).

An intuition on why the problem is harder (actually, undecidable: Theorem 6.19) for non-deterministic specification register automata is that there is one additional quantifier alternation: for universal specification register automata, Eve has to ensure that for all action sequences provided by Adam, *all* runs over all compatible data words are accepting. To the contrary, for non-deterministic specification register automata, she only has to guarantee that for all action sequences and all compatible data words, *there exists* an accepting run.

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

6.3.2. Test-free Non-deterministic Register Automata

To recover decidability in the non-deterministic case, we consider a subclass of register automata that we call test-free nondeterministic register automata. These automata do not perform tests on input data, and we additionally ask that on output transitions, they only allow data that are equal to the content of some register, as if they were non-deterministic transducers.

This restriction is inspired from [45], which defines transformations of data words using MSO interpretations with an MSO origin relation. The MSO interpretation describes the transformation over labels (called the *string transduction*), as in [127], while the MSO origin relation describes the relation between input and output data. This relation does not depend on (un)equalities between different input data: it uniquely maps each output position to an input position, expressing that the output data value at this position is equal to the corresponding input data value. They show that this model is equivalent to two-way deterministic transducers with *data variables*, themselves equivalent to one-way streaming string transducers with data variables and parameters. These parameters are reminiscent of the guessing mechanism described in [38] (see Section 4.10.1). The data variables are used to implement the MSO origin relation: they are registers in which the transducer can store the input data values and output them, but it is not allowed to perform any test on the stored data, contrary to our model of register automata.

This is what we call the test-free restriction, that we apply to our one-way model of register automata: they correspond to *non-deterministic one-way* transducers with data variables. These machines can only rearrange input data (duplicate, erase, move) regardless of the actual data values, as there are no tests. This way, as stated in Proposition 6.23, registers induce an origin relation between input and output data.

To avoid confusion between the nature of specifications and implementations, we prefer to define them as restricted register automata, instead of transducers.

Definition 6.7 (Test-free register automaton) A non-deterministic register automaton is *test-free* if:

1. Its input transitions do not depend on tests over input data: for all input transitions $t = p \xrightarrow{\sigma, \phi, \text{asgn}} q \in \Delta_i$, we have $\phi = \top$.
2. Its output transitions consist in outputting the content of some register: for all output transitions $t' = q \xrightarrow{\gamma, \psi, \text{asgn}'} p \in \Delta_o$, we have $\psi = r^=$ for some $r \in R$. Up to remembering additional information in the states, we moreover assume that $\text{asgn}' = \emptyset$ (see Proposition 4.8).

Note that since there are no tests, the choice of the data domain is irrelevant.

We now make the connection with the notion of origin precise: as highlighted in [128, Section 5], one can encode origin graphs using data words. Our encoding is slightly different, as the data values are not ordered, and the order on positions is instead given by the positions in the data word, but the idea is the same: if the input data are all

[45]: Durand-Gasselin and Habermehl (2016), ‘Regular Transformations of Data Words Through Origin Information’

[127]: Courcelle (1994), ‘Monadic Second-order Definable Graph Transductions: A Survey’

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

[128]: Dartois, Filiot, and Lhote (2018), ‘Logics for Word Transductions with Synthesis’

pairwise distinct, and if moreover all output data appear in the input, the occurrence of a data value *originates* from its unique occurrence on the input.

Let $\rho = p^0 \xrightarrow{\sigma_0, \top, \text{asgn}_0} q^0 \xrightarrow{\gamma_0, \star=r_0, \emptyset} p^1 \dots \in (Q\Delta)^\omega$ be a run of some test-free non-deterministic register automaton. Its *associated origin function* is

$$o_\rho : j \in \mathbb{N} \mapsto \max\{i \leq j \mid r_j \in \text{asgn}_i\}$$

with the convention that $\max \emptyset = -1$. Thus, $o_\rho(j)$ is the last input position at which the register that is output at position j was assigned, where -1 means that it contains its initial value $\#$. As a consequence, the data value output at position j is equal to the data value at input position $o_\rho(j)$, with the convention that $\text{dt}(w[-1]) = \#$ for any data word w .

Now, for an origin function $o : \mathbb{N} \rightarrow \mathbb{N}$ and a pair of data words $(u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$, we say that (u, v) is *compatible* with o , denoted $(u, v) \models o$, whenever for all $j \in \mathbb{N}$, $\text{dt}(v[j]) = \text{dt}(u[o(j)])$.

The following proposition expresses that the actual data values in a pair of data word (u, v) do not matter with respect to being accepted by some test-free register automaton, only the compatibility with origin functions does. Recall that $L_{\forall\#}$ is defined in Example 4.1 as the data language of words whose data values are pairwise distinct: $L_{\forall\#} = \{d_0 d_1 \dots \in \mathbb{N}^\omega \mid \forall i \neq j, d_i \neq d_j\}$.

Proposition 6.23 *Let $(u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$, and ρ a run of some test-free register automaton. Then:*

- (i) *If ρ is a run over (u, v) , then $(u, v) \models o_\rho$*
- (ii) *If ρ is a run over (u, v) and $\text{dt}(u) \in L_{\forall\#}$, then for all $o : \mathbb{N} \rightarrow \mathbb{N}$, $(u, v) \models o \Leftrightarrow o = o_\rho$*
- (iii) *If (u, v) and ρ have the same labels and if $(u, v) \models o_\rho$, then ρ is a run over (u, v)*

Proof. Items (i) and (iii) are easily shown by induction on the length of a partial run, as they follow from the semantics of test-free NRA. The $\Leftarrow$ direction of item (ii) is exactly item (i). Now, assume that $(u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ is such that $\text{dt}(u) \in L_{\forall\#}$. Let $j \in \mathbb{N}$ be such that $\text{dt}(v[j]) = \text{dt}(u[o(j)])$. By item (i), we know that $\text{dt}(v[j]) = \text{dt}(u[o_\rho(j)])$, so $\text{dt}(u[o(j)]) = \text{dt}(u[o_\rho(j)])$. Since $\text{dt}(u) \in L_{\forall\#}$, this implies $o(j) = o_\rho(j)$. Overall, $o = o_\rho$. $\square$

It is not clear whether the specification $W_{S,k}$ associated with the $\mathcal{T}_{R_k}^{\text{SA=}}$ template is regular for test-free specifications, but we show that it suffices to consider another set denoted $W_{S,k}^{\text{tf}}$ which is easier to analyse (and can be proven ω -regular), which describes the behaviour of S over input with pairwise distinct data values. Indeed, test-free automata cannot conduct test on input data, so they behave the same on an input word whose data values are pairwise distinct, and this choice ensures that two equal input data values do not ease the task of the implementation (Proposition 6.23). An interesting side-product of this approach is that it implies that we can restrict to implementations computed by test-free register transducers, i.e. transducers whose transitions do not depend on tests over input data values.

Recall that $\mathcal{T}_R^{\text{SA=}}$ is the single-assignment template (Proposition 6.6), which is sound and complete for transducers with registers R .

Definition 6.8 (Test-free register transducer) A sequential register transducer $T = (Q, q^i, \Sigma, \Gamma, R, \delta)$ is *test-free* if its transition function is of type $\delta : Q \times \Sigma \times \{\top\} \rightarrow 2^R \times \Gamma \times R$, i.e. its tests are restricted.

Proposition 6.24 Let S be a test-free specification, $k \geq 1$ and $A_i^{\text{ff}} = \Sigma \times \{\text{locFresh}_{R_k}\}$. The following are equivalent:

Technically, A_i^{ff} depends on k . We omit it to lighten the notations.

1. S is realisable by a register transducers with k registers
2. S is realisable by a test-free transducer with k registers
3. The following (finite alphabet) specification is realisable by a (register-free) transducer with input alphabet A_i^{ff} :

$$W_{S,k}^{\text{ff}} = \{\beta \in (A_i^{\text{ff}} A_o^k)^{\omega} \mid \text{there exists } (u, v) \in \text{CompIO}(\beta) \cap S \text{ such that } \text{dt}(u) \in L_{\forall \neq}\}$$

Proof. $2 \Rightarrow 1$ is trivial.

Now, assume that S is realisable by some transducer with k registers. The idea is to generalise its behaviour when it only reads locally (or globally) fresh data values, to the whole data domain, which is enough to satisfy the specification thanks to Proposition 6.23. Thus, consider the specification $W_{S,k}$ associated with S for the template $\mathcal{T}^{\text{SA}=\}$ (or any other complete template that contains locFresh_R). By Theorem 6.8 (or, more precisely, Lemma 6.7), it is realisable by some (register-free) transducer T . Now, since transducers are closed under alphabet restriction, $W_{S,k}^{\text{locFresh}} = W_{S,k} \cap (A_i^{\text{ff}} A_o^k)^{\omega}$ is realisable by T restricted to the input alphabet A_i^{ff} . Moreover, $W_{S,k}^{\text{locFresh}} \subseteq W_{S,k}^{\text{ff}}$. Indeed, let $\beta \in W_{S,k}^{\text{locFresh}}$. Then, $\text{CompIO}(\beta) \subseteq S$. As all tests over input data values consist in locFresh , β is in particular compatible with an input data word $u \in (\Sigma \times \mathbb{D})^{\omega}$ whose data values are pairwise distinct, i.e. $\text{dt}(u) \in L_{\forall \neq}$, associated with some output $v \in (\Gamma \times \mathbb{D})^{\omega}$. Thus, $(u, v) \in \text{CompIO}(\beta) \cap L_{\forall \neq}$, so $\text{CompIO}(\beta) \cap S \cap L_{\forall \neq} \neq \emptyset$. This means that $W_{S,k}^{\text{ff}}$ is realisable by any sequential transducer realising $W_{S,k}^{\text{locFresh}}$.

Recall that locFresh_R is defined as $\text{locFresh}_R := \bigwedge_{r \in R} \star \neq r$, see Notation 4.17.

$3 \Rightarrow 2$: finally, assume $W_{S,k}^{\text{ff}}$ is realisable by some transducer T . We show that T , when ignoring the locFresh input tests, is actually an implementation of S . Thus, let T' be the same transducer as T except that all input tests locFresh have been replaced with $\top$. Formally, $p \xrightarrow[T']{\sigma, \top | \text{asgn}, \gamma, r}$

q iff $p \xrightarrow[T]{\sigma, \text{locFresh} | \text{asgn}, \gamma, r} q$. Note that T' , interpreted as a register transducer, is test-free. Let $u \in (\Sigma \times \mathbb{D})^{\omega}$, and $\beta_i = \text{lab}(u) \otimes (\text{locFresh})^{\omega}$ be the input action in A_i^{ff} with same labels as u . Let $\beta_o = T'(\beta_i)$, and $\beta = \langle \beta_i, \beta_o \rangle$ be their interleaving. Finally, let $(u', v') \in \text{CompIO}(\beta) \cap S \cap L_{\forall \neq}$ (such an (u', v') exists because, as above, $\text{CompIO}(\beta) \cap L_{\forall \neq} \neq \emptyset$). Then, since $\text{lab}(w) = \text{lab}(w')$, they admit the same run $\rho^{T'}$ in T' , so $w, w' \models o_{\rho^{T'}}$. Now, $w' \in S$, so it admits an accepting run ρ^S in S , which implies $w' \models o_{\rho^S}$. Moreover, $w' \in L_{\forall \neq}$ so, by Proposition 6.23 (ii), we get $o_{\rho^i} = o_{\rho^S}$. Therefore, $w \models o_{\rho^S}$, so, by (iii), w admits ρ^S as a run, i.e. $w \in S$. Overall, $\llbracket T' \rrbracket \subseteq S$, meaning that T' is a (test-free) implementation of S . $\square$

We are now ready to show that the register-bounded synthesis problem is decidable for test-free specifications.

Theorem 6.25 ([41, Theorem 4.11]) *The register-bounded synthesis problem for specifications given by test-free register automata is in 2-ExpTIME.*

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

Since the specification is test-free, the choice of the data domain does not matter, so we omit it from the statement.

Proof. Let S be a test-free register automaton with n states and r registers recognising some specification $\llbracket S \rrbracket$. By Proposition 6.24, $\llbracket S \rrbracket$ is realisable by a transducer with k registers if and only if the following (finite alphabet) specification is ω -regular:

$$W_{S,k}^{\text{tf}} = \left\{ \beta \in (A_i^{\text{tf}} A_o^k)^\omega \mid \begin{array}{l} \text{there exists } (u, v) \in \text{CompIO}(\beta) \cap \llbracket S \rrbracket \\ \text{such that } \text{dt}(u) \in L_{V \neq} \end{array} \right\}$$

First, let S^{tf} be the same automaton as S except that all input transitions $p \xrightarrow{\sigma, \tau, \text{asgn}} q$ are replaced with $p \xrightarrow{\sigma, \text{locFresh}_{R_k}, \text{asgn}} q$. For all $\beta \in (A_i^{\text{tf}} A_o^k)^\omega$, there exists $u \in \text{CompIO}(\beta) \cap \llbracket S \rrbracket$ such that $\text{dt}(u) \in L_{V \neq}$ if and only if there exists $u \in \text{CompIO}(\beta) \cap \llbracket S^{\text{tf}} \rrbracket$: the $\Rightarrow$ direction is trivial, and the $\Leftarrow$ stems from the fact that an input in $L_{V \neq}$ only takes transitions with $\phi = \text{locFresh}$. As a consequence, we have

$$W_{S,k}^{\text{tf}} = \{ \beta \in (A_i^{\text{tf}} A_o^k)^\omega \mid \text{CompIO}(\beta) \cap \llbracket S^{\text{tf}} \rrbracket \neq \emptyset \}$$

Then, the following data language is definable by a non-deterministic register automaton over data domain $(\mathbb{D}, =, \#)$:

$$L_{S,k}^{\text{tf}} = \left\{ (u, v) \otimes \beta \mid \begin{array}{l} \beta \in (A_i^{\text{tf}} A_o^k)^\omega \\ (u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \\ (u, v) \in \text{CompIO}(\beta) \cap S^{\text{tf}} \end{array} \right\}$$

Indeed, by Lemma 6.11,

$$L_k^{\text{tf}} = \left\{ \beta \otimes (u, v) \mid \begin{array}{l} \beta \in (A_i^{\text{tf}} A_o^k)^\omega \\ (u, v) \in (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega \\ (u, v) \in \text{CompIO}(\beta) \end{array} \right\}$$

is definable by a NRA A_k^{tf} of size exponential in k , so its product with S^{tf} recognises $L_{S,k}^{\text{tf}}$. Finally, $W_{S,k}^{\text{tf}} = \text{lab}(L_{S,k}^{\text{tf}})$, which is ω -regular (Proposition 4.19). More precisely, it is recognised by a non-deterministic ω -automaton that consists in extending the state space of $S^{\text{tf}} \otimes A_k^{\text{tf}}$ with constraints, that has $n(2^k + 1)B_{r+k}$ states. By Theorem 3.15, solving the synthesis problem for $W_{S,k}^{\text{tf}}$ is in ExpTime , which yields a 2-ExpTime complexity upper bound for the synthesis problem for S , since $W_{S,k}^{\text{tf}}$ is of size exponential in k . $\square$

Unbounded Synthesis of Register Transducers

7.

In this chapter, we study the (unbounded) Church synthesis problem, where we do not bound the number of registers of the implementation, as well as the reactive synthesis problem, which targets any synchronous implementation (i.e. it can manipulate data values with arbitrary computational power, with no constraints to use registers nor to have finitely many control states). The choice of starting with the study of the register-bounded case is motivated by the fact that most results are negative in the general setting, that we call ‘unbounded’ to make the distinction clear. Indeed, already over data domains with equality only, the Church synthesis problem is undecidable, both for specifications expressed by non-deterministic and universal register automata. We start by formally establishing these two results (Theorems 6.20 and 7.1).

Then, we shift to the case of specifications given by deterministic register automata. We introduce the automaton game, which consists in interpreting the specification automaton as an arena. Contrary to the finite alphabet case, this does not always yield a sound nor complete abstraction; we investigate when this is the case through the notions of game-soundness and game-completeness. For data domains $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, 0)$, this yields an ExpTime decision procedure for the reactive synthesis problem (Theorems 7.15 and 7.16) from deterministic specification automata; we additionally show that it coincides with the Church synthesis problem over $(\mathbb{D}, =, \mathbb{C})$ (again Theorem 7.15) and for one-sided specifications over $(\mathbb{Q}, <, 0)$ (Theorem 7.17). Informally, a specification is one-sided when the system picks her output data values among those that are present in the registers. This corresponds to the single-sided restriction of [33], in the case where it is the protagonist who is restricted to a Boolean set.

The case of $(\mathbb{N}, <, 0)$ is more intricate: for two-sided specifications, the reactive synthesis problem is undecidable (Theorem 7.19), since alternation between the system and the environment allows to simulate Minsky machines: one player suggests the value of an increment or a decrement, and the other checks that she does not cheat. Decidability is recovered for one-sided specifications. The automaton game is not ω -regular, so it does not suffice to use the above results. However, through a careful analysis of feasible action sequences in $(\mathbb{N}, <, 0)$, we show that one can consider an ω -regular approximation of the automaton game; as a side-product, we get that it suffices to consider register transducer implementations.

Related Publications Sections 7.1 and 7.2 describe work that was published in the conference paper [37], later on extended to [41] that includes full, simplified proofs, again reworked in this document. Section 7.3 gathers results from [37, 41] (subsections 7.3.1 and 7.3.2); Section 7.3.4 describes a contribution that appeared as [42], dedicated to the case of $(\mathbb{N}, <, 0)$. Both works have been partially factored together with the introduction of the notion of automaton game, which allows to give general conditions under which one can finitely abstract the synthesis problem for specifications given by deterministic register automata through the

[33]: Figueira, Majumdar, and Praveen (2020), ‘Playing with Repetitions in Data Words Using Energy Games’

It is the topic of a separate publication, cf infra.

We use the term two-sided in contrast with that of one-sided: a specification is two-sided if it is not necessarily one-sided.

[37]: Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Ed. by Wan Fokkink and Rob van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019

[41]: Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *Logical Methods in Computer Science* 17.1 (2021)

[42]: Léo Exibard, Emmanuel Filiot, and Ayrat Khalimov. ‘Church Synthesis on Register Automata over Linearly Ordered Data Domains’. In: *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany*. Ed. by Benjamin Monmege and Markus Bläser. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021

related notion of game-readiness. While [37] and [41] were focused on the case of one-sided specifications over $(\mathbb{D}, =, \#)$, we extend the study to the case of two-sided specifications over $(\mathbb{D}, =, \mathbb{C})$ and to the data domain $(\mathbb{Q}, <, 0)$ (Section 7.3.3), that almost comes for free thanks to the above development.

Summary

7.1. Specifications Expressed by Non-Deterministic Register Automata

Our study of the register-bounded synthesis problem for specifications expressed by non-deterministic register automata led us to the conclusion that this formalism is too expressive, independently of the implementation formalism as soon as it contains register transducers with one register. In particular:

Theorem 6.22 *The (unbounded) Church synthesis problem for specifications given as non-deterministic register automata over some data domain $\mathcal{D}$ is undecidable, even if we restrict to specifications with total domain.*

See also Theorems 6.19, 6.20 and 6.21.

Subsequently, if we want to get a decidable synthesis problem for non-deterministic, we need to limit the expressiveness of the specification formalism. A good candidate is that of test-free non-deterministic register automata (Section 6.3.2). However, this problem is still open as of now:

Open Problem 7.1 The decidability status of the Church synthesis problem for specifications expressed by test-free non-deterministic register automata is unknown.

7.2. Specifications Expressed by Universal Register Automata

Now, we show that the Church synthesis problem is also undecidable for specifications expressed by universal register automata over data domains with equality only, answering a question left open in [36]. The proof is more involved than for the non-deterministic case, since then register-bounded synthesis was already undecidable.

Theorem 7.1 ([41, Theorem 3.3]) *The Church synthesis problem for specifications expressed by universal register automata over $(\mathbb{D}, =)$ is undecidable, even if we restrict to specifications with total domain.*

[36]: Khalimov, Maderbacher, and Bloem (2018), ‘Bounded Synthesis of Register Transducers’

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

Proof. We present a reduction from the emptiness problem of universal register automata over finite words. This problem is undecidable by a direct reduction from the universality problem of non-deterministic register automata, which is undecidable [101, Theorem 18].

In the following, $\$$ denotes a distinguished data value which does not appear in the rest of the input (i.e. $\$ \notin \mathbb{D}$). It can be simulated either

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

by a label, or by considering the data domain $\mathbb{D}^\$$ with an additional data value $\$$ and a corresponding constant symbol. In the following, we simply assume it is given, to reduce notational overhead.

A specification which is realisable only when the input contains finitely many distinct data values First, consider the fragment of specification

$$S_\infty = \left\{ (u\$v, u\$w) \mid \begin{array}{l} u \in \mathbb{D}^*, v \in \mathbb{D}^\omega, w \in \mathbb{D}^\omega \\ \text{each data value of } u \\ \text{appears infinitely often in } w \end{array} \right\}$$

S_∞ is recognised by a universal register automaton with one register: upon reading a data value d in u , it stores it in its register and checks that it appears infinitely often in w using a Büchi condition.

Now, for $k \geq 1$, define the specification fragment

$$S_{\leq k} = \left\{ (u\$v, u\$w) \mid \begin{array}{l} u \in \mathbb{D}^*, v \in \mathbb{D}^\omega, w \in \mathbb{D}^\omega \\ u \text{ contains at most } k \text{ distinct data} \end{array} \right\}$$

We also define the two specification fragments $S_{>k}$ (u contains more than k distinct data) and T (the input does not contain $\$$) to get a total domain; both allow any behaviour on the output:

$$S_{>k} = \left\{ (u\$v, w) \mid \begin{array}{l} u \in \mathbb{D}^*, v, w \in \mathbb{D}^\omega \\ u \text{ contains more than } k \text{ distinct data} \end{array} \right\}$$

$$T = \{(u, w) \mid u \notin \mathbb{D}^*\mathbb{D}^\omega, w \in \mathbb{D}^\omega\}$$

For all $k \geq 1$, $(S_\infty \cap S_{\leq k}) \sqcup S_{>k} \sqcup T$ is realisable by a transducer with k registers. On reading u , the transducer stores each distinct data value in one register. If it encounters more than k distinct data, it simply copies its input (in that case, any behaviour is allowed). Otherwise, once it reads the $\$$, it outputs the content of its registers in a round-robin fashion (r_1 , then r_2 , etc until r_k , then repeat from r_1).

However, S_∞ is not realisable: on reading the $\$$ separator, any implementation must have all the data of u in its registers, but the number of data of u is not bounded: u can have pairwise distinct data and be of arbitrary length.

The candidate specification Now, let A be a universal register automaton over finite data words. Consider the specification $S = S_\infty \sqcup T \cup S_{L(A)}$, where

$$S_{L(A)} = \{(u\$v, u\$w\$^\omega) \mid u \in \mathbb{D}^*, v \in \mathbb{D}^\omega, w \in L(A)\}$$

S has total domain, and is recognisable by a universal register automaton. Indeed, these automata are closed under union, by the same product construction as for the intersection of non-deterministic ones [21, Theorem 3] (see also Proposition 4.3). Moreover, each part is recognisable by a URA. S_∞ is, as described above. $S_{L(A)}$ consists in simulating A on the output to check that $w \in L(A)$ (which is doable as A is a universal register

[21]: Kaminski and Francez (1994), ‘Finite-Memory Automata’

automaton) and accepting $\$^\omega$ on the output only if in a final state. Finally, T simply checks an ω -regular property.

Correctness of the reduction Now, if $L(A) \neq \emptyset$, then let $w \in L(A)$, and let $D_w = \{d \in \mathbb{D} \mid w[i] = d \text{ for some } 0 \leq i < |w|\}$ be the set of data that appear in w . We also let $k = |D_w|$ be the number of distinct data. S is realisable by a transducer with $k + 1$ registers. Its set of registers is $R \sqcup \{r_c\}$, where $|R| = k$. We let $\pi : D_w \rightarrow R$ be some bijection (its choice does not matter). Before reading the $\$$, the transducer stores the k first distinct data in its registers in R , and uses its additional register r_c to copy its input. We denote by $\nu : R \sqcup \{r_c\} \rightarrow \mathbb{D}$ the valuation of its registers at the moment it reads the $\$$ if such symbol occurs in the input. If it never met, the transducer simply keeps copying its input and hence satisfies the specification. Now, assume there is a $\$$ in the input.

- When the number of data in u is lower than or equal to k , the transducer realises S_∞ , in the way that we explained before.
- When it is greater than k , it outputs $u\$v(\pi(w))\$^\omega$. As data languages recognised by register automata are closed under automorphisms (Proposition 4.10), we know that $v(\pi(w)) \in L(A)$.

Conversely, if $L(A) = \emptyset$, then $S_{L(A)} = \emptyset$ and $S = S_\infty \sqcup T$ is not realisable. If it were, then in particular the implementation would comply with S_∞ on inputs containing a $\$$. However, we have seen above that this is not the case. Thus, S is realisable iff $L(A) \neq \emptyset$, which concludes the proof. $\square$

7.3. Specifications Expressed as Deterministic Register Automata

As the Church synthesis problem is undecidable for both semantics, even for the most elementary data domain, we need to consider restrictions of the problem. The case of specifications expressed by deterministic register automata yields positive results. Note that there is a subtlety here, as the term of *deterministic* might be misleading: obviously, we do not assume that the specification is recognised by a sequential (also known as ‘input-deterministic’) transducer, otherwise the problem is trivial. Instead, we ask that the specification automaton is deterministic when it reads both inputs and outputs. Still, many outputs can be associated with a single input; for instance, the specification that accepts all data words is easily recognisable by a deterministic specification automaton.

7.3.1. The Automaton Game

Recall that when it is deterministic, a specification ω -automaton S naturally induces a game: its transition system is an arena, and the winning condition is its acceptance condition. A winning strategy in this game then corresponds to an implementation of S ; the implementation is moreover finite-memory when the strategy is (Proposition 3.13). We extend this idea to specification register automata.

Formally, let $S = (Q, q^i, \Sigma, \Gamma, R, \delta, \Omega)$ be a deterministic specification automaton. Its states are partitioned into $Q = Q_i \sqcup Q_o$, where $q^i \in Q_i$, and

its transition function is $\delta = \delta_i \sqcup \delta_o$, where $\delta_\sigma : Q_\sigma \times \Sigma \times \text{Tests}_{\mathcal{D}}(R) \rightarrow 2^R \times Q_{\bar{\sigma}}$. Its associated game is $G_S^f = (A_S^f, W_S^f)$, where the input belongs to Adam and the output to Eve. Vertices of G_S^f are states of S , moves of Adam are $(\sigma, \phi, \text{asgn}) \in \Sigma \times \text{Tests}_{\mathcal{D}}(R) \times 2^R$, and $(\gamma, \phi, \text{asgn}) \in \Gamma \times \text{Tests}_{\mathcal{D}}(R) \times 2^R$ for Eve. Note that asgn is determined by (σ, ϕ) (respectively (γ, ϕ)), but we include it in the moves to keep the correspondence with the automaton. The winning condition is however more complex, as the semantics of register automata does not directly result from their syntax. Consider a play $\rho = p_0(\sigma_0, \phi_0, \text{asgn}_0)q_0(\gamma_0, \psi_0, \text{asgn}'_0)p_1 \dots$. It is winning if $p_0q_0p_0q_1 \dots$ is accepting *when* the action sequence $(\phi_0, \text{asgn}_0)(\psi_0, \text{asgn}'_0)(\phi_1, \text{asgn}_1)(\psi_1, \text{asgn}'_1)$ is feasible. In other words,

$$W_S^f = \left\{ \begin{array}{l} p_0(\sigma_0, \phi_0, \text{asgn}_0) \\ q_0(\gamma_0, \psi_0, \text{asgn}'_0) \in \text{Plays}(A) \\ p_1 \dots \end{array} \left| \begin{array}{l} (\phi_0, \text{asgn}_0)(\psi_0, \text{asgn}'_0) \\ (\phi_1, \text{asgn}_1) \dots \in \text{Feasible}_{\mathcal{D}}(R) \\ \Rightarrow p_0q_0p_0q_1 \dots \in \Omega \end{array} \right. \right\}$$

Contrary to the case of automatic specifications given by deterministic ω -automata, this game is not in general a sound, complete and effective abstraction of the Church synthesis problem, for multiple reasons:

- Some input transitions might be missing, which prevents Adam from inputting some data values (recall that by definition, an implementation has total domain).
- Eve can deliberately make the action sequence unfeasible. We thus have to restrict her moves.
- The game may not be decidable. This is the case for instance for the data domain $(\mathbb{N}, +1, 0)$, as one can encode runs of a Minsky machine (see L_M in Example 4.1). We show that this is also the case for $(\mathbb{N}, <, 0)$ for two-sided specifications (Theorem 7.19).
- Winning strategies might not correspond to register transducers:
 - They are not necessarily finite-memory. Indeed, the winning condition is not always ω -regular. We show that it is the case for $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$. This does not hold for $(\mathbb{N}, <, 0)$, but we show that we can still reduce to an ω -regular game for one-sided specifications.
 - The set of moves of Adam is not necessarily deterministic, as one simply asks that S is deterministic on the semantical level. This can be enforced as syntactical and semantical determinism coincide (Section 4.13). Similarly, the moves of Adam are not necessarily complete, so the corresponding transducer does not necessarily accept all inputs. This can be enforced by assuming that S is syntactically complete on input tests. Note that this latter modification might change the winner of the game. For instance, if the specification is empty, then Eve wins the game, while making it complete reveals that the specification is not realisable.
 - By definition, register transducers can only output data that have already been seen in the input. As a consequence, output transitions of the form $p \xrightarrow{\sigma, \phi, \text{asgn}} q$, where $\phi \Rightarrow \bigwedge_{r \in R} \star \neq r$ do not a priori correspond to transitions of a register transducer. We show that this property can be recovered in $(\mathbb{N}, =)$ by allowing Eve to use one additional register; this is not the case for $(\mathbb{Q}, <)$.

Recall that for $\sigma \in \{i, o\}$, we set $\bar{i} = o$ and $\bar{o} = i$.

Recall that being realisable necessitates to have total domain.

- Winning strategies of Eve only correspond to implementations with the same set of registers of the specification, which moreover use them in the same way. This is not sufficient in general.
- Finally, Eve might be tired of playing infinite-duration games with Adam.

The last limitation can be lifted since all the objects we manipulate enjoy mathematical perfection, but the others will all be a difficulty at some point.

From the Automaton Game to the Church Game: Game-Soundness

Let us first tackle the first two limitations. We group them under the notion of game-readiness. A sufficient condition for Adam to be able to input any data value is that the input transitions are complete, i.e. for all reachable configurations and all data values, there exists a transition that can be taken. Symmetrically, Eve cannot make the action sequence unfeasible if the tests on output transitions are locally concretisable, i.e. for all reachable configurations and all tests, there exists a data value which satisfies the test with regard to the valuation.

Definition 7.1 Let $S = (Q, q^i, \Sigma, R, \delta, \Omega)$ be a deterministic specification register automaton. S is *game-sound* if:

- It is complete on input transitions, i.e. for all reachable input configurations $C = (p, \nu) \in \text{Config}_i(S)$, all labelled data values $(\sigma, d) \in \Sigma \times \mathbb{D}$, there exists a transition $p \xrightarrow{\sigma, \phi, \text{asgn}} q$ that is enabled from C on reading d , i.e. $\nu, d \models \phi$.
- It is locally concretisable on output transitions, i.e. for all reachable output configurations $C' = (q, \nu') \in \text{Config}_o(S)$ and all transitions $t = q \xrightarrow{\gamma, \psi, \text{asgn}'} s$, there exists a data value $d' \in \mathbb{D}$ such that t is enabled from C' on reading d' , i.e. $\nu', d' \models \psi$.

For conciseness, we respectively denote $\text{Config}_i(S) = Q_i \times \text{Val}(R)$ and $\text{Config}_o(S) = Q_o \times \text{Val}(R)$.

As the name suggests, if a specification automaton S is game-sound, then G_S^f provides a sound abstraction for its reactive synthesis problem.

Proposition 7.2 Let S be a deterministic specification automaton that is game-sound. If Eve has a winning strategy in G_S^f , then she has a winning strategy in the Church data game G_S .

Proof idea. The main idea of the proof is that a strategy of Eve in the automaton game G_S^f consists in selecting output transitions of S as the run unrolls. It thus induces a strategy in the Church data game, as the plays that are consistent with it are runs of S . Those runs are moreover accepting if their action sequence is feasible, by definition of the acceptance condition W_S^f . Completeness on the input transitions ensures that Adam can provide any data value. Local concretisability on the output transitions guarantees that Eve can pick in the Church game a data value that corresponds to the transition.

Let us be a bit more precise. In the Church data game, Eve simulates her strategy in the finite game as follows: she maintains in memory the configuration C of S that is reached when reading the result of their interaction so far. Such a configuration exists since S is game-sound, and

is unique since S is deterministic. When she receives a labelled data value (σ, d) from Adam, she computes the input transition of S that is taken when reading (σ, d) from C (again, it exists and is unique since S is game-sound and deterministic), updates the current configuration, and feeds the transition to her strategy in G_S^f (appended to the history of the input transitions that have been played so far). The latter strategy responds with some output transition $q \xrightarrow{\gamma, \psi, \text{asgn}'} p$; correspondingly, in the Church game, Eve outputs γ , along with some data value that satisfies ψ (it exists because S is game-sound), and again updates her memory state with the resulting configuration. $\square$

Proof. We now formally establish the result. Assume that Eve has a winning strategy λ^f in G_S^f . We inductively build a strategy λ in the Church data game which maintains in memory a history h^f of G_S^f that is consistent with λ^f . Such a history consists in the states and transitions of the partial run ρ of S over the history of the play in G_S (recall that in the latter game, moves consist in data values). We also maintain the fact that if the h are increasing for the prefix order, then so are their associated h^f . Initially, λ stores the initial vertex q^f of G_S^f , which is indeed the state of the partial run on the history of the play, that is empty as no data values have been played at this point.

Now, assume that Adam and Eve are at play, and their interaction yielded an history $h^\forall \in \text{Hist}^\forall(G_S)$; it is Adam's turn to play. Write $|h^\forall| = 2i + 1$ ($i \geq 0$). By induction, we know that Eve has in memory an history $h^f = p_0 t_0 q_0 t'_0 \dots q_{i-1} t_{i-1} p_i$ (of length $2i + 1$) of G_S^f that is consistent with λ^f , and such that $p_0 q_0 \dots p_{i-1} q_{i-1} p_i = \text{states}(\rho)$ and $t_0 t'_0 \dots t_{i-1} = \text{trans}(\rho)$ (if $i = 0$, ρ simply consists in p_0 and contains no transitions), where ρ is the partial run of S on $h^\forall$. In the following, we also let $\text{valuations}(\rho) = v_0 v'_0 \dots v_i$.

Translate Adam's move in G_S^f Adam plays (σ_i, d_i) . Since S is game-sound, it is complete on its input transitions. It is also deterministic, so there exists a unique transition $t_i = p_i \xrightarrow{\sigma_i, \phi_i, \text{asgn}_i} q_i$ such that $v_i, d_i \models \phi_i$.

Then, S takes transition $(p_i, v_i) \xrightarrow[S]{t_i} (q_i, v'_i)$, where $v'_i = v_i \{\text{asgn}_i \leftarrow d_i\}$.

In G_S^f , $(\sigma_i, \phi_i, \text{asgn}_i)$ is an action of Adam, so $h^\exists = h^\forall(\sigma_i, \phi_i, \text{asgn}_i)q_i$ is an history that is consistent with λ^f .

Get Eve's answer, and translate it back to G_S Let $(\gamma_i, \psi, \text{asgn}') = \lambda^f(h^\exists)$. By definition of G_S^f , it corresponds to an output transition $t'_i = q_i \xrightarrow{\gamma_i, \psi, \text{asgn}'} p_{i+1}$. Since S is game-sound, such a transition is locally concretisable, so there exists $d'_i \in \mathbb{D}$ such that $v'_i, d'_i \models \psi$. Thus, Eve answers (γ_i, d'_i) in the Church data game. In other words, we define $\lambda(h^\forall(\sigma_i, d_i)) = (\gamma_i, d'_i)$. She also updates her memory by extending h^f to $h^f t_i q_i t'_i p_{i+1}$, which is by construction an history that is consistent with λ^f (when projecting away valuations); it moreover contains h^f as a prefix. Finally, we again have that it consists in the states and valuations of the

By Remark 3.11, one can build a strategy of Eve by induction by constructing a set of histories that contains the history v^t , and is universally closed (respectively, existentially closed) for histories of Adam (resp., Eve). We construct its memory simultaneously; if one wants to be rigorous, the reasoning consists in inductively building a set of pairs (h, ρ) , where ρ is a partial run of S over h (where the vertices of h are omitted, as they simply consist in $v_\forall$ and $v_\exists$) that is consistent with λ^f (when omitting valuations). The set of histories of λ is then defined as the projection of this set on its first component. The argument is rather phrased on a more intuitive level, so that the reader can conceive it as a play of the Church game that unfolds before their eyes.

Recall that $\text{Hist}^\forall(G_S)$ denotes the set of histories of G_S that belong to Adam.

partial run of S over $h^\forall(\sigma_i, d_i)(\gamma_i, d'_i)$ (where we omit the vertices). Thus, the inductive invariant again holds.

The constructed strategy is winning We have constructed by induction a strategy for Eve in the Church data game. We need to show that it is winning. Consider a play $\pi = v_\forall(\sigma_0, d_0)v_\exists(\gamma_0, d'_0)v_\forall(\sigma_1, d_1)v_\exists(\gamma_1, d'_1) \dots$ that is consistent with λ . The histories of λ^f that are associated with those of π are increasing for the prefix order, so we can take their limit, which is a play $p_0t_0q_0t'_0p_1t_1 \dots$. By construction, it is the sequence of states and transitions of the run ρ of S over $(\sigma_0, d_0)(\gamma_0, d'_0)(\sigma_1, d_1)(\gamma_1, d'_1) \dots$. In particular, it implies that its action sequence is feasible. By definition of W_S^f , this implies that ρ is accepting, since λ^f is winning. Thus, $((\sigma_0, d_0)(\sigma_1, d_1) \dots, (\gamma_0, d'_0)(\gamma_1, d'_1) \dots) \in \llbracket S \rrbracket$, which means that π is winning in G_S . $\square$

Remark 7.1 The form of the winning strategy that we construct is quite specific, as it consists in a strategy that picks transitions of S , with the help of a (possibly infinite) memory.

Remark 7.2 This result, as well as its converse for game-complete specifications (Proposition 7.6) could be proven by showing that G^f abstracts G in the sense of [129]. However, it would require the introduction of a considerable theoretical arsenal for a modest gain, as the proofs are already reasonably simple.

[129]: Stevens (1998), ‘Abstract interpretations of games’

Since a winning strategy in the Church data game is equivalent to an implementation (Proposition 5.2), we get:

Corollary 7.3 *Let S be a deterministic specification automaton that is game-sound. If Eve has a computable winning strategy in G_S^f , then S has an implementation.*

Remark 7.3 The game-sound condition is also sufficient for the automaton game to be a sound abstraction for *non-deterministic* specification register automata, where Eve is in charge of solving non-determinism. This calls for a study of the extension of the concept of good-for games automata [97] to register automata.

[97]: Henzinger and Piterman (2006), ‘Solving Games Without Determinization’

Good-for-Games Automata were independently discovered in [98]: Colcombet (2009), ‘The Theory of Stabilisation Monoids and Regular Cost Functions’ under the name of history-deterministic automata. See also Remark 3.14.

Targeting Register Transducer Implementations: One-sided specifications

Let us now focus on output transitions that ask for a locally fresh data value. In the case of $(\mathbb{N}, =)$, allowing these transitions does not impede decidability, but complicates the matters a bit (cf the difference between Theorems 7.12 and 7.15). In $(\mathbb{Q}, <)$, it induces a difference between the Church and the reactive synthesis problem (Theorem 7.16 and Property 7.18). We thus introduce a subclass of specification register automata, that we say are *one-sided*, that is analogous to the single-sided restriction of [33] and separately study the synthesis problems for this class.

Definition 7.2 (One-sided, transducer-like) A deterministic specification register automaton S is *one-sided* if its output transitions consist in reading the content of some register, i.e. if they all are of the form $q \xrightarrow[\gamma, \star=r, \text{asgn}]{s} s$, where $q \in Q_o, s \in Q_i, \gamma \in \Gamma, r \in R$ and $\text{asgn} \in 2^R$.

[33]: Figueira, Majumdar, and Praveen (2020), ‘Playing with Repetitions in Data Words Using Energy Games’

Observe that in this case, output transitions are necessarily locally concretisable, so S is *game-sound* if it is complete on input states, i.e. for all reachable configurations $C = (p, v) \in \text{Configs}(A)$ such that $p \in Q_i$, and for all $(\sigma, d) \in \Sigma \times \mathbb{D}$, there exists a transition t that is enabled from C on reading (σ, d) .

A one-sided automaton S can be turned into an equivalent game-sound automaton S' by adding two sink states and a linear number of transitions. More precisely, to S , add an input and an output sink state s_i and s_o , with transitions that loop between them: $s_i \xrightarrow{\sigma, \top, \emptyset} s_o$ and $s_o \xrightarrow{\gamma, \top, \emptyset} s_i$ for all $\sigma \in \Sigma, \gamma \in \Gamma$. For every input state p , add a transition $p \xrightarrow{\sigma, \phi_{p,\sigma}, \emptyset} s_o$, where $\phi_{p,\sigma}$ accepts any data value that is not accepted by other tests from p with label σ , i.e. $\phi_{p,\sigma} = \neg \bigvee \{ \phi \mid p \xrightarrow[\sigma]{\sigma, \phi, \text{asgn}} q \text{ for some } q, \text{asgn} \}$.

Finally, if it is game-sound and additionally does not conduct assignments on its output transitions, we say that it is *transducer-like*. Formally, we ask that for all $q \xrightarrow[\gamma]{\gamma, \psi, \text{asgn}} p \in \Delta_o$, we have $\text{asgn} = \emptyset$.

Assumption 7.3 Since transforming a one-sided specification into a game-sound one is linear, we assume without loss of generality that one-sided specification automata are game-sound.

Over (game-sound) one-sided automata, finite-memory strategies in the automaton game can be turned into register transducer strategies in the Church data game, as they correspond to strategies that pick transitions in the register automaton specification (see Remark 7.1), and the output transitions of a one-sided automaton closely resemble the output of a transducer transition. Thus, the automaton game also yields a sound abstraction for the Church synthesis problem. If the specification automaton is transducer-like, such a conversion is moreover direct. Formally, Proposition 7.2 can be refined into:

Proposition 7.4 *Let S be a deterministic one-sided specification automaton with states Q and registers R that is game-sound. If Eve has a finite-memory winning strategy in G_S^f with memory M , then she has a register-transducer winning strategy in the Church data game G_S with memory $Q \times M \times R^R$ and registers R .*

If S is transducer-like, its memory can be reduced to $Q \times M$.

Proof idea. If a register automaton specification is one-sided, then its output transitions are of the form $q \xrightarrow[\gamma]{\gamma, \star=r, \text{asgn}'} p$. This essentially consist in outputting the content of some register, possibly reassigning it to some other registers. Thus, the strategy of Eve that is constructed in the proof of Proposition 7.2 corresponds to a register transducer that simulates S using registers R and picks transition with its memory $Q \times M$, possibly conducting reassignments. The only operations are to conduct tests over the input (those tests moreover induce a deterministic and complete transition function), output the content of some register and update some valuation. Removing assignments on the output can always be done, since the output is by construction equal to the content of some register, as the specification is one-sided. This is done by storing additional information in the states (see Proposition 4.6), hence the factor R^R , which disappears

Since S is one-sided, we have that $\psi := \star = r$ for some $r \in R$.

We still explicitly ask for game-soundness in statements, to allow for a modular or diagonal reading of the document.

Game-soundness can be ensured without loss of generality through a linear transformation (Assumption 7.3).

The factor R^R is due to the fact that transducers do not conduct assignments over the output, so they have to be removed by adding information to the states, as in Proposition 4.6.

if we assume that S is transducer-like, i.e. already does not conduct such assignments. $\square$

Proof. Let us be more precise. Let S be a one-sided specification automaton with states Q and registers R . Come back to the construction of λ from λ^f in Proposition 7.2. Assume that λ^f is finite-memory, with memory space M . Let us show that λ can be computed by a register transducer with states $Q \times M$ and registers R , that is allowed to conduct assignments on its output. First, note that to simulate λ^f , Eve does not actually need to keep in memory the full history h^f , but only the corresponding memory state $m \in M$ of λ^f . Then, each pair of successive Adam and Eve moves $(\sigma_i, \phi_i, \text{asgn}_i)$ and $\gamma_i, \star = r_i, \emptyset$, that correspond

to transitions of S $p_i \xrightarrow{\sigma_i, \phi_i, \text{asgn}_i} q_i \xrightarrow{\gamma_i, \star = r_i, \text{asgn}'_i} p_{i+1}$, are computed by

a register transducer transition $(p_i, m) \xrightarrow{\sigma_i, \phi_i | \text{asgn}_i, \gamma_i, r_i, \text{asgn}'_i := r_i} (p_{i+1}, m')$,

where m and m' denote the memory states of λ^f before and after the successive moves. The notation $\text{asgn}' := r_i$ means that registers of asgn' are assigned the content of r_i . It is routine (although symbol-heavy) to show that such a machine indeed computes λ . By Proposition 4.6, one can then remove $\text{asgn}' := r_i$ the assignments on the output, up to blowing up the state space by a factor R^R . If S is transducer-like, no such assignments are conducted, so this last transformation is not necessary and the R^R factor is absent. $\square$

Since a register transducer winning strategy in the Church data game is equivalent to a register transducer implementation (Proposition 5.2), we get:

Corollary 7.5 *Let S be a deterministic one-sided specification automaton that is game-sound. If Eve has a finite-memory strategy in G_S^f with memory M , then S has a register transducer implementation with states $Q \times M \times R^R$ and registers R , where Q is the set of states of S , and R its set of registers. Its state space is moreover reduced to $Q \times M$ if S is transducer-like.*

From the Church Game to the Automaton Game: Game-Completeness

Over some data domains, one can translate a strategy of Eve in the Church data game back to the automaton game, in which case such a game provides a complete abstraction for the reactive synthesis problem. A sufficient condition is that of game-completeness, which asks that the input tests are locally concretisable. In this case, Eve can concretise the actions $(\sigma, \phi, \text{asgn})$ provided by Adam by a data value in a history-deterministic way. In other words, her choices should not compromise the feasibility of the whole sequence.

Example 7.1 Let us give an example to highlight what breaks when the input transitions are not locally concretisable. Consider the register automaton over $(\mathbb{D}, =)$ of Figure 7.1 (the label alphabet is unary, so labels are omitted). The environment initially provides a data value, and the subsequent input transitions depend on whether it is equal to $\#$. If Eve does not have this information, her choice for the first $\top$ test compromises either the $r = \#$ test, or the $r \neq \#$ one (independently of her strategy in the

This example might look pathological to the reader, and it indeed is. However, one can construct more meaningful ones based on the phenomenon that is illustrated here. Observe that without labels, the automaton game for one-sided specifications with a single register is never very interesting, since Eve only has one strategy in it. Here, we precisely use this fact to simplify our reasonings, as the focus is set elsewhere, on the information that Adam provides or not.

Church data game). For instance, assume she concretises $\top, \downarrow r$ by picking $\#$. Then, Adam picks the transition $q \xrightarrow{r \neq \#} s$. Eve is unable to concretise it with any data value, since $r \neq \#$ is not satisfiable (as $r = \#$). Yet, the sequence of actions $(\top, \downarrow r) \cdot (\star = r) \cdot (r \neq \#)$ can be extended in a feasible one: it suffices to take some data value distinct from $\#$ when concretising the first action. The other case is similar: if Eve initially picks a data value distinct from $\#$, Adam simply needs to pick the other transition to make her lose. Overall, Eve fails to convert her strategy in the Church data game to a winning strategy in the automaton game.

Yet, the sole strategy that Eve has is winning, since any play ends in the accepting loop between t and u . Such a strategy, when interpreted as a register automaton whose output transitions are determined by its input ones, corresponds to a register transducer that implements the function $f_{\text{fst}} : d \cdot \mathbb{D}^\omega \mapsto d^\omega$, which realises the specification, as they are equal. One thus has to impose Adam to provide full information on the data values he provides, by asking that the input tests consist in maximally consistent conjunctions of atomic formulas, a form that we called explicit in Section 4.4.4.

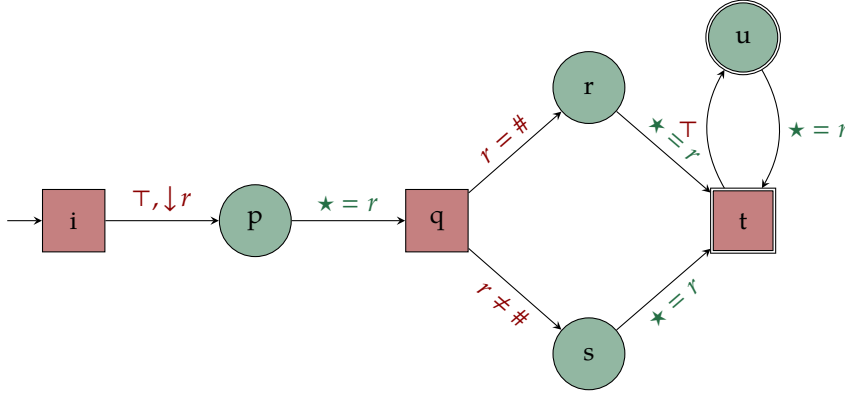


Figure 7.1: A specification automaton that recognises the function $f_{\text{fst}} : d \mathbb{D}^\omega \mapsto d^\omega$, which is easily realisable by a register transducer. Yet, Eve fails in translating her strategy in the Church data game to one in the automaton game.

Definition 7.4 Let S be a specification automaton. We say that S is *game-complete* if its input transitions are locally concretisable, i.e. for all reachable configurations $t = (p, v) \in \text{Config}_i(S)$ and all transitions $p \xrightarrow{\sigma, \phi, \text{asgn}} q$, there exists $d \in \mathbb{D}$ such that t is enabled from C on reading d , i.e. $v, d \models \phi$.

Let us show that if a specification automaton is game-complete, then from a winning strategy of Eve in the Church data game, one can build a winning strategy for Eve in the automaton game.

Proposition 7.6 Let S be a game-complete specification automaton. If Eve has a winning strategy in the Church data game G_S , she has a winning strategy in the automaton game G_S^f .

Proof idea. In Proposition 7.2, Eve feeds the data values she receives from Adam to her strategy in the automaton game and mimicks its response. Here, we need to do the converse: Eve receives an input action from Adam, and needs to concretise it with some data value to feed it to her strategy in the Church data game.

Thus, in G_S^f , Eve constructs a play which is the run of S over some play of G_S that is consistent with her strategy in the Church data game. To that

Note that here, we do not ask that S is game-sound, as we are concerned with the other direction of the transformation. We later on combine the two notions into that of game-readiness to get a sound and complete abstraction.

end, she maintains in memory the current configuration of S . When she receives a move $(\sigma, \phi, \text{asgn})$ from Adam, she arbitrarily picks a data value that satisfies the test with regard to the current valuation. She can do so since input transitions are locally concretisable. She then feeds this data value to her strategy in the data game, which reacts with some output labelled data value. Necessarily, this corresponds to a transition in S , otherwise the play cannot be winning in G_S , and we know that Eve wins such a game. Correspondingly, she picks this transition in the automaton game. Overall, the play in the automaton game must be winning, since it corresponds to the run of S over some winning play in G_S . Thus, the resulting strategy is winning. $\square$

Proof. Let us now develop the argument. Let S be a game-complete specification automaton, and assume that Eve has a winning strategy λ in the Church data game G_S .

Constructing the strategy λ^f We construct a strategy λ^f for Eve by induction, along with, for each consistent history h^f , an history h of G_S such that h^f is the sequence of states and transitions of the partial run of S over h . We also maintain the fact that if the h^f are increasing for the prefix order, then so are their associated h . When no actions have been played, we let $h = v_\forall$, as it is the initial vertex of G_S .

Recall that strategies can be build by induction on their consistent plays, see Remark 3.11.

Now, assume that the current history is $h^f = p_0(\sigma_0, \phi_0, \text{asgn}_0)q_0(\gamma_0, \star = r_0, \text{asgn}'_0)p_1 \dots p_i$, and it is Adam's turn to play. By induction, there exists a history $h = v_\forall d_0 v_\exists d'_0 \dots v_\exists d'_i v_\forall$ and sequences of valuations $(v_j)_{j \leq i}$ and $(v'_j)_{j \leq i}$ such that $(p_0, v_0) \xrightarrow{\sigma_0, \phi_0, \text{asgn}_0, d_0} (q_0, v'_0) \xrightarrow{\gamma_0, \star = r_0, \text{asgn}'_0, d'_0} (p_1, v_1) \dots (p_i, v_i)$ is the partial run of S over h (where vertices are omitted). Now Adam plays $(\sigma_i, \phi_i, \text{asgn}_i) \in \Sigma \times \text{Tests}_{\mathcal{D}}(R) \times 2^R$. The corresponding transition is $t_i^\natural = q_i^\natural \xrightarrow{\sigma_i, \phi_i, \text{asgn}_i} q_i^\circ$. Since S is game-complete, we know that there exists $d_i \in \mathbb{D}$ such that $v_i, d_i \models \phi_i$. Then, Eve feeds (σ_i, d_i) to her strategy in G_S . Formally, she sets $h^\exists = h \cdot (\sigma_i, d_i)$, and lets $(\gamma_i, d'_i) = \lambda(h^\exists)$, then updates again $h^\exists$ to get $h^\forall = h^\exists \cdot (\gamma_i, d'_i)$. The configuration of S is updated accordingly, by letting $v'_i = v_i \{\text{asgn} \leftarrow d_i\}$.

Necessarily, there exists an output transition $t'_i = q_i \xrightarrow{\gamma_i, \psi_i, \text{asgn}'_i} p_{i+1}$

such that $(q_i, v'_i) \xrightarrow[S]{\gamma_i, d'_i} (p_{i+1}, v_{i+1})$. Otherwise, it means that there

does not exist any run of S over $h^\forall$. In that case, for any infinite suffix $s = (\sigma_{i+1}, d_{i+1})(\gamma_{i+1}, d'_{i+1}) \dots$, we have that $h^\forall \cdot s \notin \llbracket S \rrbracket$, so this is the case in particular of any play consistent with λ that starts with $h^\forall$, which contradicts the fact that λ is a winning strategy in G_S (such a play necessarily exists since Eve must be able to react to any sequence of data values in G_S).

Then, Eve plays $(\gamma_i, \psi_i, \text{asgn}'_i)$ and updates h to $h^\forall$. The inductive invariant is maintained, as $h^\forall$ contains h as a prefix, and $h^f \cdot (\sigma_i, \phi_i, \text{asgn}_i) \cdot (\gamma_i, \star = r_i, \text{asgn}'_i)$ indeed corresponds to a partial run of S over $h^\forall$.

λ^f is winning Let us now show that λ^f is winning. Consider a play π^f that is consistent with λ^f . Take the limit of the histories h of G_S

associated with histories h^f of π^f : we get a play π of G_S that is consistent with λ , and such that π^f consists in the sequence of states and transitions of the run of S over π (omitting the trivial vertices of G_S). We know that π is winning, since it is consistent with λ , which means that $\pi \in \llbracket S \rrbracket$ (again, omitting vertices). Thus, the corresponding run of S is accepting, which implies that π^f is winning. $\square$

Wrapping Up: Game-Readiness

As a consequence, G_S^f is a sound and complete abstraction for the reactive synthesis problem for specifications S that are both game-sound and game-complete; we group the two notions under that of game-readiness. Then, finding an implementation for S reduces to finding a winning strategy for Eve in G_S^f . Note that G_S^f is not necessarily decidable, so this does not necessarily yield a decision procedure.

Definition 7.5 Let S be a specification register automaton. It is *game-ready* whenever it is both game-sound and game-complete, i.e.:

- It is complete and locally concretisable on its input transitions
- It is locally concretisable on its output transitions.

Observation 7.7 In a locally concretisable automaton, all paths are feasible (Observation 4.22). As a consequence, if S is game-ready, then W_S^f is ω -regular, since in that case W_S^f essentially consists in the acceptance condition Ω of S :

$$W_S^f = \left\{ \begin{array}{l} p_0(\sigma_0, \phi_0, \text{asgn}_0) \\ q_0(\gamma_0, \psi_0, \text{asgn}'_0) \in \text{Plays}(A) \\ p_1 \dots \end{array} \middle| p_0 q_0 p_0 q_1 \dots \in \Omega \right\}$$

As a consequence, solving the reactive synthesis problem can be done by playing directly on the specification automaton, as if it were an ω -automaton:

Proposition 7.8 Let S be a game-ready deterministic specification automaton. The following are equivalent:

- (i) S is realisable by some synchronous program
- (ii) Eve has a computable winning strategy in the Church data game
- (iii) Eve has a computable winning strategy in the automaton game
- (iv) Eve has a finite-memory winning strategy in the automaton game

Proof. This proposition consists in wrapping everything together. (i) $\Leftrightarrow$ (ii) is exactly Proposition 5.2. (iii) $\Rightarrow$ (ii) is Proposition 7.2. (ii) $\Rightarrow$ (iii) is Proposition 7.6. Finally, (iv) $\Leftrightarrow$ (iii) results from the finite-memory determinacy of ω -regular games (Theorem 3.8 and Corollary 3.9). $\square$

Since ω -regular games are decidable, this yields decidability of the reactive synthesis problem for game-ready specification automata:

Proposition 7.9 The reactive synthesis problem for specifications given by game-ready deterministic automata is decidable.

More precisely, if S is given as a game-ready deterministic parity register automaton, one can decide in $\mathcal{O}(n^d)$ whether it is realisable by a synchronous program, where n is the size of S and d is its parity index.

Remark 7.4 The reader might be surprised by the relatively modest (if not polynomial) complexity of solving the problem, in contrast with, e.g., solving the emptiness problem for register automata, which is PSPACE-complete, already for deterministic register automata over $(\mathbb{D}, =)$ (Theorem 4.20). The caveat lies in the transformation into a game-ready automaton, which is not always doable (e.g. for $(\mathbb{N}, <, 0)$, cf Property 4.39), and exponential for $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$ (Sections 7.3.2 and 7.3.3). In particular, if a register automaton is locally concretisable, then it is empty if and only if it is empty when interpreted syntactically as an ω -automaton (Observation 4.22).

Remark 7.5 Note that the resulting implementation is not necessarily computable by a register transducer, since the specification might ask to produce data values that are absent from the input. In the case of $(\mathbb{D}, =)$, we are able to show that register transducer actually suffice (Theorem 7.15); this is not the case anymore for $(\mathbb{Q}, <)$ (see Property 7.18). However, it has a very specific form, as it consists in picking transitions in the specification automaton with the help of a finite memory, which corresponds to very simple programs, provided the concretisation of a transition is simple. This is the case for $(\mathbb{Q}, <)$, as concretising $r_1 < \star < r_2$ can be done by picking $\frac{r_1+r_2}{2}$ (see Remark 7.9).

If the specification S is game-ready and one-sided, we get that the reactive synthesis problem is decidable and equivalent to the Church synthesis problem.

Proposition 7.10 *Let S be a game-ready one-sided specification with registers R . The following are equivalent:*

- (i) S has an implementation
- (ii) S has a register transducer implementation with registers R
- (iii) Eve has a finite-memory winning strategy in G_S^f

Proof. Again, this is a wrapper proposition. Let S be a game-ready one-sided specification. (i) $\Leftrightarrow$ (iii) is a consequence of Proposition 7.8 (items (i) and (iv)). Then, in the case of one-sided specifications that are game-sound (which is the case of game-ready ones), we know that finite-memory strategies in the automaton game correspond to register transducers (Corollary 7.5). $\square$

7.3.2. The Case of Equality

As expected, we start with the simplest data domain, namely $(\mathbb{D}, =, \mathbb{C})$. Since one can always turn a register automaton over this domain into a locally concretisable one (Proposition 4.25), we get that any specification automaton can be converted into one that is game-ready. Thus, solving the reactive synthesis problem amounts to finding a winning strategy of Eve in the automaton game if it exists, by Proposition 7.9. Moreover, we are able to provide an EXPTIME lower bound, meaning that the problem is complete for this complexity class.

We take into account the number of transitions, since there can be exponentially many tests.

Theorem 7.11 *The reactive synthesis problem for specifications given as deterministic register automata over $(\mathbb{D}, =, \mathbb{C})$ is ExpTIME -complete.*

Proof. Membership in ExpTIME essentially follows from the fact that any register automaton over $(\mathbb{D}, =, \mathbb{C})$ can be turned into a locally concretisable one, with an exponential blowup (Proposition 4.25). Hardness is obtained by lifting the proof of [28, Theorem 5.1] to alternating Turing machine using the interaction between the system and its environment: the system resolves non-determinism, while the environment resolves universality.

Membership in ExpTIME Let S be a deterministic specification automaton with state space Q of size n and registers R of size k , and acceptance condition Ω . We treat the case where Ω is a parity condition with parity index d .

By Proposition 4.25, we know that there exists a specification automaton that is equivalent to S and which is locally concretisable, so it is game-ready. Such an automaton is of size nB_k , where B_k is the k -th Bell number, that counts the number of partitions of R . Then, by Proposition 7.9, finding an implementation for S is equivalent with determining whether Eve wins the automaton game. This is a parity game, that can be solved in time $\mathcal{O}((nB_k)^d)$, which is exponential in the size of the input.

ExpTIME-hardness The following proof is an adaptation of the one establishing PSPACE-hardness of the nonemptiness problem for deterministic register automata presented in [28, Theorem 5.1]; we spell it out for completeness. It consists in a reduction from the halting problem for alternating Turing machines over a binary alphabet with linearly bounded tapes. The main idea is to use the input part to simulate universal transitions, and the output part to simulate non-deterministic ones, hence simulating alternation, which yields an ExpTIME lower bound.

An *alternating Turing machine* [120, Section 10.3] is a tuple $M = (Q, q_i, \delta)$, where:

- Q is a finite set of *states*, partitioned into *existential* ($Q_\exists$) and *universal* ($Q_\forall$) states: $Q = Q_\exists \sqcup Q_\forall$, where $q_i \in Q_\exists$ is the *initial state*
- $\delta : Q_\sigma \times \{0, 1\} \rightarrow 2^{Q_{\bar{\sigma}} \times \{0, 1\} \times \{-1, 1\}}$ is the *transition function*, where $\sigma \in \{\forall, \exists\}$ and $\bar{\forall} = \exists, \bar{\exists} = \forall$.

Then, a *configuration* of M is a triple (q, i, w) , where $q \in Q$ is the machine state, $i \in \{0, \dots, |M| - 1\}$ is the head position, and $w \in \{0, 1\}^{|M|}$ is the tape content. It is *existential* if $q \in Q_\exists$, *universal* if $q \in Q_\forall$. A configuration (q', i', w') is a *successor* of (q, i, w) if there exists $(p, a, m) \in \delta(q, w[i])$, $p = q'$, $i' = i + m \in \{0, \dots, |M| - 1\}$ and w' is such that $\forall j \neq i$, $w'[j] = w[j]$ and $w[i] = a$. We then say that $t = q \xrightarrow{w[i], a, m} p$ is the *associated transition*. Then, a configuration (q, i, w) is *accepting* if, either:

1. $q \in Q_\exists$ is an existential state and at least one of the successor configurations is accepting
2. $q \in Q_\forall$ is a universal state and all the successor configurations are accepting

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

There is no known simple closed form for B_k . For the present complexity analysis, we only need that it is bounded by an exponential function. Observe that it is trivially bounded by 2^{k^2} , which counts the number of binary relations over a set of size k , since a partition is in particular a binary relation. Finer bounds exist, e.g. we have, for all $k \geq 0$, $B_k < \left(\frac{0.792n}{\ln(n+1)}\right)^n$ from [130]: Berend and Tassa (2010), ‘Improved bounds on bell numbers and on moments of sums of random variables’ (Theorem 2.1).

[120]: Sipser (2013), *Introduction to the Theory of Computation*

Note that we ask that non-deterministic and universal states alternate. This is done to simplify the proof, and is without loss of generality.

We call this problem ‘halting problem’ and not membership problem since we fix the input, as we only consider the initial configuration where the tape only contains 0.

Note that the base case of this inductive definition consists in universal configurations with no successor. The following problem is ExpTime-hard [27, Theorem 3.3]: given an alternating Turing machine M , decide whether its initial configuration $(q_i, 0, 0^{|M|})$ is accepting.

[27]: Chandra, Kozen, and Stockmeyer (1981), ‘Alternation’

Finally, a *computation* is a finite sequence of successive configurations. Let $(q_0, i_0, w_0) \dots (q_n, i_n, w_n)$ be a computation of M , and $t_0 \dots t_{n-1}$ the sequence of associated transitions. We encode this computation by the following finite data word over $(\mathbb{D}, =, \mathbb{C})$ with label alphabet $Q \sqcup \delta \sqcup \{-\}$:

We use the notations of [28, Theorem 5.1].

$$(-, d_0)(-, d_1)a_0^0 a_1^0 \dots a_{|M|-1}^0 t_0 a_0^1 a_1^1 \dots a_{|M|-1}^1 t_1 \dots t_{n-1} a_0^n a_1^n \dots a_{|M|-1}^n$$

where $d_0 \neq d_1 \in \mathbb{D}$ are two distinct data values respectively encoding letters 0 and 1, and we have $\text{lab}(a_l^k) = q_k$ if $l = i_k$ and $\text{lab}(a_l^k) = -$ otherwise. Then, $\text{dt}(a_l^k) = d_0$ if $w_k[l] = 0$ and $\text{dt}(a_l^k) = d_1$ if $w_k[l] = 1$. $\text{dt}(t_k)$ does not matter.

As in [28, Theorem 5.1], we can construct a *deterministic* register automaton A_M which accepts a data word w if and only if w has a prefix that encodes a computation of M from the initial configuration to a configuration with no successor. Since transitions are part of the input, they do not have to be guessed: neither non-deterministic nor universal branching is needed here (they are respectively simulated by the system and the environment). To present a self-contained proof, we recall the construction: A_M has states Q , along with an $|M|$ -bounded counter l to keep track of the position of the reading head in w_k , a variable i taking its values in $\{0, \dots, |M| - 1\}$ used to store the value of i_k and a variable t taking its values in δ to memorise t_k , which overall yields a $\mathcal{O}(|M|^4)$ state space. Its label alphabet is $\Sigma = Q \sqcup \delta \sqcup \{-\}$, and it has $|M| + 2$ registers: r_0 and r_1 respectively store d_0 and d_1 , and, for all $0 \leq l < |M|$, r'_l successively stores the different values of $w_k[l]$ for $0 \leq k \leq n$. A run of A_M unfolds as follows: initially, A_M stores d_0 and d_1 and verifies that they are distinct, then checks that $w_0 = 0^{|M|}$. Now, it has to check successorship of configurations, while maintaining the invariant that at any step k , r'_l contains $w_k[l]$. It proceeds as follows: when reading $t_k = q \xrightarrow{c, a, m} p$, it checks that $q = q_k$ (q_k was stored as the target of t_{k-1}), $c = w_k[i_k]$ (i.e. that r'_{i_k} contains d_c), and updates the value of i_k to $i_{k+1} = i_k + m_k$, while verifying that $i_k \in \{0, \dots, |M| - 1\}$. Then, with the help of its registers and its counter l , it checks that $w_{k+1}[l] = w_k[l]$ for all $l \neq i_{k+1}$, and that $w_{k+1}[i_{k+1}] = d_a$.

From such an automaton, by adding $\$$ s to enforce alternation between input and output, we can build a deterministic specification automaton such that the environment provides the encoding of the successive configurations and resolves universal branching, while the system has to resolve non-determinism (i.e. it chooses which non-deterministic transition to take). Then, if the environment can force the computation to go on ad infinitum, he wins, otherwise (if either the provided encoding is not correct, or if the computation is finite), the system wins, i.e. the specification is satisfied. Formally:

The data values associated with $\$$ and the t_i are irrelevant and not depicted. We broke our code of conduct of not colouring text to highlight the fact that transitions are alternately picked by the environment and the system (t_0 and t_2 are in **red**, while t_1 is in **green**). It also clarifies who carries the burden, e.g. that it is the environment who provides encodings of configurations. Recall that $\Sigma = Q \sqcup \delta \sqcup \{-\}$.

$$\begin{aligned}
S = & \left\{ \left(\begin{array}{l} (-, d_0) \$ (-, d_1) \$ \\ \langle c_0, \$^{|M|} \rangle t_0 \$ \\ \langle c_1, \$^{|M|} \rangle \$ t_1 \\ \langle c_2, \$^{|M|} \rangle t_2 \$ \dots \\ \langle c_n, \$^{|M|} \rangle (\$ \$)^\omega \end{array} \right) \mid \begin{array}{l} d_0 \neq d_1 \text{ and } c_0 t_0 c_1 t_1 c_2 t_2 \dots t_{n-1} c_n \text{ is} \\ \text{the encoding of a computation of } M \end{array} \right\} \\
& \cup \left\{ \langle u, v \rangle \mid \begin{array}{l} u, v \in (\Sigma \times \mathbb{D})^\omega \\ \text{there exists a prefix of } w \text{ which is not the encoding} \\ \text{of a computation of } M \end{array} \right\} \\
& \cup \left\{ (-, d_0) \$ (-, d_1) \$ \langle u, v \rangle \mid \begin{array}{l} d_0 = d_1 \\ u, v \in (\Sigma \times \mathbb{D})^\omega \end{array} \right\}
\end{aligned}$$

Let us insist on the fact that the even (i.e. universal) transitions are picked by the environment, while the odd (i.e. non-deterministic) transitions are picked by the system.

Remark 7.6 Note that S is recognised by a *one-sided* specification automaton.

Now, if M halts, A admits an implementation, which behaves as follows: it first checks that the d_0 and d_1 given as input are indeed distinct. Then, when reading the $sizeM$ next labelled data values, it verifies that the given input is indeed an encoding of the initial configuration, while outputting $\$$ s. It then checks that c_1 is indeed a successor of c_0 following t_0 , again outputting $\$$ s. Then, if it receives as input a $\$$, it picks a transition t_1 which is a witness that c_0 is indeed accepting, and so on. If, at some point, the given input is not a valid encoding, then it behaves arbitrarily (e.g. by outputting only $\$$ s).

Remark 7.7 Note that such an implementation can be computed by a register transducer.

Conversely, if M does not halt, then, by choosing an input whose universal transitions witness that c_0 is not accepting, then either the implementation provides some non-admissible output at some point, or the computation goes on ad infinitum, which in both cases breaks the specification S .

Thus, S is realisable if and only if M halts and its size is polynomial in $|M|$, which yields the sought ExpTime lower bound. $\square$

Now, since the automaton game is ω -regular for game-ready specification automata, we know that if such a strategy exists, then there exists one that has finite memory, which implies that for one-sided specifications, the problem coincides with the Church synthesis problem: if there exists an implementation, there exists one that can be implemented with a register transducer (Proposition 7.10).

Theorem 7.12 ([41, Theorem 3.7]) *The reactive synthesis problem and the Church synthesis problem for specifications given as deterministic one-sided register automata over $(\mathbb{D}, =, \mathbb{C})$ coincide and are both ExpTime -complete.*

Moreover, it suffices to target implementations that use the same registers as the implementation, and in the same way.

Proof. Membership in ExpTime follows from the above Theorem 7.11 and from Proposition 7.10. Hardness is obtained through the same reduction as above, thanks to Remarks 7.6 and 7.7. $\square$

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

An important corollary is that there is a threshold for register-bounded synthesis over one-sided specification automata.

Corollary 7.13 *Let $l \geq 1$. For all $k \geq l$, the k -register-bounded synthesis problem and the Church synthesis problem are equivalent for specifications given as one-sided automata with l registers over data domain $(\mathbb{D}, =, \mathfrak{C})$.*

Note that this is also the case for the reactive synthesis problem, as it is equivalent to the Church synthesis problem.

The above results again hold for $(\mathbb{Q}, <, 0)$. In the case of $(\mathbb{D}, \mathfrak{C})$, one can moreover lift the one-sided restriction for the Church synthesis: even for two-sided ones, if an implementation exists, then a register transducer one exists, with at most one additional register (provided registers can be initialised with values distinct from #):

Proposition 7.14 *Let S be a deterministic specification automaton over data domain $(\mathbb{D}, =, \mathfrak{C})$ with registers R . If S admits an implementation, it admits a register transducer implementation with registers $R \sqcup \{r_0\}$, where r_0 is some additional register.*

Proof idea. The construction is quite technical; we only give the main idea of the proof. Consider a winning strategy in the automaton game. Since it is an ω -regular game, we can assume that such a strategy is finite-memory. The only difficulty consists in output transitions that conduct a test asking for a locally fresh data value. However, with one additional register, we can maintain the invariant that the registers contain a locally fresh data value, by maintaining the fact that they contain pairwise distinct data values. This is done as follows: whenever the specification asks to store an incoming data value that would break the invariant, the implementation instead stores this information in memory, as was done when removing reassignments (Proposition 4.6). If a register transducer is allowed to take arbitrary initial values, it suffices to initialise it with pairwise distinct data values. $\square$

As a consequence, we get:

Theorem 7.15 *The reactive synthesis problem and Church synthesis problem for specifications given as deterministic automata over data domain $(\mathbb{D}, =, \mathfrak{C})$ are equivalent and decidable. They are both ExpTime -complete.*

Moreover, given a specification with k registers, it suffices to target implementations with $k + 1$ registers.

7.3.3. The Case of $(\mathbb{Q}, <, 0)$

The case of $(\mathbb{Q}, <, 0)$ is similar for the treatment of reactive synthesis and of the Church problem over one-sided specifications, since the proof relies on turning an automaton into a locally concretisable one. Thus, we have:

Theorem 7.16 *The reactive synthesis problem for specifications given as deterministic register automata over $(\mathbb{Q}, <, 0)$ is ExpTime -complete.*

Proof. The proof is the same of for Theorem 7.11, we only redo the complexity analysis: by Proposition 4.37, we know that there exists a specification automaton that is equivalent to S and which is locally concretisable, hence game-ready. The size of such an automaton can be bounded by $n2^{k^2}$ (cf Definition 4.6). Then, by Proposition 7.9, finding an implementation for S is equivalent with determining whether Eve wins the automaton game. This is a parity game, that can be solved in time $\mathcal{O}((n2^{k^2})^d)$, which is exponential in the size of the input.

The ExpTime lower bound is inherited from the case of $(\mathbb{D}, =, \mathbb{C})$. $\square$

Remark 7.8 Decidability of the above problem can be derived from [131, Section 7], which establishes that parity games where the vertices of the game can store rational numbers are decidable. However, the complexity is not clear, and the argument is more involved.

[131]: Klin and Lelyk (2019), ‘Scalar and Vectorial mu-calculus with Atoms’

For one-sided specifications, we get, by Proposition 7.10 (the ExpTime lower bound is again inherited from $(\mathbb{D}, =, \mathbb{C})$):

Theorem 7.17 *The reactive synthesis problem and the Church synthesis problem for specifications given as deterministic one-sided register automata over $(\mathbb{D}, =, \mathbb{C})$ coincide and are both ExpTime -complete.*

However, this is not anymore the case for two-sided specification.

Property 7.18 *There exists a deterministic specification automaton over $(\mathbb{Q}, <, 0)$ which can be implemented by a synchronous program but cannot be implemented by any register transducer.*

Proof. Consider the specification automaton of Figure 7.2. It can be realised by an implementation that outputs $0 \cdot 1 \cdot 2 \dots$, ignoring its input, but not by any register transducer: a register transducer can only output data values that appear in the input, so if the environment provide a constant input, e.g. 0^ω , it cannot come up with infinitely many data values (let alone in increasing order). $\square$

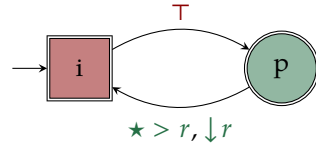


Figure 7.2: A specification automaton over $(\mathbb{Q}, <, 0)$ that asks the system to output infinitely many data values in increasing order, ignoring its input.

Remark 7.9 Note however that if a specification given by a deterministic register automaton over $(\mathbb{Q}, <, 0)$ is realisable, it can be implemented by a relatively simple program: it suffices that it is able to provide data values that satisfy tests of the form $r_1 < \star < r_2$ (this can be done by outputting $\frac{r_1+r_2}{2}$), $\star > r$ (take $r + 1$) and $\star < r$ (take $r - 1$).

7.3.4. The Case of $(\mathbb{N}, <, 0)$

The case of $(\mathbb{N}, <, 0)$ is significantly more involved. Indeed, it is not the case anymore that feasible action sequences form an ω -regular language (Property 4.38), which impedes the abstraction of the Church game in a finite way. And, as a matter of fact, the unbounded synthesis problem from specifications expressed as deterministic register automata over $(\mathbb{N}, <, 0)$ is undecidable, even if we restrict to implementations that

are computable by register transducers (Theorem 7.19; provided their registers can be initialised to distinct values). We start the section by establishing this result, and then show that if the specification is one-sided, the problem becomes decidable (Theorem 7.28). But first, let us provide an example that highlights the difficulties that arise in this setting. Intuitively, this is because $(\mathbb{N}, <, 0)$ allows some form of counting.

Example 7.2 In the game associated with the automaton of Figure 7.3, Eve only has one strategy. Assume that Adam plays $\star > 0, \downarrow r$. A bit heedlessly, Eve chooses to concretise it with the value 1 in the Church data game. Then, Adam plays $0 < \star < r$, and she is unable to provide any data value that satisfies the test, since the current valuation is $v(r) = 1$. Thus, she fails to convert her strategy in the Church data game, whereas there exists a winning strategy for her in the automaton game (the only one, actually). More generally, if Adam's strategy is not restricted, Eve does not have a

When we restrict to register transducers with initial valuation v_R^l , the problem is open (Open Problem 7.2).

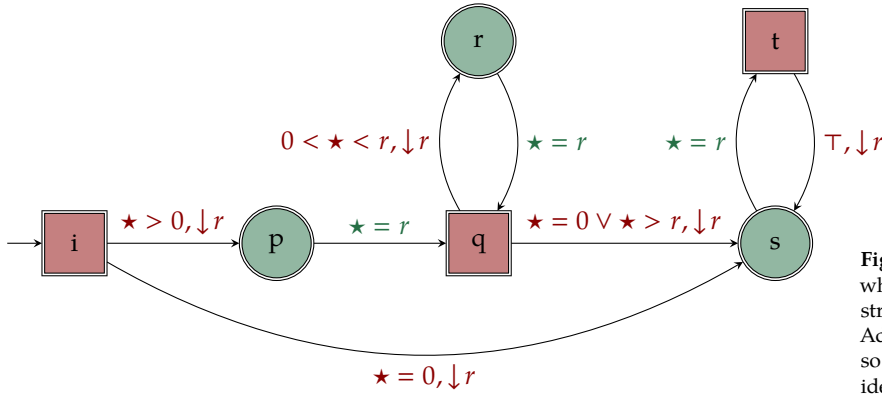


Figure 7.3.: A specification automaton in which Eve does not have a concretisation strategy that matches all behaviours of Adam. Note that all states are accepting, so the specification is realisable by the identity.

procedure to concretise data values along the run. Indeed, the strategy of Adam could consist in playing arbitrarily long sequences of tests $0 < \star < r$ transitioning from q to r . This can be concretised by picking a sufficiently large value on the first test $\star > 0, \downarrow r$, which means that Adam's sequence is feasible, but any concretisation of Eve can be beaten by a strategy of Adam that plays a longer sequence. As we demonstrate, in one-sided specifications like this one, we can recover decidability by showing that it suffices to consider finite-memory strategies for Adam.

Two-Sided Specifications

However, in the case of two-sided specifications, antagonism between the two players can be used to simulate counting, hence yielding undecidability. Indeed, if the two players are able to pick data values, one can simulate a counter as follows: Eve provides the values of the counter, and Adam checks that he does not cheat on the increments, using the fact that $c' = c + 1$ whenever $c' > c$ and there does not exist $d \in \mathbb{N}$ such that $c < d < c'$; similarly for decrements.

Theorem 7.19 ([42, Theorem 10]) *The reactive synthesis problem for specifications expressed by deterministic register automata over $(\mathbb{N}, <, 0)$ is undecidable.*

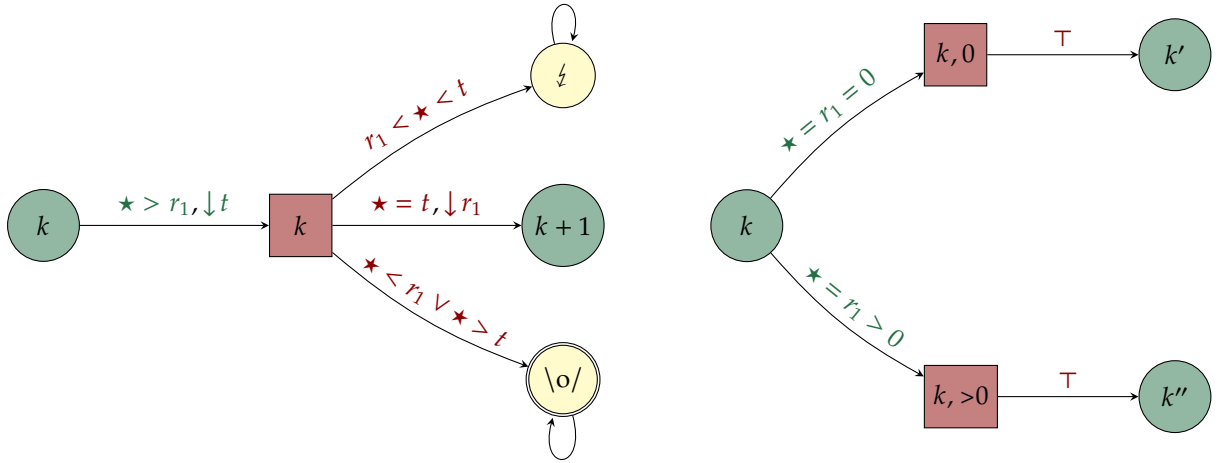
This remains the case if we restrict to implementations consisting in register transducers, provided they can take an arbitrary initial valuation.

[42]: Exibard, Filiot, and Khalimov (2021), 'Church Synthesis on Register Automata over Linearly Ordered Data Domains'

Proof idea. We show that it is undecidable whether Eve has a winning strategy in the Church data game, since the game formulation is more intuitive to explain one implication. This yields the result, as this is equivalent with deciding whether the specification is realisable, by Proposition 5.2. We reduce the problem from the emptiness problem of Minsky machines with two counters, which is undecidable since they are Turing complete [102, Theorem I]. To a 2-counter machine M , we associate a specification with registers r_1 and r_2 , each containing the value of one counter, plus an additional register t , used to check that no cheating occurs. Let us describe how to increment c_1 (cf Figure 7.4a); the case of c_2 and of decrementing are similar. Eve suggests a value $d > r_1$, which is stored in t . Then, Adam checks that the increment was done correctly: Eve cheated if and only if Adam can provide a data value d' such that $r_1 < d' < d$. If he cannot, d is stored in r_1 , thus updating the value of the counter. Zero-tests are easily simulated, since 0 is a constant of the domain (Figure 7.4b). The acceptance condition is a reachability

[102]: Minsky (1961), 'Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines'

If 0 is not given as a constant, it suffices to check that there is no data value that is strictly lower.



(a) Gadget for incrementing c_1 . k is the instruction number, stored in the state of the specification automaton. $\downarrow$ (respectively, $\backslash o/$) is a sink rejecting (resp. accepting) state. $\vee$ is a shortcut for two distinct transitions.

(b) Gadget for the zero-test.

Figure 7.4. Simulating increment (the gadget for decrement is similar) and zero-tests.

one, asking that a halting instruction is eventually met. Now, if M halts, then its run is easily simulated by a strategy of Eve that provides the values of the counters. Moreover, in that case, its run is finite and the values of the counters are bounded by some B , so Eve's strategy can be computed by a transducer with B registers that are initialised to $1, \dots, B$, that simulates the run by providing the values of the counters along the run. Conversely, if M does not halt, then no halting instruction is reachable by simulating M correctly, and Adam is able to check that Eve does not cheat during its simulation. $\square$

Proof. We now move to the formal proof of the result. We reduce from the emptiness problem of deterministic two-counter machines, which is undecidable [102, Theorem I]. For simplicity, we write the proof for machines with two counters; the reader can observe that a k -counters Machine can be simulated by a specification with $k+1$ registers. Among the multiple formalisations of counter machines, we pick the following one: a 2-counter machine has two counters which contain integers,

initially valued 0. It is composed of a finite set of *instructions* $M = (I_1, \dots, I_m)$, each instruction being of the form $\text{incr}_j, \text{decr}_j, \text{if}\theta_j(k', k'')$ for $j = 1, 2$ and $k', k'' \in \{1, \dots, m\}$, or halt . The semantics are defined as follows: a *configuration* of M is a triple (k, c_1, c_2) , where $1 \leq k \leq m$ and $c_1, c_2 \in \mathbb{N}$. The transition relation (which is actually a function, since M is deterministic) is then, from a configuration (k, c_1, c_2) :

- If $I_k = \text{incr}_1$ (respectively, incr_2), then the machine increments c_1 (resp. c_2) and jumps to the next instruction I_{k+1} : $(k, c_1, c_2) \rightarrow (k+1, c_1+1, c_2)$ (resp. $(k, c_1, c_2) \rightarrow (k+1, c_1, c_2+1)$).
- If $I_k = \text{decr}_1$ and $c_1 > 0$, then $(k, c_1, c_2) \rightarrow (k+1, c_1-1, c_2)$. If $c_1 = 0$, then the computation fails and there is no successor configuration. Similarly for decr_2 .
- If $I_k = \text{if}\theta_1(k', k'')$, then M jumps to k' or k'' according to a zero-test on c_1 : if $c_1 = 0$, then $(k, c_1, c_2) \rightarrow (k', c_1, c_2)$, otherwise $(k, c_1, c_2) \rightarrow (k'', c_1, c_2)$. Similarly for $\text{if}\theta_2$.

A *run* of the machine is then a finite or infinite sequence of successive configurations, starting at $(1, 0, 0)$. We say that M *halts* whenever it admits a finite run which ends in a configuration (k, c_1, c_2) such that $I_k = \text{halt}$. Note that since the machine is deterministic, there is a unique maximal run.

Let $M = (I_1, \dots, I_m)$ be a 2-counter machine. We associate to it the following deterministic specification automaton: S has states $Q = (\{0, \dots, m+1, \frac{1}{2}, \backslash o/\} \times \{\text{i}, \oplus, y, n\})$, and has three registers r_1, r_2 and t . r_1 and r_2 each store the value of one counter and t is used as a buffer. i and $\oplus$ respectively denote input and output states, while y and n are used to remember whether a zero-test evaluated to true or false. The initial state is $(0, \text{i})$, and acceptance is defined by a reachability condition $\Omega = Q^* F Q^\omega$ with $F = \{\backslash o/\} \times \{\text{i}, \oplus\}$. $\frac{1}{2}$ is a sink state that is not accepting. Its transitions are defined as follows:

- Initially, there is a transition $(0, \text{i}) \xrightarrow{\top} (1, \oplus)$ so that Eve can start the simulation.
- $\backslash o/$ and $\frac{1}{2}$ are sink states, so $(\backslash o/, \text{i}) \xrightarrow{\top} (\backslash o/, \oplus) \xrightarrow{\top} (\backslash o/, \text{i})$, and similarly $(\frac{1}{2}, \text{i}) \xrightarrow{\top} (\frac{1}{2}, \oplus) \xrightarrow{\top} (\frac{1}{2}, \text{i})$.
- Then, for each $k \in \{1, \dots, m\}$:
 - If $I_k = \text{incr}_j$ for $j = 1, 2$, then we add the gadget of Figure 7.4a on page 179, i.e. an output transition $(k, \oplus) \xrightarrow{\star > r_1, \downarrow} (k, \text{i})$ and input transitions $(k, \text{i}) \xrightarrow{r_1 < \star < t} (\frac{1}{2}, \oplus)$, $(k, \text{i}) \xrightarrow{\star = t, \downarrow r_1} (k+1, \oplus)$ and $(k, \text{i}) \xrightarrow{\star \leq r_1} (\backslash o/, \oplus)$, $(k, \text{i}) \xrightarrow{\star > t} (\backslash o/, \oplus)$.
 - The case $I_k = \text{decr}_j$ for $j = 1, 2$ is similar: we add an output transition $(k, \oplus) \xrightarrow{\star < r_1, \downarrow} (k, \text{i})$ and input transitions $(k, \text{i}) \xrightarrow{t < \star < r_1} (\frac{1}{2}, \oplus)$, $(k, \text{i}) \xrightarrow{\star = t, \downarrow r_1} (k+1, \oplus)$ and $(k, \text{i}) \xrightarrow{\star \geq r_1} (\backslash o/, \oplus)$, $(k, \text{i}) \xrightarrow{\star < t} (\backslash o/, \oplus)$. Note that in our definition, if $c_j = 0$, the instruction decr_j should be blocking, i.e. the computation should fail, which is consistent with the fact that in that case, the implementation cannot provide $d < r_1$.

- If $I_k = \text{if}\theta_j(k', k'')$, then we add the gadget of Figure 7.4b, i.e. output transitions $(k, \circ) \xrightarrow{\star=r_1=0} (k, y), (k, \circ) \xrightarrow{\star=r_1>0} (k, n)$ and input transitions $(k, y) \xrightarrow{\top} (k', \circ)$ and $(k, n) \xrightarrow{\top} (k'', \circ)$.
- If $I_k = \text{halt}$, we add a transition $(k, \circ) \xrightarrow{\top} (\backslash o/, i)$.

Now, assume that M halts, i.e. it admits a run $\rho = (k_1, c_1^1, c_2^1) \rightarrow \dots \rightarrow (k_n, c_1^n, c_2^n)$, where $n \in \mathbb{N}$, $k_1 = 1$, $c_1^1 = c_2^1 = 0$ and $I_{k_n} = \text{halt}$. Note that for all $l < n$, $I_{k_l} \neq \text{halt}$, since halting instructions do not have a successor configuration. The values of the counters are bounded by some $B \leq n$. We can then define a register transducer T with n states and $B + 1$ registers which ignores its input and outputs $w = c_0^{j_0} \dots c_{n-1}^{j_{n-1}} 0^\omega$, where for $1 \leq l < n$, j_l is the index of the counter modified or tested at step l (i.e. $j_l = 1, 2$ is such that $I_{k_l} = \text{incr}_{j_l}, \text{decr}_{j_l}$ or $\text{if}\theta_{j_l}(k', k'')$). Let us show that it is indeed an implementation, or equivalently that it computes a winning strategy for Eve in the Church data game G_S (see Proposition 5.2). Let $u \in \mathbb{N}^\omega$ be an input data word. We show by induction on l that in S the partial run over $\langle u, w \rangle [l]$ is either in state $\backslash o/$ or $l \leq n$ and S is in configuration (k_l, v_l) , where $v_l(r_1) = c_l^1$ and $v_l(r_2) = c_l^2$. Initially, S is in configuration $(1, v_R^1)$, so the invariant holds. Now, assume it holds up to step l . If S is in $\backslash o/$, the invariant holds at step $l + 1$ as $\backslash o/$ is a sink state. Otherwise, if $l = n$, then S is in configuration (k_n, v_n) with $I_{k_n} = \text{halt}$, so the unique outgoing transition is $(k_n, \circ) \xrightarrow{\top} (\backslash o/, i)$, which means that whatever the input data value $u[l + 1]$, S transitions to $\backslash o/$ and the invariant holds at step $l + 1$. Remains the case where $l < n$ and the current state is not $\backslash o/$. Then, S is in configuration (k_l, v_l) and there are four cases:

- $I_{k_l} = \text{incr}_j$. By definition, $j = j_l$. We treat the case $j = 1$, the other case is similar. Then, T outputs $c_l^1 = c_{l-1}^1 + 1$, which is such that $c_l^1 > v_l(r_1)$. Then, there does not exist d such that $v_l(r_1) < d < v_l(t)$ since $v_l(r_1) = c_{l-1}^1$ and $v_l(t) = c_{l-1}^1 + 1$, so the transition to $\frac{1}{2}$ cannot be taken. Now, either $u[l + 1] = v_l(t) = c_{l-1}^1 + 1$, in which case S transitions to configuration $(k_{l+1}, c_{l+1}^1, c_{l+1}^2)$, or $u[l + 1] \neq v_l(t)$ and S goes to $\backslash o/$; in both cases the invariant holds.
- The case of $I_{k_l} = \text{decr}_j$ is similar. Let us just mention that the computation does not block at this step, otherwise ρ is not a run of M . Thus, $v_l(r_j) > 0$ and the transition $d < r_j$ can indeed be taken.
- $I_{k_l} = \text{if}\theta_j(k', k'')$. Again, $j = j_l$, and we treat the case $j = 1$. T outputs c_l^1 ; there are two cases. If $c_l^1 = 0$, the transition $(k_l, \circ) \xrightarrow{\star=r_1=0} (k', i)$ is taken. Otherwise, $c_l^1 \neq 0$ and S takes the transition $(k_l, \circ) \xrightarrow{\star=r_1>0} (k'', i)$. In both cases, whatever the input, S then evolves to (k_{l+1}, v_{l+1}) (where $v_{l+1} = v_l$) and the invariant holds.
- Finally, we cannot have $I_{k_l} = \text{halt}$ since $l < n$.

As a consequence, $\backslash o/$ is eventually reached whatever the input, which means that for all $u \in \mathbb{N}^\omega$, $\langle u, \llbracket T \rrbracket(u) \rangle \in S$, i.e. T is indeed an implementation of S .

Conversely, assume that S admits an implementation I . Let ρ be the maximal run of M (i.e. either ρ ends in a configuration with no successor, or it is infinite). It is unique since M is deterministic. Let $n = |\rho|$, with the

In the following, to lighten the notations, we omit the mention of i and $\circ$ when it is clear from the context, e.g. we simply write $\backslash o/$ for $(\backslash o/, i)$ or $(\backslash o/, \circ)$, and similarly for $\frac{1}{2}$.

convention that $n = \infty$ if ρ is infinite. Let us build by induction an input data word u such that for all $l < n$, $I(u)[l] = c_l^j$ and the configuration reached by S over $\langle u, I(u) \rangle[l]$ is (k_l, v_l) . In the following, we interpret I as a winning strategy in the Church data game thanks to Proposition 5.2, to better illustrate the interplay between the implementation and the specification. Initially, let $u[0] = 0$. As the initial transition conducts test $\top$, S anyway transitions to state $(1, \circ)$, with $v(r_1) = v(r_2) = 0$.

Now, assume we built the input u up to its l -th data value. There are again four cases:

- $I_{k_l} = \text{incr}_j$. Then, Eve outputs some data value $d_{\circ} > v_l(r_j)$. Assume by contradiction that $d_{\circ} > v_l(r_j) + 1$. Then, Adam can play the input data value $d_{\circ} = v_l(r_j) + 1$, and S goes to state $\frac{1}{2}$, which is a sink rejecting state and Eve loses. So, necessarily, $d_{\circ} = v_l(r_j) + 1 = c_l^j$, and S transitions to configuration (k_{l+1}, v_{l+1}) .
- The case $I_{k_l} = \text{decr}_j$ is similar. Necessarily, $c_l^j > 0$, otherwise Eve cannot provide any output data value and loses. Thus, the computation does not block here.
- $I_{k_l} = \text{if}\theta_j(k', k'')$. The output transitions of the gadget constrains Eve to output $d_{\circ} = v_l(r_j) = c_l^j$, and S transitions to configuration (k_{l+1}, v_{l+1}) .
- $I_{k_l} = \text{halt}$. Then, it means that $n < \infty$ and $l = n$, so the invariant vacuously holds.

Since the acceptance condition is a reachability one, eventually ρ reaches $\backslash o /$, otherwise λ is not a winning strategy. This means that a halt instruction is eventually reached, so ρ is a halting run of M : M halts.

Overall, Eve has a winning strategy in G_S if and only if S admits an implementation, if and only if S admits an implementation that can be computed by a register transducer with initial valuation $v : r_i \mapsto i$ for $i \in \{1, \dots, B\}$, which means that the reactive synthesis problem for two-sided specifications given by deterministic register automata over $(\mathbb{N}, <, 0)$ is undecidable, even if we restrict to register transducer implementations (with an arbitrary initial valuation). $\square$

It is not clear whether this reduction can be adapted to the case of a constant initial valuation $v_R^t : r \mapsto 0$ or $v_R^t : r \mapsto c$ for some $c \in \mathbb{N}$.

Open Problem 7.2 The decidability status of the Church synthesis problem, i.e. when we further restrict to register transducers with initial valuation $v_R^t : r \mapsto 0$ is open.

One-Sided Specifications

In the above proof, it is crucial that Eve is able to provide data values that do not appear in the input, namely the values of the increment of counters. In this section, we show that decidability can be recovered in the case of one-sided specifications.

We already established that for game-sound specification automata, the automaton game yields a sound abstraction of the Church data game (Proposition 7.2). In other words, if Eve has a winning strategy in G_S^f , then she has a winning strategy in G_S , i.e. S is realisable by

In this paragraph, S denotes a deterministic one-sided specification automaton over $(\mathbb{N}, <, 0)$.

some synchronous program (Proposition 5.2). Moreover, for one-sided deterministic specification automata, game-soundness is without loss of generality (Definition 7.2). However, in $(\mathbb{N}, <, 0)$, feasible action sequences do not form an ω -regular language and we cannot turn all register automata into game-ready ones, so Proposition 7.10 does not apply (see Example 7.2 and Property 4.38). The main difficulty is that register automata over $(\mathbb{N}, <, 0)$ exhibit non- ω -regular behaviours: in general, one needs a deterministic max-automaton to recognise feasible action sequences (Theorem 4.46).

However, we show that it actually suffices to consider an ω -regular over-approximation of G_S^f that we call G_S^{reg} . It has the same arena as G_S^f , but its winning condition is more demanding for Eve. Recall that feasible action sequences in $(\mathbb{N}, <, 0)$ are characterised by the absence of infinite descending chains and of increasing ceiled chains. We define quasi-feasible action sequences by relaxing this condition to the absence of infinite descending chains and of *infinite* increasing ceiled chains, and ask that the parity condition of S must be satisfied for all *quasi*-feasible action sequences provided by Adam. The key property of quasi-feasibility is that it coincides with feasibility for ultimately periodic (a.k.a. regular) action sequences (Proposition 7.21). Note that a similar idea has been used in [116, Sections 6 and 7] to solve the satisfiability problem of constraint LTL, which consists in LTL enriched with comparison of data values for a linear order $<$, and Proposition 7.21 is the analogue of [116, Lemma 7.2]. We provide a self-contained proof to get a uniform presentation.

The regular game G_S^{reg} yields a sound abstraction, as it over-approximates G_S^f (any winning strategy of Eve in G_S^{reg} is winning in G_S^f). Completeness is shown by contraposition, using a pumping argument: if Eve loses G_S^{reg} , then Adam wins it with a positional strategy. As we demonstrate, finite-memory strategies cannot distinguish between unboundedly and infinitely increasing chains, so if Adam wins G_S^{reg} , he actually wins G_S^f . Moreover, finite-memoriness sets a bound on the chains provided by Adam, which allows to concretise his strategy with data values in G_S , which means that he also wins G_S .

Remark 7.10 Games with a deterministic max-automaton winning condition can be shown decidable, by expressing the existence of a winning strategy as a weak MSO+U formula over trees [132, Example 2]. However, the complexity is high, and it remains to establish that the automaton game forms a complete abstraction for the synthesis problem, i.e. to show that if Eve does not have a winning strategy in G_S^f , then it means that the specification is not realisable). We were only able to show the latter by going through G_S^{reg} .

Quasi-Feasible Action Sequences Let us first recall Lemma 4.42:

Lemma 4.42 *A consistent constraint sequence κ is feasible in $\mathbb{N}$ if and only if*

- (i) *It has no infinitely decreasing one-way chains, and*
- (ii) *Ceiled one-way chains have a uniformly bounded depth, i.e. there exists a bound $B \in \mathbb{N}$ such that increasing one-way chains of κ that are ceiled have depth at most B .*

Then, we can define quasi-feasible action sequences as follows:

Let us recapitulate the different games at play here:

- G_S is the Church data game over S , where the players' moves consists in data values; a play is winning if and only if it is accepted by S (Definition 5.4)
- G_S^f is the automaton game associated with S , where the players respectively choose input and output transitions of S ; a play is winning for Eve if either the corresponding action sequence is not feasible or the parity condition is satisfied (Section 7.3.1)
- G_S^{reg} (formally introduced later on in Definition 7.8) is the regular automaton game associated with S ; it is the same game as G_S^f except that a play is winning for Eve if either the corresponding action sequence is not *quasi*-feasible or the parity condition is satisfied.

[116]: Demri and D'Souza (2007), 'An automata-theoretic approach to constraint LTL'

[132]: Bojańczyk (2014), 'Weak MSO+U with Path Quantifiers over Infinite Trees'

The reader is invited to read Section 4.7, or at least Section 4.7.2, if they did not do so before (or if they forgot its content), otherwise this section might turn out to be a dense read.

Definition 7.6 (Quasi-feasible action sequence) An action sequence in $(\mathbb{N}, <, 0)$ is quasi-feasible whenever:

- (i) It has no infinitely decreasing chain
- (ii) It has no infinitely increasing ceiled chain.
- (iii) Initially, all registers contain 0, i.e. $\tau_{0|R} = \{r=s \mid r, s \in R\} \cup \{r=0 \mid r \in R\}$, and
- (iv) It has no decreasing one-way chain of depth ≥ 1 from any (r_i, m_i) such that τ_i implies $r_i = 0$.

To lighten the notation, we drop the distinction between feasibility and 0-feasibility.

We denote $\text{QFeasible}(R)$ the set of quasi-feasible action sequences over R . Since any infinitely increasing ceiled chain contains unboundedly increasing ceiled chains, we have that $\text{Feasible}(R) \subseteq \text{QFeasible}(R)$.

All those conditions are ω -regular, which means:

Proposition 7.20 For any set of registers R , the set of quasi-feasible action sequences over R is ω -regular.

More precisely, it is recognised by a deterministic parity automaton with $2^{\text{poly}(|R|)}$ states and $\text{poly}(|R|)$ colours.

Proof. The construction is a direct adaptation of that of Proposition 4.44: a non-deterministic Büchi automaton can easily guess an infinitely decreasing chain, or an infinitely increasing chain and check that it is ceiled, by storing the registers of the chain (and, possibly, of the associated ceiling stable chain) in memory. The two last conditions are easily verified. Finally, determinising it into a parity automaton and complementing it yields the expected result. $\square$

Definition 7.7 We say that an infinite word $w \in A^\omega$ is *ultimately periodic* when it is of the form $w = u \cdot v^\omega$ for $u \in A^*$ and $v \in A^+$.

This notion is also known as *regular word*. We prefer the ‘ultimately periodic’ terminology as it avoids overloading once more the term ‘regular’, which is already widely used in this document. Some also say that those sequences are lasso-shaped; this is the terminology we use in [42].

An important property is that over ultimately periodic action sequences, feasibility and quasi-feasibility coincide, which is reminiscent of [116, Lemma 7.2]:

Proposition 7.21 Let α be an ultimately periodic action sequence. Then, α is quasi-feasible if and only if it is feasible.

Proof. The right-to-left direction results from the fact that $\text{Feasible}(R) \subseteq \text{QFeasible}(R)$. Now, assume that α is not feasible and let κ be its induced constraint sequence. If it has an infinite descending chain, then we are done. Otherwise, we know it has increasing ceiled chains of unbounded depth. Since α is ultimately periodic, so is κ ; write $\kappa = \tau_0 \dots \tau_{k-1}(\tau_k \dots \tau_{k+l})^\omega$. Take an increasing ceiled chain γ of depth greater than $k + |R|l$. Necessarily, this chain passes through the same constraint with twice the same register in the loop $(\tau_k \dots \tau_{k+l})^\omega$, with a ‘<’ in between. Thus, we can construct an infinitely increasing ceiled chain by repeating this chain fragment indefinitely. $\square$

The associated regular game We are now ready to define the regular game G_S^{reg} associated to a specification automaton S , which consists in allowing Adam to play quasi-feasible action sequences.

Definition 7.8 To a deterministic specification register automaton $S = (Q, q^i, \Sigma, \Gamma, R, \delta, \Omega)$ over $(\mathbb{N}, <, 0)$, we associate the *regular automaton game* $G_S^{\text{reg}} = (A_S^f, W_S^{\text{reg}})$, where A_S^f is the same arena as that of G_S^f , i.e. it consists in the automaton S , where vertices are states and moves are transitions (see Section 7.3.1). The acceptance condition is the same as that of G_S^f , except that we only ask that the action sequence provided by Adam is *quasi-feasible*:

$$W_S^{\text{reg}} = \left\{ \begin{array}{l} p_0(\sigma_0, \phi_0, \text{asgn}_0) \\ q_0(\gamma_0, \psi_0, \text{asgn}'_0) \in \text{Plays}(A) \\ p_1 \dots \end{array} \middle| \begin{array}{l} (\phi_0, \text{asgn}_0)(\psi_0, \text{asgn}'_0) \\ (\phi_1, \text{asgn}_1) \dots \in \text{QFeasible}(R) \\ \Rightarrow p_0 q_0 p_1 q_1 \dots \in \Omega \end{array} \right\}$$

Since the set of quasi-feasible action sequences is ω -regular (Proposition 7.20), we get:

Proposition 7.22 *For any specification automaton S over $(\mathbb{N}, <, 0)$ with state space Q and registers R , G_S^{reg} is ω -regular.*

More precisely, it is a parity game with $\mathcal{O}(|Q|2^{\text{poly}(|R|)})$ vertices and $d|R|$ colours.

Recall that a game G is ω -regular when its arena is finite and its winning condition is an ω -regular language (Definition 3.16).

Now, let us establish G_S^{reg} correctly abstracts the reactive synthesis problem for S , and, as a side-product, that the latter is equivalent with the Church synthesis problem, in the sense that it suffices to consider register transducer implementations. This is summarised by the following proposition; its proof is the subject of the rest of the section.

Proposition 7.23 *Let S be a deterministic one-sided specification automaton over $(\mathbb{N}, <, 0)$ that is game-sound. The following are equivalent:*

- (i) *Eve has a winning strategy in G_S^{reg}*
- (ii) *Eve has a finite-memory winning strategy in G_S^{reg}*
- (iii) *Eve has a finite-memory winning strategy in G_S^f*
- (iv) *Eve has a register transducer winning strategy in G_S*
- (v) *Eve has a winning strategy in G_S*

Game-soundness can be ensured without loss of generality through a linear transformation (Assumption 7.3).

Proof. In the following, we let S be a game-sound deterministic one-sided specification automaton over $(\mathbb{N}, <, 0)$.

A winning strategy of Eve in G_S^{reg} translates to one in G_S We start by establishing the sequence of implications from items (i) to (v), as it is easier. The implication (i) $\Rightarrow$ (ii) results from the fact that G_S^{reg} is ω -regular (Proposition 7.22), and those games enjoy finite-memory determinacy (Corollary 3.9). Now, if Eve has a finite-memory winning strategy in W_S^{reg} , then the same strategy is winning in G_S^f , since $W_S^{\text{reg}} \subseteq W_S^f$, so (ii) $\Rightarrow$ (iii). The third implication (iii) $\Rightarrow$ (iv) is precisely Proposition 7.4: a finite-memory winning strategy in the automaton game can be interpreted as a register transducer implementation or, equivalently, a register transducer winning strategy in the Church data game. The last implication of the series, (iv) $\Rightarrow$ (v), is immediate. The other is harder, and relies on finite-memory determinacy of ω -regular games: if Eve cannot win, then Adam has a finite-memory winning strategy; we then show that such a strategy can be translated into a winning strategy in the Church data game G_S . Note that this is not the case for arbitrary

strategies, as witnessed by Example 7.2 (if Adam provides longer and longer sequences of decreasing values depending on the actions of Eve, the concretisation of the tests depends on the future of the play).

If Eve cannot win G_S^{reg} , she cannot win G_S Now, let us move to the last implication $(v) \Rightarrow (i)$; we show it by contraposition. Let us describe the main ideas of the proof. First, if Eve does not have a winning strategy in G_S^{reg} , then Adam has one, and we can even assume that it is finite-memory, since G_S^{reg} is ω -regular. We then show, using a pumping argument, that we can bound the depth of what we call right two-sided chains. This allows us to define a procedure that is able to choose data values that concretise the tests in a way that does not depend on the future of the play. Such a concretisation function finally yields a way to translate the strategy of Adam in the regular automaton game into one that is winning in the Church data game. This implies that Eve does not have a winning strategy in the latter game, which yields the sought implication, and provides determinacy of the Church data game as a side-product.

We now move to the formal proof. Assume that Eve does not have a winning strategy in G_S^{reg} . By finite-memory determinacy of ω -regular games, this means that Adam has a finite-memory winning strategy in G_S^{reg} (Theorem 3.8). The rest of the proof is devoted to establishing the following claim.

Claim 7.24 *If Adam has a finite-memory winning strategy in G_S^{reg} , then he has a winning strategy in G_S .*

Recall that S is a game-sound deterministic one-sided specification automaton, ambient in the whole proof.

The depth of decreasing right two-way chains is bounded First, let us show that the length of decreasing right two-way chains of such a strategy is uniformly bounded.

Definition 7.9 (Right two-way chain) A two-way chain $\gamma = (r_0, m_0) \triangleright_0 (r_1, m_1) \triangleright_1 (r_2, m_2) \triangleright \dots$ is a *right two-way chain* if it does not go to the left of position m_0 , i.e. for all $1 \leq i < |\gamma|$, $m_i \geq m_0$.

Lemma 7.25 *Let λ be a finite-memory strategy of Adam in G_S^{reg} . If λ is winning, then there exists $B \geq 0$ such that for all action sequences β of a play consistent with λ , any decreasing right two-way chain of β has depth at most B .*

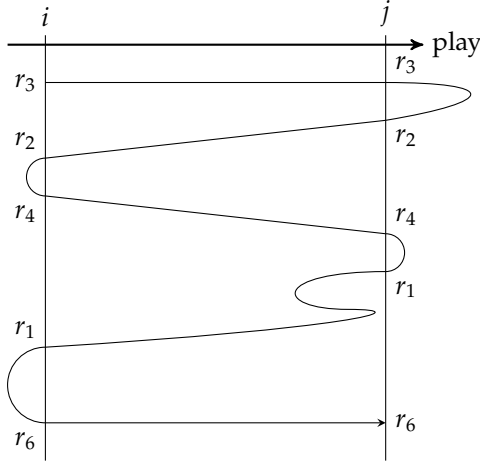
Proof. The main idea is that if Adam has a finite-memory strategy, then if a decreasing right two-way chain γ is sufficiently deep, Eve can force Adam to loop in a memory state in a way such that the loop can be iterated while preserving the chain, and which contains a strictly decreasing or increasing segment. In the first case, iterating the loop yields an infinite descending chain in $\mathbb{N}$, which witnesses that the corresponding action sequence is not feasible. The second case happens when γ is decreasing from right to left (recall that it is a two-way chain), which corresponds to an increasing segment, when read from left to right. When iterated, this yields an infinite increasing chain, which is perfectly legitimate in $\mathbb{N}$. However, it can be bounded from above with the help of γ : before decreasing from right to left, γ has to go from left to right, since it is a right chain, i.e. it is not allowed to go to the left of its initial position. On the strictly increasing segment, this left-to-right prefix is either constant

or decreasing, so when the loop is iterated it provides an upper bound for our increasing chain.

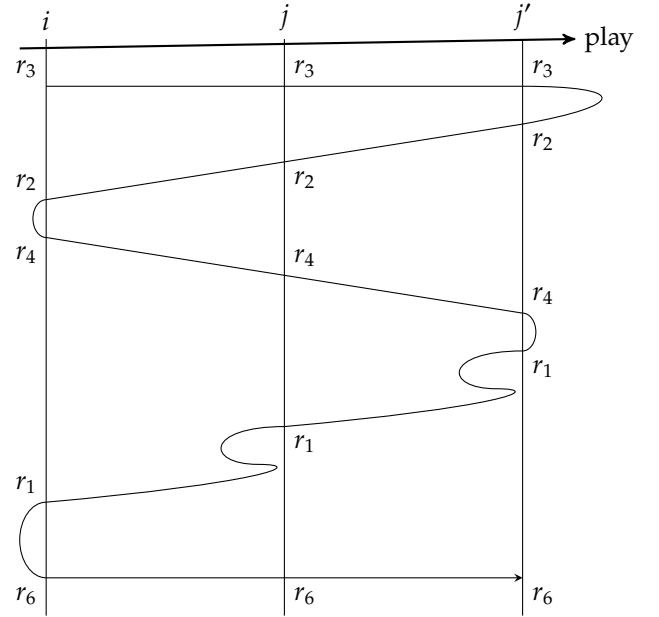
We now move to the formal proof. Let λ be a finite-memory strategy of Adam in G_S^{reg} with memory space M . Assume that λ is winning. Suppose, towards a contradiction, that there exists plays that are consistent with λ and which contain a decreasing right two-way chains of arbitrary depth. At this point, we can use a Ramsey argument in the spirit of Lemma 4.42 to extract an infinite *one-way* chain that is either increasing or decreasing. However, this amounts to breaking a butterfly upon the wheel and we prefer to rely on a simpler pumping argument, which also gives a finer-grained perception of what is happening there. In particular, it provides a bound on B that does not depend on a Ramsey number.

Thus, let ρ be a play consistent with λ whose action sequence admits a decreasing right two-way chain of depth $D > |M| \cdot F_{|R|} T_{|R|}$, where F_k is the k -th Fubini number that counts the number of weak orders over a set of size k and T_k counts the number of ordered subsets of a set of size k (both are exponential in k). We denote it $\gamma = (r_0, m_0) \triangleright_0 (r_1, m_1) \triangleright_1 (r_2, m_2) \triangleright_2 \cdots \triangleright_{n-1} (r_n, m_n)$, where for all $0 \leq i \leq n$, $\triangleright_i \in \{>, =\}$, $r_i \in R$ and $m_i \in \mathbb{N}$. Given a two-way chain and a position $i \geq m_0$, we define the *crossing section* at i as the sequence of registers that occur at position i , ordered by they appearance in the chain: $\langle \gamma \rangle_i$ is the maximal subword of γ that contains letters of the form (r, i) for some $r \in R$ (see Figure 7.5a, where we depicted a chain that has two identical crossing sections at positions i and j). This construction is reminiscent of the techniques that

There is no simple closed form for F_k , so we simply use the fact that it is bounded by 2^{k^2} , which counts the number of binary relations over a given set. $T_k = \binom{k}{0}0! + \binom{k}{1}1! + \cdots + \binom{k}{k}k!$, which is asymptotically equivalent to $ek!$. For the purpose of this argument, we only need that they are finite.



(a) A chain with two identical crossing sections.



(b) Iterating a fragment of a play. We are able to glue the chain since the crossing sections and the order between registers are the same at positions i and j .

are used to study loops in two-way automata or transducers, hence the name. However, here, we can assume without loss of generality that γ is non-looping (Lemma 4.40), so each register appears at most once, which simplifies the argument. At each position, there are $|M|$ distinct memory states for Adam, $T_{|R|}$ many distinct crossing sections and $B_{|R|}$ many

possible orderings of the registers. Thus, by the pigeonhole principle, there exists two positions $m_0 \leq i < j$ such that $\gamma_i = \gamma_j$, the memory state of Adam at position i and j is the same $q \in M$ and the order between registers at position i is the same at position j : $[v_i]_{\mathbb{Q}} = [v_j]_{\mathbb{Q}}$ and there is at least one occurrence of $>$ in the chain fragment. Since Adam's strategy is finite-memory, Eve can repeat her actions between positions i and j indefinitely to iterate this fragment of play. Since the crossing sections match ($\gamma_i = \gamma_j$) and the order between registers is the same at positions i and j , we can glue the chain fragments together to get an infinite two-way chain (see Figure 7.5b on page 187), with infinitely many occurrences of $>$. There are two cases:

Recall that $[v]_{\mathbb{Q}}$ denotes the constraint associated with v , seen as a valuation in $\mathbb{Q}$. It consists in the order relation between the registers, with ties allowed (i.e. we can have $r_i = r_j$).

- There is a fragment that strictly decreases from left to right (as the chain fragment over register r_4 in Figure 7.5b). Then, when Eve repeats her actions indefinitely, this yields an infinite descending chain, which means that the play is not feasible (Lemma 4.41). This contradicts the fact that Adam's strategy is winning.
- All decreasing fragments occur from right to left (as do the fragments over r_2 and r_1 in Figure 7.5b). Necessarily, the topmost fragment, i.e. the fragment of the register that appears first in γ_i , is left-to-right, since γ is a right two-way chain. It is not strictly decreasing, otherwise we are back to the first case. Then, the strictly decreasing fragments are bounded from above by this constant fragment. Iterating the loop yields an infinite increasing chain that is bounded from above, which means that the play is again not feasible, so we also obtain a contradiction.

Overall, we can conclude that there is a uniform bound on the depth of the decreasing right two-way chains induced by λ . $\square$

Bounded right two-way chains yield a data concretisation procedure

Now, let us show that if the depth of decreasing right two-way chains of a strategy of Adam is uniformly bounded, we can concretise his actions with data values in G_S .

Under the assumption that decreasing right two-way chains are bounded, we design a data-assignment function that maps feasible constraint sequence prefixes to register valuations satisfying it, and that is increasing for the prefix order (i.e. if κ is a prefix of κ' , then the image of κ is a prefix of that of κ').

Lemma 7.26 (Data-assignment function) *For every $B \geq 0$, there exists a data-assignment function $f : (\text{Constr}(R) \cup \text{Constr}(R \sqcup R')^+) \rightarrow \mathbb{N}^R$ such that for every finite or infinite consistent constraint sequence $\kappa = \tau_0 \tau_1 \tau_2 \dots$ whose decreasing right two-way chains have depth at most B , the sequence of register valuation $f(\tau_0|_R) f(\tau_0) f(\tau_0 \tau_1) \dots$ is compatible with κ .*

Proof idea. We define a notion of $xy^{(m)}$ -chains that allows to estimate how many insertions between the values of x and y at moment m we can expect in future. As it turns out, if we do not know the future, the distance between x and y has to be exponential in the maximal depth d of $xy^{(m)}$ -chains. Indeed, at each step, the worst case happens if some interval has to be divided in two parts, which can happen at most d times.

We thus construct a data-assignment function that maintains such exponential distances. If the constraint inserts a register x between two registers r and s with values d_r and d_s , then set $d_x = \lfloor \frac{d_r + d_s}{2} \rfloor$; and if the constraint puts a register x above all other registers, then set $d_x = d_M + 2^B$ where d_M the largest value currently held in the registers and B is the given bound on the depth of decreasing right two-way chains. $\square$

Proof. We now move to the formal proof.

$xy^{(m)}$ -connecting chains and the exponential nature of register valuations Fix an arbitrary 0-feasible constraint sequence $\tau_0\tau_1\dots$ whose decreasing right two-way chains have a depth bounded by B . Consider a moment m and two registers x and y such that τ_m implies $x > y$. We would like to construct witnessing valuations $v_0v_1\dots$ using the current history only. More precisely, $\tau_0\dots\tau_{m-1}$, we want to build a register valuation v_m that is suitable for all possible suffixes constraint sequences that induce right two-way chains with depth bounded by B .

Let us see how much space we might need between $v_m(x)$ and $v_m(y)$.

In Figure 7.6, consider decreasing right two-way chains that start at moment $i \leq m$ and end in (x, m) (shown in blue). They are defined within time moments $\{i, \dots, m\}$. Symmetrically, consider left two-way chains that start in (y, m) and end at moment $j \in \{i, \dots, m\}$ (shown in pink), defined within time moments $\{j, \dots, m\}$. Among such chains, pick a pair of chains α and β of respective depths d_α and d_β that maximise the sum $d_\alpha + d_\beta$. After seeing $\tau_0 \tau_1 \dots \tau_{m-1}$, we do not know how the constraint sequence will evolve in future, but by boundedness of right two-way chains, any right two-way chain γ starting in (x, m) and ending in (y, m) (defined within time moments $\geq m$) will have depth $d_\gamma \leq B - d_\alpha - d_\beta$. Otherwise, we could append to it the prefix α and the postfix β and construct a right two-way chain of depth greater than B . As a consequence, we get that $v_m(x) - v_m(y) \geq B - d_\alpha - d_\beta$, since the distance between the values of two registers should be greater than or equal to the longest right two-way chain connecting them.

To formalise the argument, we introduce the notion of $xy^{(m)}$ -connecting chains, which consists in the parts α and β , where x and y are directly connected (i.e. we do not have segments like $(x, m) > (z, m) > (y, m)$ or $(x, m) > (z, m + 1) > (y, m)$). An $xy^{(m)}$ -connecting chain is a right two-way chain of the form $(a, i) \triangleright_i \dots (x, m) > (y, m) \triangleright_m \dots \triangleright_{j-1} (b, j)$: it starts in (a, i) and ends in (b, j) , where $i \leq j \leq m$ and $a, b \in R$, and it *directly* connects x to y at moment m . Note that it is located solely within moments $\{i, \dots, m\}$. Continuing the previous example, the $xy^{(m)}$ -connecting chain starts with α , directly connects $(x, m) > (y, m)$, and ends with β ; its depth is $\alpha + \beta + 1$ (we have ‘+1’ no matter how many registers are between x and y , since x and y are connected directly).

With this new notion, the requirement $v_m(x) - v_m(y) \geq B - \alpha - \beta$ becomes $v_m(x) - v_m(y) \geq B - d_{xy} + 1$, where d_{xy} is the largest depth of $x y^{(m)}$ -connecting chains.

However, since we do not know how the constraint sequence evolves after $\tau_0 \dots \tau_{m-1}$, we might need more space between the registers at

To lighten the argument, we sometimes say ‘right two-way chain’ for decreasing right two-way chain. No ambiguity arises since we only consider decreasing chains.

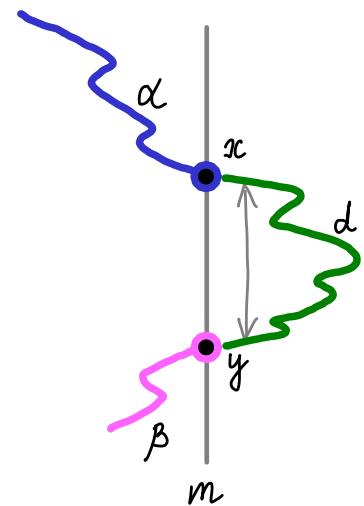


Figure 7.6.: An $xy^{(m)}$ -connecting chain.

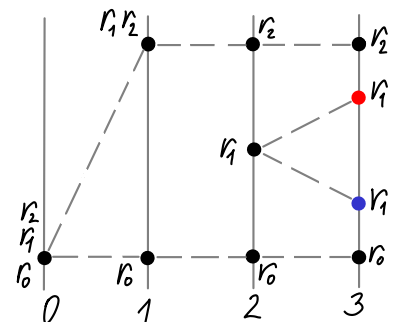


Figure 7.7.: Two possible suffix constraints, that impose to leave sufficient room between r_0, r_1 and r_2 .

moment m . Consider Figure 7.7, with $R = \{r_0, r_1, r_2\}$ and a bound $B = 3$ on the depth of right two-way chains.

- Suppose at moment 1, after seeing the constraint τ_0 , which is $\{r'_1, r'_2\} > \{r_0, r_1, r_2, r'_0\}$, the valuation is $v_1 = \{r_0 \mapsto 0; r_1, r_2 \mapsto 3\}$. It satisfies $v_1(r_2) - v_1(r_0) \geq B - d_{r_2 r_0} + 1$ (indeed, $B = 3$ and $d_{r_2 r_0} = 1$ at this moment); similarly for $v(r_1) - v(r_0)$.
- Let the constraint τ_1 be $\{r_1, r_2, r'_2\} > \{r'_1\} > \{r_0, r'_0\}$. What value $v_2(r_1)$ should register r_1 take at moment 2? Note that the assignment should work no matter what τ_2 is (provided the depth of right two-way chains does not exceed B). Since the constraint τ_1 asks that r_1 is between r_0 and r_2 at moment 2, we can only assign $v_2(r_1) = 2$ or $v_2(r_1) = 1$. If we choose 2, then the constraint τ_2 having $\{r_2, r'_2\} > \{r'_1\} > \{r_1\} > \{r_0, r'_0\}$ (the red dot in the figure) shows that there is not enough space between r_2 and r_1 at moment 2 ($v_2(r_2) = 3$ and $v_2(r_1) = 2$). Symmetrically for $v_2(r_1) = 1$: the constraint τ_2 having $\{r_2, r'_2\} > \{r_1\} > \{r'_1\} > \{r_0, r'_0\}$ (the blue dot in the figure) kills any possibilities for a correct assignment.

Thus, at moment 2, the register r_1 should be equally distanced from r_0 and r_2 , i.e. $v_2(r_1) \approx \frac{v_2(r_0) + v_2(r_2)}{2}$, since its evolution can go either way, towards r_2 or towards r_0 . This is why we need a space exponential in the depth of connecting chains between the registers.

The data-assignment function We now build the data-assignment function $f : (\text{Constr}(R) \cup \text{Constr}(R \sqcup R'))^+ \rightarrow \mathbb{N}^R$ by induction on the length of $\tau_0 \dots \tau_{m-1}$ as follows.

Initially, $f(\tau_0|_R) = v_0$ where $v_0(r) = 0$ for all $r \in R$ (recall that since f takes as input consistent constraint sequences, τ_0 is $\bigwedge_{r,s \in R} r = s \wedge \bigwedge_{r \in R} r = 0$).

Assume that at moment m , the register valuation is $v_m = f(\tau_0|_R \tau_0 \dots \tau_{m-1})$. Let C_m be the next constraint. Then, we define the register valuation $v_{m+1} = f(\tau_0|_R \tau_0 \dots \tau_m)$ as:

1. If a register x at moment $m + 1$ lies above all registers at moment m , i.e. τ_m implies $x' > r$ for every register r , then we set $v_{m+1}(x) = v_m(r) + 2^B$, where r is a register whose value is maximal at moment m . Note that this happens when the test of the action sequence at moment m asks $\bigwedge_{r \in R} \star > r$.
2. If a register x at moment $m + 1$ lies between two adjacent registers $a > b$ at moment m , then we set $v_{m+1}(x) = \lfloor \frac{v_m(a) + v_m(b)}{2} \rfloor$. This is the case when the test contains $a > \star > b$ and for each $r \neq a, b$, either $r \geq a$ or $r \leq b$.
3. If a register x at moment $m + 1$ is equal to a register r at previous moment m , i.e. τ_m implies $r = x'$, then we let $v_{m+1}(x) = v_m(r)$. This corresponds to tests of the form $\star = r$.

Let us now show that the following invariant holds:

Claim 7.27 *The data-assignment function f satisfies the following property:*

For all $m \in \mathbb{N}$ and all $\forall x, y \in R$ such that τ_m implies $x > y$, $v_m(x) - v_m(y) \geq 2^{B-d_{xy}}$

where d_{xy} is the largest depth of $xy^{(m)}$ -connecting chains and B is the bound on right two-way chains.

Proof. The invariant holds initially since all registers are initially equal, i.e. τ_0 asks that $r = s = 0$ for all $r, s \in R$.

Now, assume that it holds at step m ; let us show that it holds at step $m + 1$. Fix two registers $x, y \in R$ such that τ_m implies $x' > y'$. We need to show that $v_{m+1}(x) - v_{m+1}(y) \geq 2^{B-d_{xy}}$, where d_{xy} is the largest depth of $xy^{(m+1)}$ -connecting chains. There are four cases, depending on whether the values of x and y at moment $m + 1$ are present at moment m or not.

Case 1: both are present The values of x and y at $m + 1$ are also present at moment m . This corresponds to item 3 in the definition of f . Then, let a, b be registers such that τ_m implies $a > b$, as all as $a = x'$ and $b = y'$. Thus, we have $v_m(a) = v_{m+1}(x)$ and $v_m(b) = v_{m+1}(y)$. Note that there might be less register values between x and y than between a and b (depicted as black dots in Figure 7.8). Consider the depths of connecting chains for $ab^{(m)}$ and $xy^{(m+1)}$. Since every $ab^{(m)}$ -connecting chain can be extended to an $xy^{(m+1)}$ -connecting chain of the same depth as shown on the figure, we have $d_{ab} \leq d_{xy}$, hence $2^{B-d_{ab}} \geq 2^{B-d_{xy}}$. As a consequence, $v_{m+1}(x) - v_{m+1}(y) = v_m(a) - v_m(b) \geq 2^{B-d_{ab}} \geq 2^{B-d_{xy}}$.

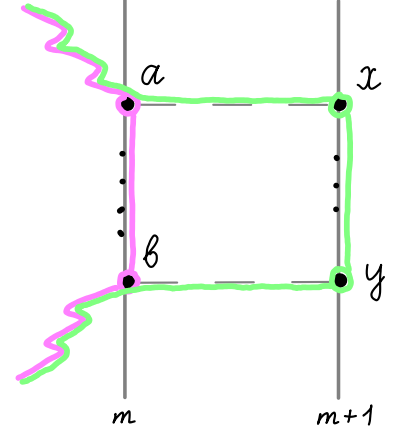


Figure 7.8.: Both x and y were already present.

Case 2: x is a new value at the top Now, assume that the register x is greater than all values of moments m and $m + 1$, and y has a value that was also present at moment m (cf Figure 7.9). This corresponds to item 1 in the definition of f . Then, we have that τ_m implies $x' > r$ for all $r \in R$, and that $y' = b$ for some $b \in R$. Let a be a register whose value is maximal at moment m , i.e. such that τ_m implies $r \leq a$ for all $r \in R$ (a and b may coincide). Thus, $v_{m+1}(x) = v_m(a) + 2^B$. The invariant holds for x, y because $v_{m+1}(x) = v_m(a) + 2^B$ and $v_m(a) \geq v_m(b) = v_{m+1}(y)$.

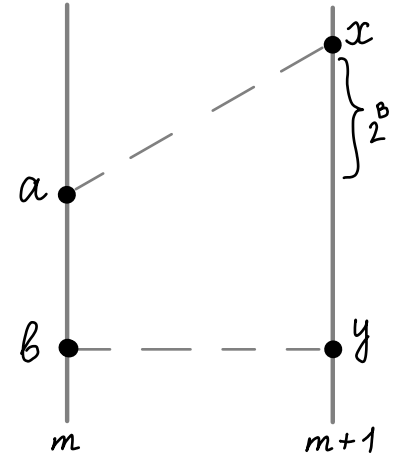


Figure 7.9.: x is a new value at the top

Case 3: x is inserted between two registers, y was already present We now treat the case where the register x at moment $m + 1$ takes a value that is between a and b , i.e. τ_m implies $a > x > b$. We take optimal a and b , i.e. such that either $r \geq a$ or $r \leq b$ for all $r \in R$. This corresponds to item 2. Then, $v_{m+1}(x) = \lfloor \frac{v_m(a) + v_m(b)}{2} \rfloor$. We also assume that the register y at moment $m + 1$ is equal to some register c at moment m , i.e. τ_m implies $c = y'$. Note that c and b may coincide. Then,

$$\begin{aligned} v_{m+1}(x) - v_{m+1}(y) &= \left\lfloor \frac{v_m(a) + v_m(b)}{2} \right\rfloor - v_m(c) \\ &= \left\lfloor \frac{v_m(a) - v_m(c)}{2} + \frac{v_m(b) - v_m(c)}{2} \right\rfloor \\ &\geq \left\lfloor \frac{v_m(a) - v_m(c)}{2} \right\rfloor + \left\lfloor \frac{v_m(b) - v_m(c)}{2} \right\rfloor \\ &\geq \lfloor 2^{B-d_{ac}-1} \rfloor + \lfloor 2^{B-d_{bc}-1} \rfloor \\ &\geq 2^{B-d_{ac}-1} + \lfloor 2^{B-d_{bc}-1} \rfloor \end{aligned}$$

We need to show that the $2^{B-d_{ac}-1} + \lfloor 2^{B-d_{bc}-1} \rfloor \geq 2^{B-d_{xy}}$. In Figure 7.10, we show how the green $xy^{(m+1)}$ -connecting chain can be constructed from the pink $ac^{(m)}$ -connecting chain, which implies that $d_{xy} \geq d_{ac} + 1$, so we get $2^{B-d_{ac}-1} \geq 2^{B-d_{xy}}$. Overall, $v_{m+1}(x) - v_{m+1}(y) \geq 2^{B-d_{ac}-1} + \lfloor 2^{B-d_{bc}-1} \rfloor \geq 2^{B-d_{xy}}$.

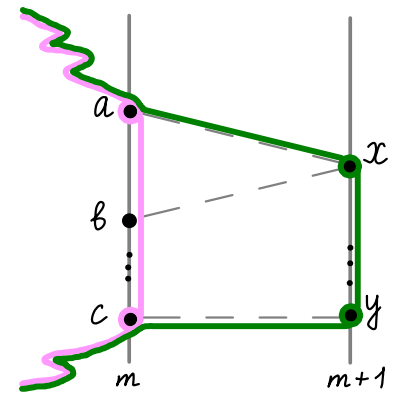


Figure 7.10.: x is inserted between two registers

Case 4: x was present, y is inserted in between The case is similar to the previous one, but we prove it for completeness. Assume that τ_m contains $a = x'$, $x' > y'$, $b > y' > c$, where b and c are adjacent (a and b might be the same). Then,

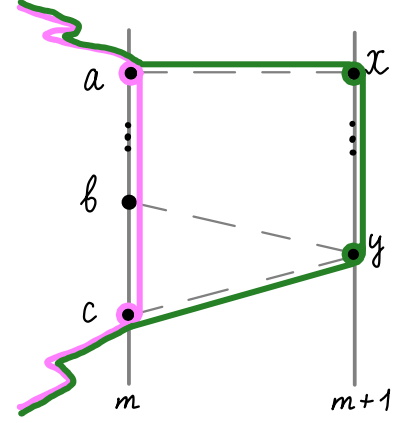
$$\begin{aligned}
 v_{m+1}(x) - v_{m+1}(y) &= v_m(a) - \left\lfloor \frac{v_m(b) + v_m(c)}{2} \right\rfloor \\
 &\geq \left\lfloor \frac{v_m(a) - v_m(b)}{2} \right\rfloor + \left\lfloor \frac{v_m(a) - v_m(c)}{2} \right\rfloor \\
 &\geq \left\lfloor \frac{v_m(a) - v_m(b)}{2} \right\rfloor + \left\lfloor \frac{v_m(a) - v_m(c)}{2} \right\rfloor \\
 &\geq \lfloor 2^{B-d_{ab}-1} \rfloor + \lfloor 2^{B-d_{ac}-1} \rfloor \\
 &\geq \lfloor 2^{B-d_{ab}-1} \rfloor + 2^{B-d_{ac}-1}
 \end{aligned}$$

Since $d_{ac} + 1 \leq d_{xy}$, we get that $v_{m+1}(x) - v_{m+1}(y) \geq \lfloor 2^{B-d_{ab}-1} \rfloor + 2^{B-d_{ac}-1} \geq 2^{B-d_{xy}}$. End of the proof of Claim 7.27. $\square$

From the invariant (Claim 7.27), it is then routine to show that for every finite or infinite consistent constraint sequence $\kappa = \tau_0 \tau_1 \tau_2 \dots$ whose decreasing right two-way chains have depth at most B , the sequence of register valuation $f(\tau_0|_R) f(\tau_0) f(\tau_0 \tau_1) \dots$ is compatible with κ . We write the argument for completeness.

Our goal is to show that for every atomic formula $(r \bowtie s)$ (respectively, $(r \bowtie s')$) of τ_m , where $r, s \in R$ and $\bowtie \in \{<, >, =\}$, the expressions $v_m(r) \bowtie v_m(s)$ (resp., $v_m(r) \bowtie v_{m+1}(s)$) hold. We treat each case separately:

- If τ_m implies $r = s$ (respectively, $r = s'$ or for some $r, s \in R$, then item 3 implies $v_m(r) = v_m(s)$ (resp., $v_m(r) = v_{m+1}(s)$).
- If τ_m implies $r > s$, then $v_m(r) > v_m(s)$ by the invariant (Claim 7.27).
- Assume that τ_m implies $r > s'$, and that the value of s at moment $m+1$ is present at moment m , i.e. there is a register t such that τ_m implies $t = s'$. Then τ_m also implies $r > t$, since it is maximally consistent. Since $v_m(t) = v_{m+1}(s)$ (item 3) and since $v_m(r) > v_m(t)$ (as τ_m implies $r > t$), we get $v_m(r) > v_{m+1}(s)$.
- Similarly for the case τ_m implies $r < s'$, where s takes a value that is also present at moment m .
- Now, if τ_m implies $r < s'$ and if the value of s is maximal at moments m and $m+1$, then $v_m(r) < v_{m+1}(s)$ because $v_{m+1}(s) \geq v_m(r) + 2^B$ by item 1.
- Finally, there are two cases left, namely τ_m implies $r > s'$ or $r < s'$, where s takes a new value at moment $m+1$, and there are higher values at moment m . This corresponds to item 2. Let a and b be two registers that are adjacent in τ_m , i.e. such that τ_m implies $a > b$ and $r \geq a$ or $r \leq b$ for all $r \in R$, and such that s is inserted between a and b , i.e. τ_m implies $a > s' > b$. Let d_{ab} be the maximal depth of $ab^{(m)}$ -connecting chains; fix one such chain. We change it by going through s at moment $m+1$, i.e. we substitute the part $(a, m) > (b, m)$ by $(a, m) > (s, m+1) > (b, m)$. The depth of the resulting chain is $d_{ab} + 1$ and it is bounded by B by boundedness of right two-way chains. Thus, $d_{ab} \leq B - 1$, so $v_m(a) - v_m(b) \geq 2$, implying $v_m(a) > \lfloor \frac{v_m(a) + v_m(b)}{2} \rfloor > v_m(b)$. When



τ_m implies $r > s'$, we get $v_{m+1}(r) \geq v_m(a)$, and when it implies $r < s'$, we get $v_{m+1}(r) \leq v_m(b)$, therefore we are done.

End of the proof of Lemma 7.26

□

It remains to put everything together, by instantiating Adam's strategy λ with concrete data values in the Church data game G_S . By Lemma 7.25, we know that the depth of the right two-way chains of the plays that are consistent with λ is uniformly bounded by some $B \in \mathbb{N}$. Instantiation is done through to the function f defined in Lemma 7.26: the idea is to construct a strategy $\lambda^{\mathbb{N}}$ in G_S from λ by translating the finite action sequence that corresponds to the current history $\phi_0 \text{asgn}_0 \dots \phi_m \text{asgn}_m$ into a finite constraint sequence through f . Then, by construction, for each play in G_S consistent with $\lambda^{\mathbb{N}}$, the corresponding run ρ in S is a play consistent with λ in G_S^{reg} . As λ is winning for Adam, ρ is not accepting, which means that the play is winning for Adam in G_S . Therefore, $\lambda^{\mathbb{N}}$ is a winning strategy for Adam in G_S , meaning that Eve loses G_S .

Let us be more precise: we build $\lambda^{\mathbb{N}}$ by induction on the length of the play. Assume that $d_0 \dots d_{k-1}$ has been played by Adam, and that Eve played $e_0 \dots e_{k-1}$ (we treat the initial case of the induction simultaneously by allowing $k = 0$). We want to know the value d_k for $\lambda^{\mathbb{N}}(e_0 \dots e_{k-1})$. Let $\rho = (p_0, v_0)(\phi_0, \text{asgn}_0)(q_0, v'_0)(\phi'_0, \text{asgn}'_0)(p_1, v_1)(\phi_1, \text{asgn}_1)(q_1, v'_1) \dots (\phi'_{k-1}, \text{asgn}'_{k-1})(p_k, v_k)$ be the partial run of S over $\langle d_0 \dots d_{k-1}, e_0 \dots e_{k-1} \rangle$. By the induction hypothesis, this corresponds to a play that is consistent with λ in G_S^{reg} (this is initially the case). Let $(\phi_k, \text{asgn}_k) = \lambda(\rho)$ be the next transition of S chosen by Adam in G_S^{reg} .

Let $\tau_0 \dots \tau_{k-1}$ be the constraint sequence built from $(\phi_0 \text{asgn}_0 \phi_k \text{asgn}_k)$ by the mapping *constr* of Lemma 4.45. By Lemma 7.25, we know that the strategy λ induces constraint sequences over R whose right two-way chains have depth at most B . Thus, we can apply the data-assignment function from Lemma 7.26. This yields a register valuation v_{k+1} , which provides a value for $\star$ (for simplicity, we can take v_{k+1} over $R \sqcup \{\star\}$ and let $d_k = v_{k+1}(\star)$). Then, S transitions to configuration (p_{k+1}, v_{k+1}) , and G_S^{reg} is now in vertex p_k . Thus, the extended play is consistent with Adam's strategy λ . This way, we build a strategy $\lambda^{\mathbb{N}}$ in G_S whose plays all correspond to plays that are consistent with λ in G_S^{reg} . Thus, all plays of G_S that are consistent with $\lambda^{\mathbb{N}}$ correspond to interactions $d_0 e_0 d_1 e_1 \dots$ whose run in S is not accepting: $\lambda^{\mathbb{N}}$ is winning in G_S .

As a consequence, if Eve does not win G_S^{reg} , she does not win G_S . End of the proof of Proposition 7.23

□

Remark 7.11 A corollary of the above proof is that G_S is determined: either Adam has a winning strategy, or Eve has one.

The above proposition states that G_S^{reg} forms a sound and complete abstraction for G_S , which implies the following:

Theorem 7.28 ([42, Theorem 17]) *The reactive synthesis problem and the Church synthesis problem for specifications given as deterministic one-sided register automata over $(\mathbb{N}, <, 0)$ coincide and are both EXPTIME-complete.*

Moreover, it suffices to target implementations that use the same registers as the implementation, and in the same way.

[42]: Exibard, Filiot, and Khalimov (2021), 'Church Synthesis on Register Automata over Linearly Ordered Data Domains'

Proof. Let S be a deterministic one-sided specification register automaton over $(\mathbb{N}, <, 0)$ with state space Q and registers R . By Proposition 7.23, we know that Eve has a winning strategy in G_S^{reg} if and only if she has one in G_S (items (i) and (v)). Solving G_S is equivalent to solving the reactive synthesis problem for S (Proposition 5.2), so, overall, it amounts to solving G_S^{reg} . By Proposition 7.22, this game has $|Q|2^{\text{poly}(|R|)}$ vertices and $\mathcal{O}(|R|)$ colours. Using Zielonka's algorithm [88] (cf Theorem 3.12), we can solve it in time $\mathcal{O}((|Q|2^{\text{poly}(|R|)})^{|R|})$, which is exponential in $|R|$, so exponential in the size of S .

Moreover, again by Proposition 7.23, we know that Eve wins G_S if and only if she wins G_S^f with a finite-memory strategy (items (i) and (iii)). By Proposition 7.4, this means that if S has an implementation, she has one that is computed by a register transducer, which means that reactive synthesis is equivalent with Church synthesis for those specifications.

Finally, the ExpTime lower bounds results from the ExpTime -hardness of the case of $(\mathbb{D}, =, \mathbb{C})$ (Theorem 7.12). $\square$

[88]: Zielonka (1998), 'Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees'

Partial Versus Total Domain

8.

In this short chapter, we discuss the connection between synthesis and uniformisation of relations, defined in Section 3.2.1 (Definition 3.1). In our context, this distinction allows to consider the question of the domain of the specification: as pointed out in Remark 3.5, synthesis usually targets implementations that have a total domain; in particular, specifications whose domain is not total are considered unrealisable. This is the case in particular for register transducers (Definition 5.2): by definition, for any register transducer T , we have $\text{dom}(\llbracket T \rrbracket) = (\Sigma \times \mathbb{D})^\omega$, so if a specification S is realisable by a register transducer T (i.e. $\llbracket T \rrbracket \subseteq S$), we have $\text{dom}(S) = (\Sigma \times \mathbb{D})^\omega$.

However, it might happen that the user provides a specification whose domain is not total, but would be content with an implementation that is correct on inputs that belong to the domain (i.e. that admit at least one corresponding acceptable behaviour). This is the case for instance if they are only interested in a subset of behaviours; e.g., they know that all input data values are pairwise distinct, or, to the contrary, that only a bounded number of distinct data values are input (as is the case for parameterised synthesis [34]), so they do not care about what happens for inputs that break this property. In the finite alphabet setting, the formalisms used to express specifications are closed under complement (whether we consider MSO, LTL or ω -automata), and it is actually not a restriction to assume that the domain of the specification is total: it suffices to complete the specification by allowing any behaviour on inputs that do not belong to the domain. Any implementation satisfying this extended specification satisfies it over its domain. This question has recently been studied under the name of *good-enough synthesis* in the context of LTL specifications [133]. In particular, the construction that we sketched for allowing arbitrary outputs on the complement of the domain is detailed in Theorem 1, which states that the more general problem of synthesising good-enough implementations is 2-ExpTime-complete, as is the case when targeting implementations that accept all inputs [13].

The landscape radically changes in the infinite alphabet case, for specifications given by register automata: since they are not closed under complement, the above approach does not work anymore, and the problem actually becomes undecidable.

This retrospectively justifies our efforts to provide undecidability proof for specifications whose domain is known to be total, so as to remove what is already a source of undecidability (Theorems 6.19, 6.20 and 7.1).

8.1. Domains of Specification Automata

First, let us show that one cannot decide whether a specification register automaton has total domain, already when it is deterministic.

Theorem 8.1 ([41, Theorem 5.1]) *Given a deterministic specification automaton S over $(\mathbb{D}, =, \mathbb{C})$, it is undecidable whether its domain is total.*

[34]: Bérard et al. (2020), ‘Parameterized Synthesis for Fragments of First-Order Logic Over Data Words’

[133]: Almagor and Kupferman (2020), ‘Good-Enough Synthesis’

[13]: Pnueli and Rosner (1989), ‘On the Synthesis of a Reactive Module’

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

Proof. We reduce from the universality problem of non-deterministic register automata, which is undecidable [101, Theorem 5.3] (see also Theorem 4.21). The main idea is to encode the language of a NRA as the domain of some deterministic specification register automaton: the input transitions are the same as those of the original automaton, but when there is some non-determinism, its resolution is postponed to the corresponding output transition, whose label corresponds to the chosen transition. In the vocabulary of games, we make Adam choose the label and the test over the input data value, while Eve is in charge of solving non-determinism.

We move to the formal proof. Let $A = (Q, I, \Sigma, R, \Delta, \Omega)$ be a non-deterministic register automaton over $(\mathbb{D}, =, \mathbb{C})$. Assume without loss of generality that its tests consist in maximally consistent tests (Proposition 4.5). We construct a deterministic specification register automaton S_A over $(\mathbb{D}, =, \mathbb{C})$ with label alphabet $\Sigma \sqcup \delta$ recognising the relation

$$R_A = \left\{ \left(\begin{array}{l} (\sigma_1, d_1)(\sigma_2, d_2) \dots, \\ (t_1, d_1)(t_2, d_2) \dots \end{array} \right) \mid \begin{array}{l} t_1 t_2 \dots \text{ is a run of } A \\ \text{over } (\sigma_1, d_1)(\sigma_2, d_2) \dots \end{array} \right\} \quad (8.1)$$

Define $S_A = (Q \sqcup Q \times (\Sigma \times \text{MCTests}(R)), \{q_0\}, \Sigma \sqcup \delta, R \sqcup \{r_0\}, \delta', \Omega')$, where transitions are defined as follows. Let $q \in Q$, $\sigma \in \Sigma$ and $\phi \in \text{MCTests}(R)$.

We define the input transition $p \xrightarrow[\text{S}_A]{\sigma, \phi, \{r_0\}} (p, (\sigma, \phi))$, that simply consists in storing the input label and test in memory, and the input data value in r_0 . Then, for all transitions $t = p \xrightarrow[\text{A}]{\sigma, \phi, \text{asgn}} q \in \delta$, we define the output

transition $(p, (\sigma, \phi)) \xrightarrow[\text{S}_A]{t, \phi \wedge (\star=r_0), \text{asgn}} q$ that consists in outputting the label t and simulating the transition. The acceptance condition only depends on the states of A , and is given as $\Omega' = \pi_1^{-1}(\Omega)$. By construction, S_A is deterministic and recognises the relation R_A of Equation 8.1. Thus, $\text{dom}(S_A)$ is total if and only if $L(A) = (\Sigma \times \mathbb{D})^\omega$. This concludes the proof. $\square$

This result obviously extends to non-deterministic and universal specification register automata, since deterministic ones form a special case. Note that the unbounded synthesis problem for deterministic register automata specifications is not reducible to deciding whether the domain is total: if the specification is not realisable, it is not possible in general to determine whether it is because its domain is not total or because it is not realisable by a sequential machine (e.g. it asks to output right away a data value that will only be input in the future).

As a corollary, observe that it is undecidable whether the domain of a deterministic specification register automaton is recognisable by a deterministic register automaton.

Corollary 8.2 *There is no algorithm that, given a deterministic specification automaton S over $(\mathbb{D}, =, \mathbb{C})$, exhibits a deterministic register automaton D such that $L(D) = \text{dom}(S)$ if it exists and answers No otherwise.*

Proof. The set of all data words is trivially recognisable by a deterministic register automaton which simply ignores its input and accepts. Thus, we can reduce from the above problem as follows: let S be a DRA specification. If its domain cannot be recognised by a deterministic register automaton,

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

Recall that maximally consistent tests consist in maximally consistent conjunctions of atomic formulas. We take this form as this allows to restrict to a finite set of tests.

then in particular it cannot be the set of all data words. If it is, compute some deterministic register automaton D that recognises $\text{dom}(S)$ and check that $L(D) = (\Sigma \times \mathbb{D})^\omega$. This can be done by checking whether $D^c = \emptyset$, since complementing deterministic register automata amounts to complementing their acceptance condition and emptiness is decidable for (non-)deterministic register automata. $\square$

8.2. Uniformisation Problem

As a consequence, it is not possible to preserve decidability if one equips register transducers with an acceptance condition and considers the uniformisation problem. Recall that $f : (\Sigma \times \mathbb{D})^\omega \rightarrow (\Gamma \times \mathbb{D})^\omega$ uniformises $R \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ whenever (Definition 3.1):

- (i) $\text{dom}(f) = \text{dom}(R)$
- (ii) $f \subseteq R$, i.e. for all $x \in \text{dom}(f)$, $(x, f(x)) \in R$

Then, the uniformisation problem is defined as

Problem 3.1: UNIFORMISATION PROBLEM

- Input:** A relation $R \in \mathcal{R}$
- Output:** A finitely represented function $f \in \mathcal{F}$ which uniformises R
if it exists
No otherwise

Formally, a register transducer $T = (Q, q^i, \Sigma, \Gamma, R, \delta)$ can be equipped with an acceptance condition $\Omega \subseteq Q^\omega$; it then recognises the partial function that associates an input data word with the output of the unique run only if it is accepting (i.e. its sequence of states satisfy Ω). It is a simple observation that the domain of such functions is recognisable by a deterministic register automaton. Thus, by Corollary 8.2 one gets:

Theorem 8.3 *The uniformisation problem for specifications given by deterministic register automata to register transducers with an acceptance condition is undecidable.*

8.3. Good-Enough Synthesis

We can finally show that the good-enough variant of the register-bounded synthesis problem for specifications given by universal register automata is undecidable. Formally, it is defined as follows:

Problem 8.1: GOOD-ENOUGH REGISTER-BOUNDED SYNTHESIS PROBLEM
OVER DATA WORDS

- Input:** A data word specification $S \subseteq (\Sigma \times \mathbb{D})^\omega \times (\Gamma \times \mathbb{D})^\omega$ and an integer bound $k \geq 1$
- Output:** A register transducer T with k registers computing a function $f_T : (\Sigma \times \mathbb{D})^\omega \rightarrow (\Gamma \times \mathbb{D})^\omega$
such that for all $x \in \text{dom}(S)$, $(x, f_T(x)) \in S$
if it exists
No otherwise

This is in contrast with Theorem 6.14.

[41]: Exibard, Filiot, and Reynier (2021), ‘Synthesis of Data Word Transducers’

Theorem 8.4 ([41, Theorem 5.2]) *The good-enough register-bounded synthesis problem for specifications given by universal register automata over $(\mathbb{D}, =, \mathfrak{C})$ is undecidable.*

Proof. The proof is a direct reduction from the emptiness problem of universal register automata, which is undecidable ([101, Theorem 18] and Theorem 4.50).

Let A be a universal register automaton. Consider the specification

$$S = \left\{ (w\$u(\sigma, \#)\$x, w\$(\sigma, \#)u\$x) \mid \begin{array}{l} w \in L(A) \\ u \in (\Sigma \times \mathbb{D})^*, \sigma \in \Sigma \\ x \in (\Sigma \times \mathbb{D})^\omega \end{array} \right\}$$

It is well-known that no sequential machine can conduct the transformation $u(\sigma, \#) \mapsto (\sigma, \#)u$, as it implies guessing σ before reading it. Thus, there is a register transducer implementation that is good-enough for S if and only if $S = \emptyset$, if and only if $L(A) = \emptyset$. This concludes the proof. $\square$

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

Unbounded Synthesis for Register Automata In this part, we have given a picture of the decidability landscape of the synthesis of register transducers from register automata specifications over data domains $(\mathbb{D}, =, \mathbb{C})$, $(\mathbb{Q}, <, 0)$ and $(\mathbb{N}, <, 0)$, which respectively provide reasonably expressive formalisms for implementations and specifications. As demonstrated in Chapter 7, the landscape of unbounded synthesis is quite grim, as we quickly get undecidability: the (unbounded) synthesis problem is undecidable for both non-deterministic and universal specification register automata, already over $(\mathbb{D}, =, \#)$. This seems to be intrinsic to the case of infinite alphabets, as most known models have an undecidable universality or emptiness problem, which is often enough to calm synthesis passions. Still, results about decidability of the unbounded synthesis problem for deterministic specifications can be read as decidability of classes of games over data values, which is encouraging for further game-theoretic perspectives. Specifically, we plan to generalise the study of unbounded synthesis to good-for-games register automata. Multiple notions can be defined, depending on whether we want to impose constraints on the strategy that chooses transitions. Then, the main question is whether such a class is decidable, and how expressive it is.

Note also that the techniques that are developed for $(\mathbb{N}, <, 0)$ should be of interest to a more general audience. In particular, the proof of correctness based on strategy transfers through concretisation of tests along the run requires developments that are relevant in their own right. Besides, two problems remain open, namely unbounded synthesis for test-free non-deterministic specification automata, that we believe is decidable but difficult, and Church synthesis for deterministic one-sided specification automata over $(\mathbb{N}, <, 0)$ (i.e. when we target register transducer implementations with a fixed, constant initial valuation), for which we do not have a conjecture.

Register-Bounded Synthesis for Register Automata Moreover, register-bounded synthesis raises spirits, since decidability is recovered for universal register automata over $(\mathbb{D}, =, \mathbb{C})$, $(\mathbb{Q}, <, 0)$. The next part actually allows, as a side-product, to further generalise decidability of the register-bounded synthesis problem to the class oligomorphic data domains (under reasonable computability assumptions), that contain both data domains (see Chapter 14 (Thesis Conclusion)). Indeed, as we have seen, the key ingredient is that we are able to abstract the behaviour of register automata through the notion of type, which we store as constraints and yields a finite abstraction when there are finitely many. We did not include this development in this part for the sake of clarity. The case of register-bounded synthesis for URA over $(\mathbb{N}, <, 0)$ is left open in this manuscript as it is the topic of current research; it appears that a closer examination of constraint sequences allows to reduce to an ω -regular approximation of the problem, by extending the techniques that we developed for the unbounded case.

Towards Logics As illustrated by the case of test-free non-deterministic register automata, the transfer theorem yields a generic method for the decision of register-bounded synthesis, and paves the way for applications to other formalisms. Over data words, the landscape of relations between automata and logics is much more complex than in the finite alphabet case [101, 124]. In particular, the correspondence between ω -automata and MSO means that specifications given by ω -automata capture the intrinsic difficulty of the synthesis problem over finite alphabets; such a correspondence remains to be discovered for models over data words (if it exists at all). While undecidability results over logics for data words are somewhat deterring, they form a key line of research, as the ultimate goal of synthesis is to obtain high-level formalisms. In particular, two-variable fragments of first order logic look promising [128, 134, 135]. The synthesis problem from specifications expressed as first-order formulas with two variables and predicates (equality of data values), $<$ (order on positions) and $+1$ (successor on positions) is undecidable [34, Theorem 5], but the problem remains open when we restrict to $<$ and $+1$ (without the successor relation). It is also open in the more general setting of ordered data values (again without successor), i.e. with two predicates $<_p$ (order on positions) and $<_d$ (order on data), whose finite satisfiability problem has been shown decidable (more precisely, ExpSPACE-complete) in [135, Theorem 3.1].

Other Models of Computation It would also be relevant to consider other models, such as pebble automata [101, Section 2.2] and deterministic class-memory automata [103], which are each expressively incomparable with register automata (respectively [101, Figure 1] and [103, Proposition 4.2]). Note that the quest is doomed for non-deterministic class-memory automata (or, equivalently [103, Proposition 3.7], data automata [134, Section IV]), since they strictly contain non-deterministic register automata [103, Theorem 4.1], whose register-bounded and unbounded synthesis problem are undecidable (Theorems 6.19 and 6.20), unless possibly when targeting other classes of implementations.

Embracing Undecidability Another direction, initiated over infinite alphabets in [44] for register automaton formalism and furthered in [46] for Temporal Stream Logic, is to accept undecidability and look for semi-algorithms. Those two papers present encouraging results, which means that it is probably the way to go for practical applications in the near future. The techniques developed in [136, 137] could also yield semi-decision procedures for synthesis problems over (alternating) data automata, which is undecidable. Still, the field of synthesis over infinite alphabets is fairly recent, and many theoretical results are yet to be obtained so as to pinpoint the sources of the complexity and (un)decidability of the problems we consider.

Challenges for an Implementation As it reduces to the finite alphabet case, the method exposed in Chapter 6 allows to leverage existing tools dedicated to classical reactive synthesis. However, the main challenge is to tame the abstraction of the behaviour of register automata, more precisely their projection over labels: a naive construction systematically blows-up the state space by a factor that is exponential in the number

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’
 [124]: Benedikt, Ley, and Puppis (2010), ‘Automata vs. Logics on Data Words’

[128]: Dartois, Filiot, and Lhote (2018), ‘Logics for Word Transductions with Synthesis’
 [134]: Bojanczyk et al. (2006), ‘Two-Variable Logic on Words with Data’
 [135]: Schwentick and Zeume (2012), ‘Two-Variable Logic with Two Order Relations’
 [34]: Bérard et al. (2020), ‘Parameterized Synthesis for Fragments of First-Order Logic Over Data Words’

[135]: Schwentick and Zeume (2012), ‘Two-Variable Logic with Two Order Relations’

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

[103]: Björklund and Schwentick (2010), ‘On notions of regularity for data languages’

[134]: Bojanczyk et al. (2006), ‘Two-Variable Logic on Words with Data’

[44]: Ehlers, Seshia, and Kress-Gazit (2014), ‘Synthesis with Identifiers’

[46]: Finkbeiner et al. (2019), ‘Temporal Stream Logic: Synthesis Beyond the Booleans’

[136]: Iosif, Rogalewicz, and Vojnar (2016), ‘Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems’

[137]: Iosif and Xu (2018), ‘Abstraction Refinement for Emptiness Checking of Alternating Data Automata’

of registers, independently of the relevance of the constraints that are explored. We expect that antichain techniques [16] or binary decision diagram-based approaches [138] might be of use to control this blowup; this of course does not preclude the development of other techniques that are specific to registers.

Another possibility is to relax the requirement of feasibility for the action sequences, and to progressively restore it in case of a negative answer, using a CEGAR [139] approach. This is particularly relevant in the case of $(\mathbb{N}, <, 0)$, where feasibility checking requires some amount of counting.

To this challenge adds all the ones that already exist for classical synthesis. In particular, efforts have been made to yield compositional algorithms for synthesis [16, 19, 140], but there is still a lot to be done. And, contrary to verification, where one can independently check different properties, a specification should right away make precise all the requirements that the system has to meet, which means that they are typically bigger than specifications for verification, and makes compositionality even more important. Still, tremendous progress has happened in the last decade and, as pointed out in the previous paragraph, positive results already exist in the infinite alphabet case.

This concludes this part, and we now move to the study of the case of asynchronous specifications and implementations, again expressed by machines with registers.

[138]: Ehlers (2011), ‘Unbeast: Symbolic Bounded Synthesis’

[139]: Clarke et al. (2003), ‘Counterexample-guided abstraction refinement for symbolic model checking’

[16]: Filiot, Jin, and Raskin (2011), ‘Antichains and compositional algorithms for LTL synthesis’

[19]: Finkbeiner, Geier, and Passing (2021), ‘Specification Decomposition for Reactive Synthesis’

[140]: Lustig and Vardi (2013), ‘Synthesis from component libraries’

Part II.

**COMPUTABILITY OF FUNCTIONS
DEFINED BY TRANSDUCERS
OVER INFINITE ALPHABETS**

In this part, we study systems which do not react to their inputs in a synchronous way. More precisely, we are interested in non-deterministic asynchronous transducers with registers. As for their synchronous counterparts, these machines are equipped with a finite set of registers, that they use to store data values. We do not assume anymore that they are sequential, and instead study their non-deterministic variant. Finally, asynchrony means that on reading an input, the machine outputs a finite data word (which may be empty) instead of a single data value. Thus, while a synchronous transducer can be seen as an automaton operating over the product of the input and output alphabets, an asynchronous transducer corresponds to an automaton with two tapes that can move independently from left to right (except that the one operating over the input is not allowed to stay at the same spot for too long).

Already in the finite alphabet case, the reactive synthesis problem for specifications expressed by non-deterministic transducers is undecidable [48, Theorem 17], so we instead focus on the properties of the relations computed by these transducers. We are interested in particular in the subclass of transducers which compute functional relations. Note that, as we show, membership to this class is decidable for a large class of data domains which includes all the data domains that we study in Chapter 4 (Theorems 12.8, 12.40 and 12.64, respectively for $(\mathbb{D}, =)$, for oligomorphic data domains and for $(\mathbb{N}, <, 0)$).

We then examine the conditions that make a function in this class computable. As we are working over infinite words, the notion of computability has to be extended to ω -computability: a function is ω -computable when there exists a deterministic Turing machine that, on reading longer and longer prefixes of the input, produces longer and longer prefixes of the output (Definition 11.2 and Section 11.1). Letters of those words moreover take their values in an infinite alphabet, so we further need to extend the notion of a Turing machine. Those machines can store a data value in a single cell, and their transitions depend on tests over these data. This is of course an abstraction, as one cannot encode an infinite alphabet using a bounded amount of memory. We show that ω -computability is robust to encoding, in the sense that it also holds when we instead consider a machine that explicitly works with an encoding of the data. This is not the case for uniform computability, yet this notion still proves relevant as it allows a fine-grained analysis of the quantity of data values that have to be read before being able to produce some output.

In [51], Dave et al. established that over finite alphabets, ω -computability is equivalent to continuity for the Cantor distance for functions definable by (possibly two-way) non-deterministic transducers. Our main result is an extension of this equivalence over infinite words, for one-way register transducers. We moreover show that, even over data words, both notions are decidable for all the data domains that we consider (modulo mild computability assumptions). We draw the reader's attention to the fact that all those results hold when non-deterministic reassignment (also known as guessing) [38] is allowed, which strictly increases the expressive power of register transducers. This poses no difficulty in the case of $(\mathbb{D}, =, \mathbb{C})$ and oligomorphic domains, but necessitates additional work for the case of $(\mathbb{N}, <, 0)$. In particular, note that continuity was also studied in [50], where it is shown that deciding continuity of f reduces to deciding functionality of its topological closure. Such an approach

[48]: Carayol and Löding (2015), 'Uniformization in Automata Theory'

Oligomorphic data domains are data domains whose k -tuples are finite up to isomorphisms, for all $k \in \mathbb{N}$. They include $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$, but not $(\mathbb{N}, <, 0)$. See Section 12.2.

Some works use the terminology of 'computability' for ' ω -computability'. We prefer to use distinct terms to avoid confusion, as our study relies on establishing computability and decidability of some algorithmic problems in the traditional acceptance, i.e. the existence of some *terminating* program that solves the problem.

[51]: Dave et al. (2020), 'Synthesis of Computable Regular Functions of Infinite Words'

Continuity is to be understood in the usual mathematical sense (cf Definition 11.6).

In the Cantor topology, the distance between two words is inversely proportional to the length of their longest common prefix (Definition 11.5).

[38]: Kaminski and Zeitlin (2010), 'Finite-Memory Automata with Non-Deterministic Reassignment'

[50]: Prieur (2002), 'How to decide continuity of rational functions on infinite words'

again works for $(\mathbb{D}, =, \mathbb{C})$ and oligomorphic data domains, but fails over $(\mathbb{N}, <, 0)$ in the presence of non-deterministic reassignment. For this reason, we chose a different approach.

Related Publications Our results were partially published in conference proceedings [141] for the case of data domains the equality predicate only, without guessing. This is the topic of Section 12.1, which also allows guessing as it comes for free. This publication led to the writing of an extended version, that is currently being reviewed [142]. It generalises the study to oligomorphic data domains, that subsume $(\mathbb{D}, =, \mathbb{C})$ (Section 12.2), and to $(\mathbb{N}, <, 0)$ (Section 12.3), all with guessing.

Summary

[141]: Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘On Computability of Data Word Functions Defined by Transducers’. In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Lecture Notes in Computer Science. Springer, 2020

[142]: Léo Exibard, Emmanuel Filiot, Nathan Lhote, and Pierre-Alain Reynier. ‘Computability of Data-Word Transductions over Different Data Domains’. In: *Logical Methods in Computer Science (2021)*. Ed. by Jean Goubault-Larrecq and Barbara König

Non-Deterministic Asynchronous Register Transducers

10.

This part is largely independent of the other, so we recall here the notion of data domain, valuation and test.

Definition 4.1 (Data domain, data words) A *data domain* is a triple $\mathcal{D} = (\mathbb{D}, \mathfrak{R}, \mathfrak{C})$ where $\mathbb{D}$ is a countably infinite set of *data values*, $\mathfrak{R}$ is a finite set of interpreted *relational symbols*, also known as *predicates* and $\mathfrak{C}$ a finite set of interpreted *constant symbols*, disjoint from $\mathfrak{R}$. The set $\mathfrak{S} = \mathfrak{R} \sqcup \mathfrak{C}$ is called the *signature* of $\mathcal{D}$. We assume that the symbol $=$ is in $\mathfrak{R}$, and we interpret it as the equality relation.

A *data ω -word*, or simply *data word*, is an infinite sequence of data $x = d_0 d_1 \dots \in \mathbb{D}^\omega$. *Finite data words* are defined analogously.

Definition 4.7 (Valuation) A *valuation* of variables X over data domain $\mathcal{D}$ is a total function $\nu : X \rightarrow \mathbb{D}$. We denote by $\text{Val}_{\mathcal{D}}(X) = X^{\mathbb{D}}$ the set of valuations of X over $\mathcal{D}$.

Definition 4.10 (Test) Let R be a finite set of variables, that we call registers, and let $\star \notin R$ be a variable, denoting the input data value. A *test* in $\mathcal{D}$ over R is a quantifier-free formula ϕ over variables the $R \sqcup \{\star\}$.

The set of tests in $\mathcal{D}$ over R is denoted $\text{Tests}_{\mathcal{D}}(R)$. It is infinite in general, but finite up to logical equivalence (see Proposition 4.5).

For a valuation $\nu : R \rightarrow \mathbb{D}$, a data value $d \in \mathbb{D}$ and a test ϕ , we often write $\nu, d \models \phi$ for $\nu\{\star \leftarrow d\} \models \phi$.

Remark 10.1 (On Labels) Contrary to Part I, in this part we are only concerned with data words which have no labels, as they can be included in the data domain. Formally, for a finite alphabet Σ and a data domain $\mathcal{D} = (\mathbb{D}, \mathfrak{R}, \mathfrak{C})$, one can instead consider the product data domain $\Sigma \times \mathcal{D} = (\Sigma \times \mathbb{D}, \mathfrak{R} \sqcup \{=\sigma\}_{\sigma \in \Sigma}, (\Sigma \times \mathfrak{C}))$, where relations in $\mathfrak{R}$ are interpreted over the data component and, for $\sigma \in \Sigma$, $=_\sigma(a, d)$ whenever $a = \sigma$. This choice is made to reduce notational complexity. Besides, the reader can check that labels would anyway play no role in our proofs. Note that they are present in some examples; this should pose no difficulty.

10.1. The Model

We are now ready to define non-deterministic asynchronous register transducers. They are defined by equipping non-deterministic (register-free) asynchronous transducers (see, e.g., [71, 143, Chapter IV Section 1.5]) with registers, in the same way as synchronous sequential register transducers extend their register-free counterparts. Recall that instead of outputting a single letter, an asynchronous transducer is allowed to output a (possibly empty) word. Note that here, non-determinism is to be understood as the full-fledged notion, i.e. we allow non-deterministic reassignment (or guessing) of data values. Finally, for simplicity, we only study the Büchi acceptance condition. This is without loss of generality as it can express the other ω -regular acceptance conditions (Fact 3.1).

This part is meant to be readable independently. However, the reader who skipped Chapter 4 is invited to consult the beginning of this chapter if they are not familiar with the notions of tests and valuations (Section 4.2), as it provides some examples of their use. More generally, this part assumes an elementary knowledge of register automata (Definition 4.13), as they underly our definition of transducers and are used as a tool in the proofs. This knowledge can be found in Chapter 4, which states properties from the literature that we use here. Finally, recall that the `knowledge` package links all notions to the place they are defined.

Data domains without equality constitute pathological cases that we want to rule out.

For simplicity, in the following, we confuse the symbols with their interpretation.

Note that data words are infinite, otherwise they are called finite data words.

$\mathcal{D}$ is omitted when clear from the context.

As previously stated, we explicitly treated labels in the first part for modelling purposes. Here, we drop this treatment to lighten the argument, since we lie on a more theoretical level.

[71]: Sakarovitch (2009), *Elements of Automata Theory*

[143]: Sakarovitch (2003), *Éléments de théorie des automates*

To express non-deterministic reassignment in an operational way, we slightly modified the syntax of transitions of the model from that of sequential transducers: there is now no dedicated `asgn` set, and the transition instead expresses whether the value of the register is kept, set to the input data value or guessed.

Assumption 10.1 In the following, we assume that the set of $\mathbb{C}$ is not empty and contains a distinguished constant $\#$. As for register automata, this is needed for the initialisation of registers. Given a set R of registers, the *initial valuation* is defined as $v_R^t : r \in R \mapsto \#$.

Definition 10.2 (Non-deterministic asynchronous register transducer) A *non-deterministic asynchronous register transducer* (with Büchi acceptance condition) over data domain $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathbb{C})$ is a tuple $T = (Q, I, R, \Delta, F)$ where:

- Q is a finite set of *states*
- $I \subseteq Q$ is a set of *initial states*
- $F \subseteq Q$ is a set of *accepting states*
- R is a finite set of *registers*
- Δ is the *transition relation*. It is a finite subset of

$$\underbrace{Q}_{\text{current state}} \times \underbrace{\text{Tests}_{\mathcal{D}}(R)}_{\text{current registers + input data}} \times \underbrace{\{\text{keep, set, guess}\}^R}_{\text{register operations}} \times \underbrace{R^*}_{\text{output word}} \times \underbrace{Q}_{\text{target state}}$$

As usual, we write $p \xrightarrow[T]{\phi | \text{regOp}, v} q$ for $t = (p, \phi, \text{regOp}, v, q) \in \Delta$, and omit T and/or t when they are clear from the context.

Such a transducer is *non-guessing* if for all transitions, $\text{regOp} : R \rightarrow \{\text{keep, set}\}$. It is *deterministic* when, additionally, its transition relation is a partial function $\delta : Q \times \text{Tests}_{\mathcal{D}}(R) \rightarrow Q \times \{\text{keep, set}\}^R \times R^*$, which is moreover such that for all states $q \in Q$, for all valuations $v : R \rightarrow \mathbb{D}$ and all data values $d \in \mathbb{D}$, there exists at most one $\phi \in \text{Tests}_{\mathcal{D}}(R)$ such that $(q, \phi) \in \text{dom}(\delta)$ and $v, d \models \phi$.

In the following, we simply say ‘non-deterministic register transducer’ or even ‘register transducer’ for ‘non-deterministic asynchronous register transducer’, as no ambiguity arises.

Now, a *configuration* C of T consists of a state and a valuation, i.e. $C = (q, v) \in Q \times \text{Val}_{\mathcal{D}}(R)$. A configuration is *initial* if $q \in I$ and $v = v_R^t$. It is *final* if $q \in F$. The set of configurations of T is written $\text{Configs}(T)$.

Let $C = (p, v)$ and $C' = (q, \lambda)$ be two configurations; let $d \in \mathbb{D}$ and $t = p \xrightarrow[T]{\phi, \text{regOp}, v} q$ be a transition. We say that C' is a *successor configuration* of C by reading d through t and *producing* $w \in \mathbb{D}^{|v|}$ when:

- The test is satisfied, i.e. $v, d \models \phi$
- Registers are *updated* correctly, i.e.
 - for all $r \in R$, if $\text{regOp}(r) = \text{keep}$, then $\lambda(r) = v(r)$
 - for all $r \in R$, if $\text{regOp}(r) = \text{set}$, then $\lambda(r) = d$
 - for all $r \in R$, if $\text{regOp}(r) = \text{guess}$, then $\lambda(r) \in \mathbb{D}$ can take any data value in $\mathbb{D}$
- The output data word indeed corresponds to the content of the output registers: for all $0 \leq i < |v|$, $w[i] = \lambda(v[i])$.

Note that, as for synchronous transducers, w is output *after* the possible assignment in t .

We then write $C \xrightarrow[T]{d, \phi | \text{regOp}, w} C'$ when this is the case. In that case, we say that t is *enabled* from C . If they are clear from the context, we omit T and/or t ; we sometimes omit ϕ and regOp and simply write $C \xrightarrow{d|w}_t C'$.

The fact that t is enabled only depends on C and ϕ , as it amounts to asking whether there exists $d \in \mathbb{D}$ such that $C, d \models \phi$.

A *run* over input $x \in \mathbb{D}^\omega$ is an infinite sequence $\rho = C_0 t_0 C_1 t_1 \dots \in (\text{Configs}(T)\Delta)^\omega$ such that for all $i \in \mathbb{N}$, $C_i \xrightarrow[t_i]{x[i]v_i} C_{i+1}$. It *produces* the (possibly finite) output $w = v_0 v_1 \dots \in \mathbb{D}^\infty$. It is *initial* if C_0 is initial. It is *final* if $\text{states}(\rho)$ satisfy the Büchi acceptance condition defined by F , i.e. infinitely many configurations of ρ are final. It is *accepting* if it is both initial and final. We write $p \xrightarrow{x|w}$ when there is a run from state p that produces $w \in \mathbb{D}^\infty$ on input $x \in \mathbb{D}^\omega$. The notion of (initial, final, accepting) partial run are defined in the expected way; we write $p \xrightarrow{u|v} q$ when there is a partial run from p to q with input $u \in \mathbb{D}^*$ and output $v \in \mathbb{D}^*$.

Formally, ρ is *accepting* whenever $\text{Inf}(\text{states}(\rho)) \cap F \neq \emptyset$.

Then, the semantics of T is defined as $\llbracket T \rrbracket \subseteq \mathbb{D}^\omega \times \mathbb{D}^\infty$, with

$$\llbracket T \rrbracket = \{(x, w) \mid \text{there exists an accepting run over } x \text{ that produces } w\}$$

Size of a register transducer Complexity-wise, the relevant parameters for T are its number of states $|Q|$, its number of registers k and its *maximum output length* M , which is the maximum length of any output word v appearing on the transitions. The size of T is then defined as their product. Note that k is implicitly encoded in unary, since in general it requires $\mathcal{O}(k)$ bits to describe a transducer with k registers.

Graphical depiction of register transducers In the examples, we often provide a pictorial representation of the transducers we consider. As for register automata, the graphical depiction of transducers slightly differs from its mathematical definition, to get a more intuitive rendering. States and transitions are respectively depicted as circles and arrows, as usual. Given a transition $p \xrightarrow{\phi|\text{regOp},w} q$, we extract from regOp the set $\text{asgn} = \{r \in R \mid \text{regOp}(r) = \text{set}\}$ of assigned registers and $\text{ndasgn} = \{r \in R \mid \text{regOp}(r) = \text{guess}\}$. Registers whose content is unaffected, i.e. those such that $\text{regOp}(r) = \text{keep}$, are not depicted so as not to clutter the figure. Then, as for register automata, we write $\downarrow r$ to signify that $r \in \text{asgn}$ (the input data value is stored in r) and $?r$ when $r \in \text{ndasgn}$ (the content of r is guessed).

These graphical conventions are recapitulated in Section A.6.

Finally, when labels from a finite alphabet Σ are involved (Remark 10.1), we unify the notations with Part I and write σ, ϕ for the test $\sigma(\star) \wedge \phi$, which conducts test ϕ and additionally asks that the label is σ , for all $\sigma \in \Sigma$.

Example 10.1 As an example, consider the register transducer T_{rename} depicted in Figure 10.1 on the following page. It operates over data domain $(\Sigma \times \mathbb{D}, =, \text{del}, \text{ch}, \$)$, where $\Sigma = \{\text{del}, \text{ch}, \$\}$ is an alphabet of labels (see Remark 10.1). By a slight abuse of notation, del (respectively, ch , $\$$) are unary predicates specifying that the label is del (resp. ch , $\$$). It deals with communication logs between a set of clients. A log is an infinite sequence of pairs consisting of a tag, chosen in some finite alphabet Σ , and the identifier of the client delivering this tag, chosen in some infinite set of data values. The transducer must modify the log as follows: for a given client that needs to be modified, each of its messages should now be associated with some new identifier. It should verify that this new

identifier is indeed free, i.e. never used in the log. Before treating the log, the transformation receives as input the id of the client that needs to be modified (associated with the tag *del*), and then a sequence of client identifiers (associated with the tag *ch*), ending with *\$*. The transducer is non-deterministic as it does not impose the choice of the identifier that is used to replace the one of the client. In particular, observe that it may associate multiple output words to a same input if two such free identifiers exist. Labels on the output are irrelevant and unconstrained.

This specification can be described as the following relation: its inputs are of the form $(\text{del}, \text{id}_0)(\text{ch}, \text{id}_1) \dots (\text{ch}, \text{id}_k)(\$, \text{id}_{k+1})(\sigma_0, \text{id}'_0)(\sigma_1, \text{id}'_1) \dots$, where $\text{id}_0, \dots, \text{id}_{k+1} \in \mathbb{D}$ and $(\sigma_i, \text{id}'_i) \in \Sigma \times \mathbb{D}$ for all $i \in \mathbb{N}$. The corresponding admissible outputs are of the form $\text{id}''_0 \text{id}''_1 \dots$ (labels are omitted), where there exists a replacement identifier with index $j \in \{1, \dots, k\}$ which is distinct from the identifier to replace, i.e. $\text{id}_j \neq \text{id}_0$ and distinct from all subsequent identifiers that are not replaced, i.e. for all $i \geq 0$, if $\text{id}'_i \neq \text{id}_0$ then $\text{id}_j \neq \text{id}'_i$. Then we ask that the replacement is conducted correctly, i.e. if $\text{id}'_i = \text{id}_0$, then $\text{id}''_i = \text{id}_j$.

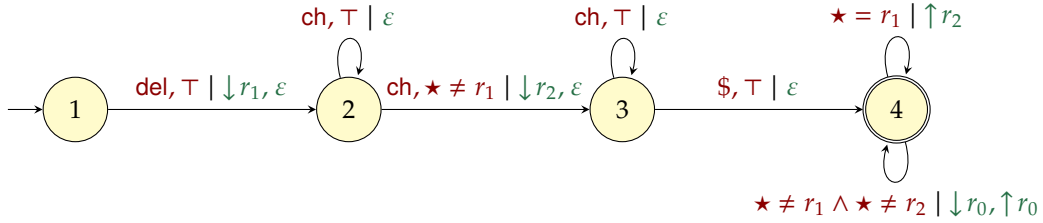


Figure 10.1. A register transducer T_{rename} . It has three registers r_1 , r_2 and r_0 and four states. Register r_1 stores the id of *del* and r_2 the chosen id of *ch*, while r_0 is used to output the last data value read as input. $\downarrow r$ means that the input data value is stored in r , and $\uparrow r$ that the content of r is output. Predicates σ in tests are depicted as σ, ϕ instead of $\sigma(\star) \wedge \phi$ to unify with Part I.

Now, consider the following variant: instead of getting the replacement identifier from its input, the transducer initially guesses some new identifier, and then operates the replacement as above, while checking that the identifier it guessed indeed does not appear in the input. This guessing mechanism can be modelled using non-deterministic reassignment (cf Figure 10.2).

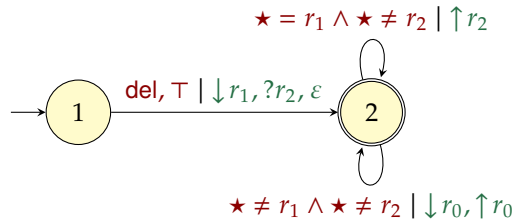


Figure 10.2. A variant T_{rename2} of the register transducer T_{rename} , that uses non-deterministic reassignment to replace a given client id with a new identifier. Note that the transducer needs to check that its guess is distinct from the modified identifier when it reads it (hence the requirement that $\star \neq r_2$ when $\star = r_1$) since, in our model, properties of the guessed data values are checked a posteriori (this is done to lighten the syntax). The notation $?r_2$ means that r_2 is non-deterministically assigned some data value.

Assumption 10.3 In the following, we are only interested in transducers that produce infinite outputs, i.e. $\llbracket T \rrbracket \subseteq \mathbb{D}^\omega \times \mathbb{D}^\omega$. Restricting the accepting runs of a transducer to the ones that produce data ω -words is a Büchi

condition and can easily be done by adding one bit of information to states.

10.2. Register Transducers and Register Automata

As expected, the model of register transducer is tightly linked with that of register automata. In this section, we establish a series of (unsurprising but useful) technical results that allow to reduce problems over register transducers to the simpler model of register automaton. For our purpose, a register automaton is simply a register transducer with no outputs.

Definition 10.4 (Register Automaton) A *non-deterministic register automaton* (with guessing) is a tuple $A = (Q, I, R, \Delta, F)$, where Q is a finite set of *states*, $I \subseteq Q$ a set of *initial states*, $F \subseteq Q$ of *accepting states*, and $\Delta \subseteq Q \times \text{Tests}_{\mathcal{D}}(R) \times \text{RegOp}_R \times Q$ is a *transition relation*.

See also Definition 4.13 in Chapter 4.

Non-guessing register automata are defined as expected.

We do not formalise the semantics of the model, as it is naturally inherited from that of register transducers. The reader can check that it corresponds to the model of register automaton introduced in Chapter 4 *with* non-deterministic reassignment (Definition 4.35), with a Büchi acceptance condition, modulo a minor adaptation of the syntax. In this part, register operations are grouped in a single object regOp since, unlike in Part I, guessing is natively supported by the model.

To a transition $p \xrightarrow{\phi, \text{regOp}} q$ of the model we define here, one can associate the transition $p \xrightarrow{\phi, \text{asgn}, \text{ndasgn}} q$, where $\text{asgn} = \{r \in R \mid \text{regOp}(r) = \text{set}\}$ and $\text{ndasgn} = \{r \in R \mid \text{regOp}(r) = \text{guess}\}$ of the model of Chapter 4, and conversely.

10.2.1. Traces of Runs of Register Transducers

Assumption 10.5 (Separator) In the following, we assume the existence of a distinguished constant $\$$ that we use as a *separator*. We implicitly assume that it is not present in the input, except at the places where we specify it. Given a data domain $\mathcal{D}$, one can simply consider $\mathcal{D} \sqcup \{\$\}$, where the equality relation is extended accordingly. The structure of $\mathcal{D}$ is only mildly affected by this addition: the automorphism group of $\mathcal{D} \sqcup \{\$\}$ is isomorphic with that of $\mathcal{D}$ (see Proposition 12.14).

Register transducers can be simulated by register automata in the following sense: the runs of a register transducer can be flattened to their traces, so as to be recognised by a non-deterministic register automaton (Proposition 10.1).

Definition 10.6 Let T be a non-deterministic register transducer. Given a run $\rho = (q_0, v_0) \xrightarrow{d_0 | y_0} (q_1, v_1) \dots$, we define its (marked) *trace* $\text{tr}(\rho) = d_0 \cdot y_0 \$ d_1 \cdot y_1 \$ \dots$.

Given a register transducer, it is straightforward to construct a non-deterministic register automaton over data domain $\mathcal{D} \sqcup \{\$\}$ which recognises the language of marked traces of accepting runs of T :

Proposition 10.1 Let T be a register transducer. The data language

$$\text{traces}(T) = \{\text{tr}(\rho) \mid \rho \text{ is an accepting run of } T\}$$

is recognised by a non-deterministic register automaton A_T^r with a polynomial number of states and the same set of registers. Moreover, if T is non-guessing, then so is A_T^r .

Besides, for all configurations $(p, v), (q, \lambda) \in \text{Config}(T)$, all data values $d_0, \dots, d_n$ and for all finite data words $y_0, \dots, y_n$, there is a partial run $(p, v) \xrightarrow{d_0 \cdot y_0 \$ d_1 \cdot y_1 \$ \dots \$ d_n \cdot y_n \$} (q, \lambda)$ in A_T^r if and only if there is a partial run $(p, v) \xrightarrow{d_0 \dots d_n | y_0 \dots y_n} (q, \lambda)$ in T .

Proof. We simply show how to translate a single transition of T , the general construction follows. The automaton stores the name t of the transition in its states, and conducts the test using a first transition. Then, it simulates the output, by checking that it successively reads the data values contained in the registers of the output word. To that end, it uses a counter bounded by the length of the output word of the transition. It finally reads \$.

Formally, A_T^r has states $Q \sqcup (Q \times \Delta \times \{0, \dots, M+1\})$, where M is the maximum output length of T . Consider $t = p \xrightarrow[\text{regOp}, v]{\phi} q$, where $v = r_0 \dots r_{n-1} \in R^n$ for $n = |v|$. Construct the transitions $p \xrightarrow{\phi, \text{regOp}} (p, t, 0) \xrightarrow{\star=r_0} (p, t, 1) \dots \xrightarrow{\star=r_{n-2}} (p, t, n-1) \xrightarrow{\star=r_{n-1}} (p, t, n) \xrightarrow{\star=\$} q$. Correctness of the construction follows from the semantics of both models. $\square$

Those traces can then be interleaved, in order to be compared.

Definition 10.7 Given two runs ρ_1 and ρ_2 with respective traces $\text{tr}(\rho_1) = d_0 \cdot y_0 \$ d_1 \cdot y_1 \dots$ and $\text{tr}(\rho_2) = e_0 \cdot z_0 \$ e_1 \cdot z_1 \dots$ in $(\mathbb{D}\mathbb{D}^*\$)^\omega$, we define the interleaving of their traces as $\rho_1 \otimes \rho_2 = d_0 \cdot y_0 \$ e_0 \cdot z_0 \$ d_1 \cdot y_1 \$ e_1 \cdot z_1 \dots \in ((\mathbb{D}\mathbb{D}^*\$)^2)^\omega$.

Given a register transducer T , we can define the language of its interleaved traces

$$L_\otimes(T) = \{\rho_1 \otimes \rho_2 \mid \rho_1 \text{ and } \rho_2 \text{ are accepting runs of } T\} \quad (10.1)$$

One can then compute the product of A_T^r with itself, as is done in [144], and that uses the separators \$ as synchronisers to get:

Proposition 10.2 Let T be register transducer with k registers. The data language $L_\otimes(T)$ is recognised by a non-deterministic register automaton $A_\otimes^T$ with $2k$ registers, whose size is polynomial in $|T|$.

Besides, for all configurations $(p, v), (p', v'), (q, \lambda), (q', \lambda') \in \text{Config}(T)$, all data values $d_0, \dots, d_n, e_0, \dots, e_n$ and all finite data words $y_0, \dots, y_n$ and $z_0, \dots, z_n$, there is a partial run

$$((p, p'), (v, v')) \xrightarrow{d_0 \cdot y_0 \$ e_0 \cdot z_0 \$ \dots \$ d_n \cdot y_n \$ e_n \cdot z_n} ((q, q'), (\lambda, \lambda'))$$

in $A_\otimes^T$ if and only if there are two partial runs

$$(p, v) \xrightarrow[\text{regOp}, v]{\phi} (q, \lambda) \quad (p', v') \xrightarrow[\text{regOp}, v']{\phi} (q', \lambda')$$

[144]: Béal et al. (2003), ‘Squaring transducers: an efficient procedure for deciding functionality and sequentiality’

The second technical property of the statement is needed to make a connection between the structure of T and that of A_T^r . This is particularly useful in pumping arguments.

in T such that (p, v) and (p', v') are jointly reachable in T , i.e. there exists a finite input data word $u \in \mathbb{D}^*$ such that $C^i \xrightarrow[T]{u|u_1} (p, v)$ and $C^i \xrightarrow[T]{u|u_2} (p', v')$.

With the same notations, there is a run

$$((p, p'), (v, v')) \xrightarrow{d_0 \cdot y_0 \$ e_0 \cdot z_0 \$ d_1 \cdot y_1 \$ e_1 \cdot z_1 \dots} ((q, q'), (\lambda, \lambda'))$$

in $A_{\otimes}^T$ if and only if there are two runs

$$(p, v) \xrightarrow[T]{d_0 d_1 \dots | y_0 \cdot y_1 \dots} \quad (p', v') \xrightarrow[T]{e_0 e_1 \dots | z_0 \cdot z_1 \dots}$$

10.2.2. Properties Inherited from Register Automata

The links between register automata and register transducers allow to transfer some results from one model to the other. First, a straightforward extension of the proof of Proposition 4.1 establishes that over a finite set of data values, non-deterministic register transducers that are *non-guessing* behave like non-deterministic ω -transducers:

Proposition 10.3 *Let T be a non-guessing non-deterministic register transducer over data domain $\mathcal{D}$. For any finite subset $X \subseteq \mathbb{D}$, $\llbracket T \rrbracket \cap (X^\omega \times X^\omega)$ is recognised by a non-deterministic ω -transducer.*

This non-deterministic ω -transducer has $n(s+1)^k$ states, where n is the number of states of T , k its number of registers, and $s = |X|$ is the size of X .

Remark 10.2 Note that this is not the case in general when guessing (a.k.a. non-deterministic reassignment) is allowed, already for register automata. For instance, over $(\mathbb{N}, +1, 0)$, one can define a non-deterministic register automaton which ignores its input and simulates a run of a Minsky machine using two registers, each containing the current value of one counter. Those registers are initialised with 0. At each step, the automaton guesses the values of the counters and checks that increments, decrements and zero-tests are carried out correctly. Due to the syntax of the model, those tests must be conducted on the next input step, but this poses no difficulty. Finally, it reaches an accepting state whenever the machine that it simulates reaches an halting state. Overall, this automaton accepts some input if and only if it accepts any input, if and only if the Minsky machine it simulates does not halt. This means that the restriction to a finite number of data values cannot be *effectively* ω -regular. One can then modify the construction so that the automaton recognises for instance $L_{\text{count}} = \{d_0^n \cdot d_1^n \cdot \$^\omega \mid d_0, d_1 \in \mathbb{N}, d_0 \neq d_1\}$, which is such that $L_{\text{count}} \cap \{0, 1\}^\omega = \{0^n \cdot 1^n \cdot \$^\omega\}$, which is notoriously not ω -regular. Note that this construction actually allows to recognise any recursively enumerable language (and even non-computable ones by allowing non-computable predicates).

On the contrary, this property is recovered over $(\mathbb{D}, =, \mathbb{C})$ [38, Proposition 9]; see also Proposition 12.2.

Besides, one can adapt the proof of Proposition 4.2 to show that, as for non-deterministic register automata, NRT are closed under union.

[38]: Kaminski and Zeitlin (2010), 'Finite-Memory Automata with Non-Deterministic Reassignment'

Proposition 10.4 Let T_1, T_2 be two non-deterministic register transducers over the same data domain $\mathcal{D}$, with respectively n_1 and n_2 states and k_1 and k_2 registers, and with final states F_1 and F_2 .

Then, $\llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket$ is recognised by a non-deterministic register transducer with $n_1 + n_2$ states and $\max(k_1, k_2)$ registers, and with final states $F_1 \cup F_2$.

As their finite alphabet counterparts, they are however not closed under intersection.

Property 10.5 Relations recognised by NRT over $(\mathbb{D}, =, \mathfrak{C})$ are not closed under intersection.

Proof. This is a consequence of the fact that NRT behave as transducers when restricted over a finite alphabet. It is well-known that the latter are not closed under intersection, see e.g. [71, Remark 1.1 in Chapter IV].

More precisely, build two NRT that respectively recognise $\{(a^n, b^n c^* \$^\omega) \mid n \in \mathbb{N}\}$ and $\{(a^n, b^* c^n \$^\omega) \mid n \in \mathbb{N}\}$ for some fixed constants $a, b, c \in \mathfrak{C}$. Then, their intersection is $\{(a^n, b^n c^n \$^\omega) \mid n \in \mathbb{N}\}$. A simple pumping argument establishes that such a relation cannot be recognised by any NRT (since it cannot be recognised by any transducer over alphabet $\{a, b, c\}$).

Note that the proof extends to the case where $|\mathfrak{C}| < 3$ by making the transducer first ask for three distinct data values that play the role of a, b and c . The proof also extends to $(\mathbb{Q}, <, 0)$ and $(\mathbb{N}, <, 0)$. $\square$

[71]: Sakarovitch (2009), *Elements of Automata Theory*

10.3. Functions Recognised by Asynchronous Transducers

Asynchronous register transducers define in general relations and not necessarily functions since they are non-deterministic, as illustrated by Example 10.1. As our goal is to study notions of ω -computability and continuity, which apply to functions, we are mainly interested in functional register transducers.

Definition 10.8 (Functional Transducer) A register transducer is *functional* if $\llbracket T \rrbracket$ is a partial function.

Example 10.2 Consider again the transducer T_{rename} of Example 10.1. In general, it defines a relation. One can reinforce the specification by fixing some k and asking that the transducer reads k identifiers, and chooses the first that is free. This transformation is realised by the register transducer $T_{\text{rename}3}$ depicted in Figure 10.3 on the following page (for $k = 2$).

Note that using non-determinism only, one cannot specify this behaviour without fixing k . This can be modelled through universality (a.k.a. *co-non-determinism*), but this yields a model for which the problems that we study are undecidable (including functionality).

Consider a slightly different setting, in which the identifiers are ordered. We want to replace the identifier with the maximal identifier appearing in the input. This can be modelled, again using non-deterministic reassignment, as illustrated in Figure 10.4 on the next page.

We implicitly require that the identifier to replace is not the maximal one.

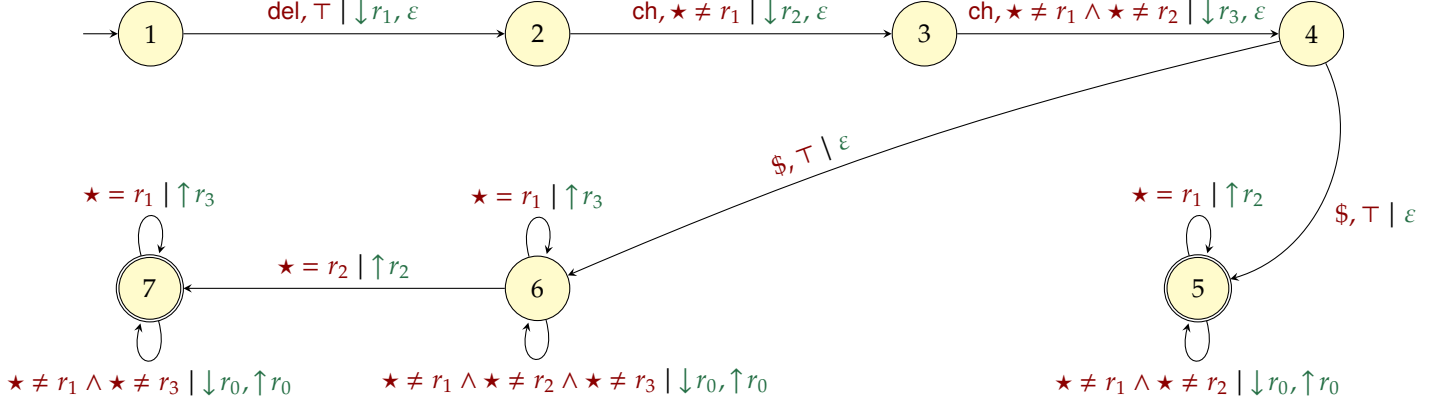


Figure 10.3. A non-deterministic register transducer T_{rename3} , with four registers r_1, r_2, r_3 and r_0 (the latter being used, as in Figure 10.1, to output the last read data value). After reading the $\$$ symbol, it guesses whether the value of register r_2 appears in the suffix of the input word. If not, it goes to state 5, and replaces occurrences of r_1 by r_2 . Otherwise, it moves to state 6, waiting for an occurrence of r_2 , and replaces occurrences of r_1 by r_3 .

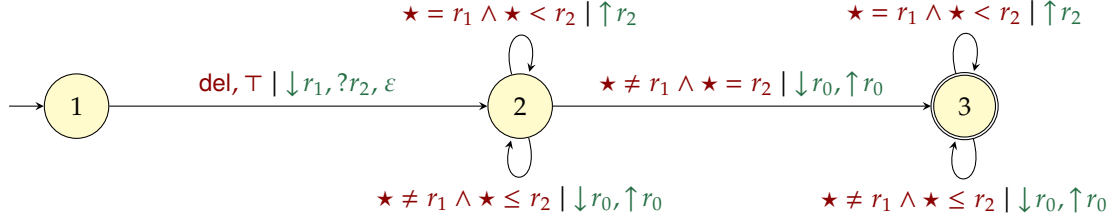


Figure 10.4. A variant T_{rename4} of the register transducer T_{rename3} , that uses non-deterministic reassignment to guess the maximal identifier, for some given order over the identifiers. The transducer checks that its guess is always greater than or equal to the client identifiers, and that it appears at least one in the input (transition from state 2 to state 3).

The corresponding decision problem is the functionality problem. We show that it is decidable and more precisely PSPACE-complete for the data domains we consider. See Theorems 12.8, 12.40 and 12.64, respectively for $(\mathbb{D}, =, \mathbb{C})$, oligomorphic data domains and for $(\mathbb{N}, <, 0)$.

Problem 10.1: FUNCTIONALITY PROBLEM

Input: A register transducer T
Output: Yes if $\llbracket T \rrbracket$ is a partial function
 No otherwise

Note that when the functionality problem is decidable, it allows to decide equivalence of functional non-deterministic register transducers on the intersection of their domain.

Proposition 10.6 *If the functionality problem is decidable, then one can decide, given two functions f and g defined by non-deterministic register transducers, whether for all $x \in \text{dom}(f) \cap \text{dom}(g)$, $f(x) = g(x)$.*

Moreover, the reduction is polynomial.

Proof. Let f and g be two functions, respectively defined by the transducers T_f and T_g . The relation $f \cup g = \{(x, y) \mid y = f(x) \text{ or } y = g(x)\}$ is recognised by the union of T_f and T_g , which can be constructed in polynomial (even linear) time (cf Proposition 10.4). Then, $f \cup g$ is functional if and only if for all $x \in \text{dom}(f) \cap \text{dom}(g)$, $f(x) = g(x)$. $\square$

In particular, this yields decidability of the equivalence of NRT with total domain. Note however that it is undecidable whether two NRT have the

Recall that the language-equivalence problem for non-deterministic register automata is undecidable, since the universal language $\mathbb{D}^\omega$ is trivially NRA-definable and universality is undecidable, by [101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’ (Theorem 5.3, see also our Theorem 4.21).

same domain, or even whether a given NRT has total domain, already in the synchronous case, as a direct consequence of Theorem 8.1.

10.4. Closure under Composition

A desirable property of a class of functions is that it is closed under composition, as it allows to define complex functions from simpler elements. We thus start this study by showing that this class is closed under composition. We actually establish a more general result, as this property even holds for relations.

Theorem 10.7 ([141, Theorem 12]) *Let f and g be two functions defined by non-deterministic register transducers over the same data domain $\mathcal{D}$, that admits quantifier elimination. Then, their composition $f \circ g$ is effectively definable by a non-deterministic register transducer.*

[141]: Exibard, Filiot, and Reynier (2020), ‘On Computability of Data Word Functions Defined by Transducers’

$$f \circ g : x \mapsto f(g(x))$$

Proof (main ideas). The proof is similar to the data-free case, where the composition is computed via a product construction which simulates both transducers in parallel, executing the second on the output of the first. Quantifier elimination allows to express the existence of a path in the transducer computing f that executes over the output of the one computing g using first-order logic formulas. The construction is quite technical, so we only describe the main ideas behind it.

Let $\mathcal{D}$ be a data domain that admits quantifier elimination. Then, we can write tests as first-order formulas, and then convert them to quantifier-free ones to get a proper register transducer. Let f and g be two functions from $\mathbb{D}^\omega$ to $\mathbb{D}^\omega$, respectively computed by transducers $T_f = (Q_f, I_f, R_f, \Delta_f, F_f)$ and $T_g = (Q_g, I_g, R_g, \Delta_g, F_g)$. Assume without loss of generality that R_f and R_g are distinct. Then, we define $T_{f \circ g} = (Q_f \times Q_g, I_f \times I_g, R_f \sqcup R_g, \Delta' F_f \times F_g)$. We now define the transitions. We want, for all configurations $(p, v), (p', v')$ of T_f and $(q, \lambda), (q', \lambda')$ of T_g , for all $d \in \mathbb{D}$, all $w \in \mathbb{D}^*$, all $\phi \in \text{Tests}(R)$, $\text{regOp} \in \text{RegOp}_R$, that $((p, v), (q, \lambda)) \xrightarrow[T_{f \circ g}]{d, \phi | \text{regOp}, w} ((p', v'), (q', \lambda'))$ whenever:

$$(i) (q, \lambda) \xrightarrow[T_g]{d, \phi | \text{regOp}', w'} (q', \lambda')$$

(ii) By writing $w' = e_1 \dots e_n$, we have:

$$(p, v) \xrightarrow[T_f]{e_1, \phi_1 | \text{regOp}_1, w_1} (p_1, v_1) \dots (p_{n-1}, v_{n-1}) \xrightarrow[T_f]{e_n, \phi_n | \text{regOp}_n, w_n} (p', v')$$

where $w_1 \dots w_n = w$

Both conditions can be expressed in first-order logic, so this can be written as a (big) test; recall that the length of output words on each transition is bounded by a constant M that only depends on the transducer, so the length n of the path of (ii) is bounded. One just need reassignments to handle register operations correctly, since the path in T_f has to be simulated in one step, but we know that reassignments can be removed without loss of expressive power (Proposition 4.6).

These observations allow to define an NRT that computes $f \circ g$. We leave out the details. $\square$

Remark 10.3 In the proof, we do not use the hypothesis that T_f and T_g are functional. This actually shows a stronger result, namely that relations defined by NRT are closed under composition, where

$$R \circ S = \{(x, y) \in \mathbb{D}^\omega \times \mathbb{D}^\omega \mid \exists z \in \mathbb{D}^\omega, (x, z) \in S \text{ and } (z, y) \in R\}$$

As a consequence, we get the following:

Corollary 10.8 *Functions computed by register transducers over $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, 0)$ are closed under composition.*

Proof. This results from the fact that both domains admit quantifier elimination [115, Section 2.3]. $\square$

[115]: Marcja and Toffalori (2003), ‘Quantifier Elimination’

Since $(\mathbb{N}, <, 0)$ is a substructure of $(\mathbb{Q}, <, 0)$, we also get:

Corollary 10.9 *Functions computed by register transducers over $(\mathbb{N}, <, 0)$ are closed under composition.*

ω -Computability and Continuity

11.1. Computability

Given a function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ recognised by some register transducer, one might want to check whether it can be computed by some program. We thus introduce a notion of ω -computability for functions over infinite words. As we are working on data words, we right away consider it over infinite alphabets, but this is a direct generalisation of the notion of computability defined in [51, Definition 3] (see also [145, Chapter 2]). One might wonder why a relatively elementary model as transducers do not always induce ω -computable functions. The answer is that over ω -words (and, a fortiori, data ω -words), non-determinism is very expressive. For instance, the machine can guess whether some letter (or data value) appears infinitely often in the input, which cannot be decided by reading any finite prefix of the input (cf f_{again} in Example 11.2).

11.1.1. Representation

To define programs over data ω -words, we need the data values to be (finitely) representable. This is the case for $(\mathbb{N}, =)$ through binary encoding, as well as for $(\mathbb{N}, <)$ and $(\mathbb{Q}, <)$, but not for $(\mathbb{R}, =)$ (see Example 11.1).

Formally, let $\mathcal{D} = (\mathbb{D}, \mathfrak{R}, \mathbb{C})$ be a data domain. An *encoding* of $\mathcal{D}$ is an injective function $\text{enc} : \mathbb{D} \rightarrow A^*$ for some finite alphabet A such that membership to following sets is decidable: $\{\text{enc}(d) \mid d \in \mathbb{D}\}$, $\{\text{enc}(c) \mid c \text{ is a constant of } \mathbb{D}\}$ and, for all predicates $P \in \mathfrak{R}$ of arity $k \in \mathbb{N}$, $\text{enc}_P = \{\text{enc}(d_1)\varepsilon \dots \varepsilon \text{enc}(d_k) \mid (d_1, \dots, d_k) \in P\}$ (where $\varepsilon \notin A$). When such an encoding exists, we say that $\mathcal{D}$ is *representable*.

Example 11.1 The following data domains are representable:

- $(\mathbb{N}, =)$, with the usual binary encoding
- $(\mathbb{N}, <)$ with the same encoding
- $(\mathbb{Q}, <)$, by encoding a rational number r as the pair $(p, q) \in \mathbb{Z} \times (\mathbb{N} \setminus \{0\})$, where $r = \frac{p}{q}$ is an irreducible fraction, and p and q are themselves encoded in binary.

The set of real numbers is not representable, as there does not exist an injective function $\mathbb{R} \rightarrow A^*$ for any finite set A , since $\mathbb{R}$ is not countable.

Then, a data ω -word is encoded as an ω -word by applying enc pointwise and adding separators. Formally, for $w = d_0 d_1 \dots \in \mathbb{D}^\omega$ we let $\text{enc}(w) = \text{enc}(d_0)\varepsilon \text{enc}(d_1)\varepsilon \dots \in (\{0, 1\}^* \varepsilon)^\omega$.

We now define how a Turing machine can compute a function from $\mathbb{D}^\omega$ to $\mathbb{D}^\omega$. We consider deterministic Turing machines which have three tapes:

- A one-way read-only *input tape* on alphabet $A \sqcup \{\varepsilon\}$, which contains the encoding of the input.

The reader might wonder why we start with the presentation of ω -computability, before providing a solution to the functionality problem. Indeed, if the class of functions recognised by register transducers were undecidable, it would make less sense to study whether a given function is ω -computable (and it would moreover be undecidable). Fortunately, functionality is decidable for the data domains we consider (Theorems 12.8, 12.40 and 12.64), so the reader can let their mind be at peace. And, as it happens, the proof techniques that we use are quite similar for the functionality and ω -computability problem, so it is better to present them successively.

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

[145]: Weihrauch (2000), *Computable Analysis - An Introduction*

Determinism plays a crucial role here, as non-determinism strictly increases expressiveness over infinite words; see Example 11.2.

Note that the tape of a Turing machine cannot be initialised with an infinite word. Here, we assume that the input data ω -word is given as a stream, i.e. the content of the i -th cell is written when the reading head reaches it.

- A two-way read-write *working tape* over some work alphabet W
- A one-way write-only *output tape* over alphabet $A \sqcup \{\epsilon\}$, on which the machine writes the encoding of the output data ω -word.

Convention 11.1 (Implicit Encoding) Since we always work modulo encoding, from now on and in the rest of the chapter, we assume for simplicity that each cell of the Turing machine contains a data value $d \in \mathbb{D}$. Note that the working alphabet can be encoded with constants. Thus, when we say that the input tape contains the input data ω -word $x \in \mathbb{D}^\omega$, we actually mean that it contains the encoding of x , similarly for the output. We discuss in Section 11.1.3 how the computability notions that we introduce hereafter are sensitive to encoding.

11.1.2. Computability and Uniform Computability

Now, consider a Turing machine M built on the above model, and an input data word $x \in \mathbb{D}^\omega$. For any integer $k \in \mathbb{N}$, we let $M(x, k)$ denote the finite output data word written by M on its output tape when it reaches cell number k on its input tape, assuming it does. Since the output tape is write-only, the sequence of data words $(M(x, k))_{k \geq 0}$ is increasing. We denote by $M(x) \in \mathbb{D}^\omega$ the limit content of the output tape.

Definition 11.2 (ω -Computability) Let $\mathcal{D}$ be a representable data domain. A partial data word function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ is ω -computable if there exists a deterministic multi-tape Turing machine M such that for all $x \in \text{dom}(f)$, $M(x) = f(x)$. We say that M *computes* f .

Example 11.2 Consider the (partial) function f_{swap} which reads blocks of data separated by $\$$ and places the last data value of each block at the beginning. Formally, $f_{\text{swap}} : (\mathbb{D}^+ \$)^\omega \rightarrow (\mathbb{D}^+ \$)^\omega$ takes a data word of the form $x_0 d_0 \$ x_1 d_1 \$ \dots$ (where for all $i \in \mathbb{N}$, $x_i \in \mathbb{D}^*$ and $d_i \in \mathbb{D}$) and outputs $d_0 x_0 \$ d_1 x_1 \$ \dots$. This function is ω -computable by a machine which reads its input block by block: on each block $\$ x_i d_i \$$, it goes to the end separator $\$$, goes one step left, stores d_i , goes back to the begin separator, outputs d_i and copies x_i . Besides, it is recognised by the non-deterministic register transducer (with non-deterministic reassignment) of Figure 11.1.

This limit exists and is in $\mathbb{D}^\omega$ since $(\mathbb{D}^\omega, \|\cdot, \cdot\|)$ is complete (cf Proposition 11.1).

Recall that contrary to our models of register automata and register transducer, Turing machines are 2-way, i.e. they can go left and right on their input.

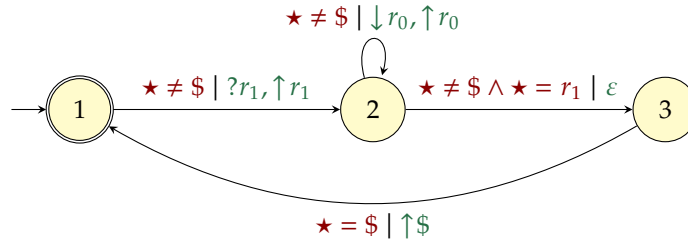


Figure 11.1: A register transducer with non-deterministic reassignment which recognises f_{swap} of Example 11.2. Note that $\uparrow \$$ is not allowed in the syntax, but can be simulated, e.g. by adding a register that is required to contain $\$$.

On the contrary, define a function f_{again} which takes an input data word and repeatedly outputs the first data value d if it appears again in the input, and copies its input otherwise. Intuitively, f is not ω -computable, since the machine cannot decide between outputting d and copying its input as long as it does not see d appear again. Formally, $f_{\text{again}} : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ is defined as $f_{\text{again}}(dx) = d^\omega$ if there exists $i \in \mathbb{N}$ such that $x_i = d$, and $f_{\text{again}}(dx) = dx$ otherwise. This function is not ω -computable: assume

that there is some machine M which computes f_{again} , and consider an input data word dx such that d does not appear in x . Then, there exists some $k \in \mathbb{N}$ such that $dx[0] \leq M(x, k)$, since $f_{\text{again}}(dx) = dx$. As the input tape is one-way and M is deterministic, it behaves the same on all inputs that coincide with x on the first k data values. This is the case in particular for input $x' = d \cdot x[0:k] \cdot d \cdot y$ for some $y \in \mathbb{D}^\omega$, i.e. if we set the $k+1$ -st data value of the input to d . However, $f_{\text{again}}(x') = d^\omega$, so in particular $f_{\text{again}}(x')[1] \neq x[0]$, since we assumed that d does not appear in x . Note that f_{again} is recognised by the non-deterministic register transducer of Figure 11.2.

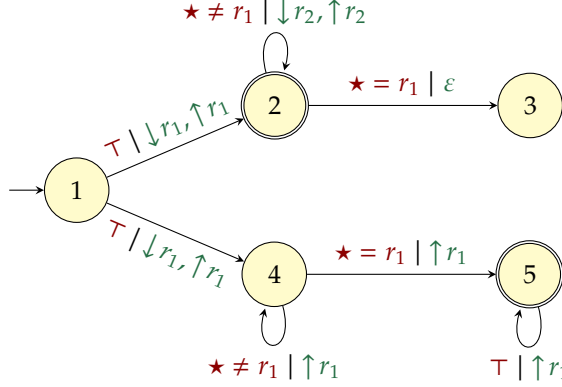


Figure 11.2: A non-deterministic register transducer which recognises f_{again} of Example 11.2. It initially guesses whether d appears again, and checks that its guess is correct along the run. The state 3 can be discarded, as it does not have outgoing transitions, but is left for symmetry.

We can also define stricter notions of computability, e.g. by asking that there exists a bound on the length of the input prefix that is read before the machine is able to produce an output:

Definition 11.3 (Uniform Computability) Let $\mathcal{D}$ be a representable data domain. A partial data word function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ is *uniformly computable* if there exists a deterministic multi-tape Turing machine M , along with a computable function $m : \mathbb{N} \rightarrow \mathbb{N}$ such that M computes f and, moreover: for all $i \geq 0$ and all $x \in \text{dom}(f)$, $|M(x, m(i))| \geq i$. The function m is called a *modulus of computability* for f . In that case, f is called *m -computable*. We say that M *uniformly computes* f , or that M *m -computes* f .

Example 11.3 Come back to the function f_{swap} of Example 11.2. It is not uniformly computable, as the next separator might be arbitrarily far in the input. Formally, assume that f_{swap} is m -computable for some $m : \mathbb{N} \rightarrow \mathbb{N}$. Take input $x = d^{m(1)} \cdot e \cdot \$ \cdot y$ for some $d \neq e \in \mathbb{D}$, and some suffix $y \in (\mathbb{D}^+ \$)^\omega$ whose choice is irrelevant. Then we have, by definition $|M(x, m(1))| \geq 1$. In other words, at least one data value has been output; we call it f . At this point the machine cannot know e as the input reading head is on position $m(1)$ (recall that the input tape is one-way). If, by chance, $f = d$, consider instead an input x' where e is replaced with some $e' \neq e$. Since the machine is deterministic, it cannot distinguish between these two inputs as it only read the first $m(1)$ data values. As a consequence, M does not compute f_{swap} , as it already makes a mistake on the first output data value.

Remark 11.1 (Synchronous programs) If f is synchronously computable, then it is uniformly computable with modulus of computability $\text{id} : i \in \mathbb{N} \mapsto i$, as witnessed by any program P whose ω -behaviour computes f .

Of course, this is not by accident that the terminology resembles that of ‘uniform continuity’ and ‘modulus of continuity’ (cf Theorems 12.10, 12.39 and 12.66).

Recall that a program is a Turing machine.

11.1.3. Robustness to Encoding

Recall that we now work modulo encoding (cf Convention 11.1). Let us examine the incidence of this assumption on our computability notions. Let $\mathcal{D}$ be a representable data domain, and let $\text{enc} : \mathcal{D} \rightarrow A^*$ be its encoding function. For a partial function $f : \mathcal{D}^\omega \rightarrow \mathcal{D}^\omega$, it is encoded as the following $f_{\text{enc}} : (A^*\$)^\omega \rightarrow (A \sqcup \$)^\omega$. It takes an input $w_0\#w_1\#\dots \in (A^*\$)^\omega$, which is meant to be the encoding of some data word $x = \text{enc}^{-1}(w_0)\text{enc}^{-1}(w_1)\dots$. This input x is then fed to f , which yields an output $f(x) = d_0d_1\dots \in \mathcal{D}^\omega$. It finally encodes it as $\text{enc}(d_0)\$\text{enc}(d_1)\$\dots \in (A^*\$)^\omega$. Then, one can check that f is ω -computable if and only if f_{enc} is ω -computable.

However, this is not the case for uniform computability. Take for instance the function f_{snd} which takes as input a data word over some representable data domain $\mathcal{D}$ and removes its first data value. As the encoding of the first data value can be arbitrarily long, there is no bound on the number of bits that have to be read before outputting the first bit (which depends on the second data value). Formally, $f_{\text{snd}} : \mathcal{D}^\omega \rightarrow \mathcal{D}^\omega$ is defined as $f_{\text{snd}}(d_0d_1d_2\dots) = d_1d_2\dots$. Assume that f_{snd} is m -computed by some machine M for some $m : \mathbb{N} \rightarrow \mathbb{N}$, and let $e \neq f \in \mathcal{D}$. Let $i \in \mathbb{N}$ be some position such that $\text{enc}(e)[i] \neq \text{enc}(f)[i]$. Take some $d \in \mathcal{D}$ such that $|\text{enc}(d)| > m(i)$. It necessarily exists since $\mathbb{N}$ is infinite. Now, compare the executions of M on input de^ω and df^ω . By a similar reasoning as for Example 11.3, one can establish that the machine makes a mistake on the i -th output bit.

Despite this limitation, working modulo encoding allows to provide guarantees on the maximal number of input data values that have to be read to produce a given number of output data, even though the encoding of those data can be arbitrarily large.

For instance, for the binary encoding over $\mathbb{N}$, it suffices to take $2^{m(i)}$, encoded as $1 \cdot 0^{m(i)}$.

Note that in practice, most data domains are actually finite, e.g. the integers are encoded over fixed-length sequences of 32 or 64 bits, and the corresponding data values are thus stored in $\mathcal{O}(1)^\dagger$.

11.2. Continuity

As in the finite alphabet case, the notion of ω -computability can be related to the notion of continuity for the Cantor distance [51, Theorem 6]. We first introduce some preliminary notions.

Definition 11.4 (Prefix, longest common prefix, mismatch) Let A be a possibly infinite set. Given two (finite or infinite) words $u, v \in A^\omega$, u is a *prefix* of v , written $u \leq v$, when there exists $w \in A^\omega$ such that $u \cdot w = v$ or $u = v$. We then define $u^{-1}v = w$. It is a *strict prefix*, written $u < v$, when $w \neq \varepsilon$. For $u, v \in A^\omega$, we say that u and v *match*, denoted $u \parallel v$ if either $u \leq v$ or $v \leq u$, and that they *mismatch*, written $u \nparallel v$, otherwise.

Finally, for $u, v \in A^\omega$, we denote $u \wedge v$ their *longest common prefix*: if $u = v$, then $u \wedge v = u$, otherwise it is the longest finite word w such that $w \leq u$ and $w \leq v$. Note that in that case, $|u \wedge v| = \min\{i \in \mathbb{N}, u[i] \neq v[i]\}$.

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

In other words, A is either an alphabet or a data set.

The condition $u = v$ is needed for infinite words, since then we cannot write $u \cdot \varepsilon = v$ as concatenating on the right is undefined.

We distinguish the case of $u = v$, otherwise the notion of longest common prefix is not well-defined when both u and v are infinite.

[†]: One might even argue, although maybe with a little touch of bad faith, that anyway all computers have finitely many states. However, in that case, why even bother with register automata as computers can be modelled with finite-state automata (admittedly in a very inefficient way).

Definition 11.5 (Cantor distance and convergence) We equip the set $\mathbb{D}^\infty$ with the *Cantor distance*: for $u, v \in \mathbb{D}^\infty$, we set $\|u, v\| = 0$ if $u = v$, and $\|u, v\| = 2^{-|u \wedge v|}$ otherwise.

A sequence of (finite or infinite) (data-)words $(w_n)_{n \in \mathbb{N}} \in (A^\infty)^\mathbb{N}$ *converges* to some word $w \in A^\infty$ if:

For all $\epsilon > 0$, there exists $N \geq 0$ such that for all $n \geq N$, $\|w_n, w\| \leq \epsilon$

In our setting, this is better reformulated in terms of the longest common prefix. We ask:

For all $i \in \mathbb{N}$, there exists $N \geq 0$ such that for all $n \geq N$, $|w_n \wedge w| \geq i$

We then say that w is the *limit* of $(w_n)_{n \in \mathbb{N}}$, and that $(w_n)_{n \in \mathbb{N}}$ is *convergent*.

Given a (data-) language $L \subseteq A^\infty$, its *topological closure* $\bar{L}$ is the set of words which can be approached arbitrarily close by words of L , i.e. the set of words which are limits of convergent sequences of L .

Remark 11.2 The structure of the metric space $(A^\infty, \|\cdot, \cdot\|)$ substantially depends on whether A is infinite. Indeed, when A is finite, this space is compact, but it is not when A is infinite. For instance, for $A = \mathbb{N}$, the sequence defined, for all $n \in \mathbb{N}$, by $w_n = n \cdot 0^\omega$ does not have a convergent subsequence, as for all $m \neq n$, $\|w_m, w_n\| = \frac{1}{2}$.

We are now ready to define continuity for functions over infinite (data) words. This is the textbook notion, as one can find it in any analysis lesson.

Definition 11.6 (Continuity) Let $f : I^\omega \rightarrow O^\omega$ be some partial function, where I and O are possibly infinite sets. We say that f is *continuous* at x if, equivalently:

- (i) for all sequences of words $(x_n)_{n \in \mathbb{N}}$ which converge to $x \in \text{dom}(f)$, and which are such that for all $n \in \mathbb{N}$, $x_n \in \text{dom}(f)$, we have that $(f(x_n))_{n \in \mathbb{N}}$ converges to $f(x)$.
- (ii) for all $i \geq 0$, there exists $j \in \mathbb{N}$ such that for all $y \in \text{dom}(f)$, if $|x \wedge y| \geq j$ then $|f(x) \wedge f(y)| \geq i$.

The function f is called *continuous* when it is continuous at each $x \in \text{dom}(f)$.

Definition 11.7 (Uniform continuity) Now, a partial function $f : A^\omega \rightarrow A^\omega$ is *uniformly continuous* if moreover in item (ii) the choice of j does not depend on x and y . Formally, this can be written as:

For all $i \geq 0$, there exists $j \geq 0$ such that for all $x, y \in \text{dom}(f)$,
if $|x \wedge y| \geq j$ then $|f(x) \wedge f(y)| \geq i$

This is equivalent to asking that there exists a mapping $m : \mathbb{N} \rightarrow \mathbb{N}$ such that:

For all $i \geq 0$ and all $x, y \in \text{dom}(f)$, if $|x \wedge y| \geq m(i)$ then $|f(x) \wedge f(y)| \geq i$

The function $m : \mathbb{N} \rightarrow \mathbb{N}$ is called a *modulus of continuity* for f . We also say that f is *m-continuous*.

In the following, we mostly formulate the definitions with regard to the longest common prefix, but the reader can check that this is equivalently formulated using the Cantor distance.

A metric space (X, d) is (sequentially) *compact* when all sequences of X have a convergent subsequence which converges to a point in X .

As noted earlier, the notions of continuity, uniform continuity and modulus of continuity are usually defined with regard to the distance, while here they depend on the length of the longest common prefix, which varies inversely to the distance. The usual notion of modulus of continuity can be recovered, given a mapping $m : \mathbb{N} \rightarrow \mathbb{N}$ which satisfies our definition, by setting $\omega : x \mapsto 2^{-m(\lceil \log_2(\frac{1}{x}) \rceil)}$.

One can also define an intermediate notion of continuity, namely Cauchy continuity. A Cauchy continuous function maps any Cauchy sequence to a Cauchy sequence. As we will see, this notion also admits a counterpart notion of Cauchy computability (cf Definition 11.10).

Definition 11.8 (Cauchy sequence) A *Cauchy sequence* in A is a sequence $(x_n)_{n \in \mathbb{N}}$ such that:

For all $i \in \mathbb{N}$, there exists some $N \geq 0$ such that
for all $m, n > N$, $|x_m \wedge x_n| \geq i$

It is well-known that for any finite alphabet A , the set $(A^\omega, \|\cdot, \cdot\|)$ is *complete*, i.e. every Cauchy sequence converges to an element of A^ω . We show that this is also the case for infinite alphabets.

Proposition 11.1 For any (possibly infinite) set $\mathbb{D}$, $(\mathbb{D}^\omega, \|\cdot, \cdot\|)$ is complete.

The proof is actually the same, as it actually does not depend on the hypothesis that A is finite.

Proof. Let $\mathbb{D}$ be some set, and let $(x_n)_{n \in \mathbb{N}} \in \mathbb{D}^\mathbb{N}$ be a Cauchy sequence. We first define the candidate limit x . Let $i \in \mathbb{N}$. We know that there exists some $N \geq 0$ such that for all $m, n > N$, $|x_m \wedge x_n| \geq i$. Define $x[i] = x_m[i]$ for some $m \geq N$. The choice of m is irrelevant as all x_m coincide on their first i values. Then, for each $i \in \mathbb{N}$, choose N as above. For all $n \geq N$, we have $|x_n \wedge x| \geq i$, which means that (x_n) converges to x . $\square$

Definition 11.9 (Cauchy continuity) A partial function $f : A^\omega \rightarrow B^\omega$ is *Cauchy continuous* if for all Cauchy sequences $(x_n)_{n \in \mathbb{N}} \in (\text{dom}(f))^\mathbb{N}$, $(f(x_n))_{n \in \mathbb{N}}$ is a Cauchy sequence.

Given a Cauchy continuous function f , we denote by $\bar{f}$ its continuous extension over $\text{dom}(f)$.

Fact 11.1 An interesting property of this class of functions is that they always admit a (unique) continuous extension to the completion of their domain. Since we deal with A^ω which is complete, the completion of a set is simply its closure.

Example 11.4 By definition, uniform continuity implies Cauchy continuity. In the finite alphabet case, and in general for compact spaces, Cauchy continuity and uniform continuity actually coincide. In particular, the function f_{snd} of Section 11.1.3 is Cauchy continuous. Conversely, one can check that the function f_{swap} is not Cauchy continuous, e.g. by showing that it does not admit any continuous extension on data words which do not contain the \$ separator.

Remark 11.3 Over infinite alphabets, equivalence between Cauchy continuity and uniform continuity does not hold anymore. Consider for instance the function $f_{\text{decr}} : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$ which takes as input a data word $u \cdot 0 \cdot x$, where $u \in \mathbb{N}^*$ is strictly decreasing, and outputs $x \in \mathbb{N}^\omega$. This function is not uniformly continuous, since two words may be arbitrarily close and have different images. Formally, for all $i \in \mathbb{N}$, take $u = i \cdot (i-1) \cdot \dots \cdot 1$, then we have $|u \cdot 0 \cdot 0^\omega \wedge u \cdot 0 \cdot 1^\omega| = i$, but $|0^\omega \wedge 1^\omega| = 0$. Note that f_{decr} can be computed by a sequential register transducer, which waits to see 0 in its input and then copies its input.

However, one can check that the image of a Cauchy sequence is indeed Cauchy: up to removing a prefix, we can assume that all x_n start with the same data value, which sets a bound on the length of the input to

read before reaching 0 and outputting something. Formally, let $(x_n)_{n \in \mathbb{N}}$ be a Cauchy sequence in $\text{dom}(f_{\text{decr}})$. First, let $K \geq 0$ be such that for all $m, n \geq K$, $x_n[0] = x_m[0] = d \in \mathbb{N}$. Such a K exists since (x_n) is Cauchy, so in particular at some point all elements of the sequence are at distance less than $\frac{1}{2}$. Now, let $i \geq 0$; we want to find some $N \geq 0$ such that all elements after N have a common prefix of length at least i . Choose N to be such that for all $m, n \geq N$, $|x_m \wedge x_n| \geq d + i + 1$. Then we know that there exists $j \leq d$ such that $x_m[j] = x_n[j] = 0$, since the input has to strictly decrease until 0 is reached. Thus, $x_m = y \cdot 0 \cdot z \cdot t_m$ and $y_n = y \cdot 0 \cdot z \cdot t_n$, where y is of length $j - 1$, z is of length i , and $t_m, t_n \in \mathbb{N}^\omega$. As a consequence, $f(x_m) = z \cdot t_m$ and $f(x_n) = z \cdot t_n$, so $z \leq f(x_m) \wedge f(x_n)$, which means that $|f(x_m) \wedge f(x_n)| \geq i$. Overall, this means that $(f(x_n))_{n \in \mathbb{N}}$ is a Cauchy sequence.

Remark 11.4 We could further refine the notions of continuity, by studying m -continuity for particular m . For instance, for $m : i \mapsto i + b$ for some $b \in \mathbb{N}$, m -continuous functions are exactly 2^b -Lipschitz continuous functions. Similarly, for $m : i \mapsto a \cdot i + b$ (where $a \in \mathbb{N} \setminus \{0\}$), m -continuous functions are $\frac{1}{a}$ -Hölder continuous functions.

However, both notions are very sensitive to the metric that we choose. For instance, the metric $d(x, y) = \frac{1}{|x \wedge y|}$ induces the same topology over $\mathbb{D}^\omega$, but yields different notions of Lipschitz and Hölder continuity. For this reason, we stick to continuity, Cauchy continuity and uniform continuity.

11.3. ω -Computability versus Continuity

In this section, we show that the different computability notions imply their respective continuity notion, and give general conditions for the converse to hold. First, let us introduce the notion of Cauchy computability, which is the counterpart of Cauchy continuity.

Definition 11.10 (Cauchy computability) Let $\mathcal{D}$ be a representable data domain. A data word partial function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ is *Cauchy computable* if there exists a deterministic multi-tape Turing machine M that computes f and such that for all x in the topological closure $\overline{\text{dom}(f)}$ of $\text{dom}(f)$, the sequence $(M(x, k))_{k \in \mathbb{N}}$ converges to an infinite word. In other words, a Cauchy computable function is a function which admits a continuous extension to the closure of its domain and which is ω -computable. We say M *Cauchy computes* f .

11.3.1. From ω -Computability to Continuity

In the Cantor topology, continuity means that a given output element only depends on a finite prefix of the input; Cauchy and uniform continuity successively refine this idea. In our definition, the computation is conducted by a Turing machine which deterministically reads its input one-way, so an element of output is determined by a finite prefix. We used this fact in Example 11.2, to show that f_{again} is not ω -computable, and in Example 11.3, to establish that f_{swap} is not uniformly computable (see also [5, Section 6]). This is formally captured as follows:

Theorem 11.2 ([142, Theorem 2.14]) Let $\mathcal{D}$ be a representable data domain. Let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$, be a partial function. The following implications hold:

[5]: Thomas (2009), ‘Facets of Synthesis: Revisiting Church’s Problem’

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

- (i) If f is ω -computable, then f is continuous
- (ii) If f is Cauchy computable, then f is Cauchy continuous
- (iii) If f is uniformly computable, then f is uniformly continuous
- (iv) If f is m -computable for some mapping $m : \mathbb{N} \rightarrow \mathbb{N}$, then f is m -continuous

Proof. We start by jointly proving items (i) and (ii). Assume that f is ω -computable by a deterministic multi-tape Turing machine M . Let x be in the topological closure of $\text{dom}(f)$, and let $(x_n)_{n \in \mathbb{N}}$ be a sequence in $\text{dom}(f)$ converging to x . We show that $(f(x_n))_{n \in \mathbb{N}}$ converges to $M(x)$ if $M(x)$ is infinite. Since $\lim_{n \rightarrow \infty} x_n = x$, for all $k \geq 0$, there exists an $n_k \in \mathbb{N}$ such that for all $m \geq n_k$, $x[k] \leq x_m$. As M is a deterministic machine, it implies that $M(x, k) \leq f(x_m)$. As a consequence, for all k , $M(x, k) \leq f(x_m)$ for all but finitely many m . If $M(x)$ is an infinite word, it means that $(f(x_m))_{m \in \mathbb{N}}$ converges to $M(x) = \lim_{n \rightarrow \infty} M(x, k)$. It is the case for all $x \in \text{dom}(f)$ since f is ω -computable, which means that f is continuous. If additionally f is Cauchy computable, this also holds for all x in the closure of $\text{dom}(f)$, so f is Cauchy continuous.

We now show item (iv), which entails item (iii). Assume that f is m -computable by some machine M . We show f is m -continuous. So, let $i \geq 0$ and $x, y \in \text{dom}(f)$ such that $|x \wedge y| \geq m(i)$. We must show that $|f(x) \wedge f(y)| \geq i$. Since M is deterministic, we know that $M(x, m(i)) = M(y, m(i))$, as $|x \wedge y| \geq m(i)$. Since M m -computes f , we also have $|M(x, m(i))| \geq i$. Finally, $M(x, m(i)) \leq f(x)$ and $M(y, m(i)) \leq f(y)$, we get $|f(x) \wedge f(y)| \geq i$, which concludes the proof. $\square$

11.3.2. From Continuity to ω -Computability

As we have seen, ω -computability implies continuity, and similarly for the refined notions of uniform and Cauchy computability. The converse does not hold in general, already over finite alphabets:

Example 11.5 Assume some enumeration $(M_i)_{i \in \mathbb{N}}$ of Turing machines, and define $f_{\text{halt}} : A^\omega \rightarrow \{0, 1\}^\omega$ which simply ignores its input and outputs the sequence $h_0 h_1 \dots \in \{0, 1\}^\omega$, where, for all $i \in \mathbb{N}$, $h_i = 1$ whenever M_i halts. f_{halt} is not ω -computable, as a machine that computes it would be able to answer the halting problem, which is undecidable [1, 119, 120]. However, f_{halt} is continuous, even uniformly so, as it is constant.

We give a sufficient condition under which continuity implies ω -computability (and similarly for the refined notion), namely that f has a computable next-letter problem. This problem asks, given as input two finite data words $u, v \in \mathbb{D}^*$, to output, if it exists, a data value $d \in \mathbb{D}$ such that for all suffixes $y \in \mathbb{D}^\omega$ such that $u \cdot y \in \text{dom}(f)$, we have $v \cdot d \leq f(u \cdot y)$. Note that d is unique if it exists, since y is universally quantified.

Problem 11.1: NEXT-LETTER PROBLEM

Input: $u, v \in \mathbb{D}^*$
Output: $d \in \mathbb{D}$ such that for all $y \in \mathbb{D}^\omega$ which satisfy $u \cdot y \in \text{dom}(f)$,
 $v \cdot d \leq f(u \cdot y)$
if it exists
No otherwise

Here, we consider usual Turing machines, i.e.; those which operate over a finite alphabet. We assume they do not take any input.

Note that the choice of the input alphabet A does not matter since f_{halt} ignores its input.

[1, 119]: Turing, Turing (1936, 1938), ‘On Computable Numbers, with an Application to the Entscheidungsproblem’, ‘On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction’; see also [120]: Sipser (2013), *Introduction to the Theory of Computation*. Here, we restrict to machines that ignore their input, but it is easy to reduce from the emptiness problem for Turing machines, which is undecidable [120, Theorem 5.2].

A constant function is continuous for about any (reasonable) notion of continuity that one can imagine.

We call *next-letter decision problem* the decision version of the above problem, which simply asks whether the asked $d \in \mathbb{D}$ exists.

If the above problem is computable, then the converse of Theorem 11.2 holds:

Theorem 11.3 ([142, Theorem 2.15]) *Let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ be a partial function with a computable next-letter problem. The following implications hold:*

- (i) *If f is continuous, then f is ω -computable*
- (ii) *If f is Cauchy continuous, then f is Cauchy computable*
- (iii) *If f is uniformly continuous, then f is uniformly computable*
- (iv) *If f is m -continuous for some mapping $m : \mathbb{N} \rightarrow \mathbb{N}$, then f is m -computable*

Proof. Assume that there exists a procedure $\text{Next}_f(u, v)$ which computes the next-letter problem. We exhibit a deterministic multi-tape Turing machine M_f common to the four statements, in the sense that it computes f when f is continuous, respectively Cauchy computes f when f is Cauchy continuous, etc. M_f executes the following pseudo-code:

Algorithm 2: The Turing machine M_f defined in pseudo-code.

Input: $x \in \mathbb{D}^\omega$

```

1  $v := \varepsilon$ ;
2 for  $i = 0$  to  $+\infty$  do
3    $d := \text{Next}_f(x[:i], v)$ ;
4   while  $d \neq \text{No}$  do
5     output  $d$ ;                // write  $d$  on the output tape
6      $v := v \cdot d$ ;
7      $d := \text{Next}_f(x[:i], v)$ ;
   /* end while loop when  $d = \text{No}$  */

```

We now show that if f is continuous, then M_f computes f , i.e. $M_f(x) = f(x)$ for all $x \in \text{dom}(f)$. For all $i \geq 0$, define $v_i = d_0^i d_1^i \dots$ the (possibly infinite) sequence of data values output at line 4 at the i -th iteration of the for loop of line 2. Note that the test at line 5 can be forever true, which makes the corresponding v_i infinite, and entails that the following j -th iterations may not exist. In that case, we set $v_j = \varepsilon$. As the input reading head moves to the right at the end of the for loop, we have that $M_f(x, i) = v_0 \cdot v_1 \dots v_i$ for all $i \geq 0$. Moreover, by definition of the next-letter problem, we also have $M_f(x, i) \leq f(x)$. Finally, on input x , note that M_f outputs $M_f(x) = v_0 \cdot v_1 \cdot v_2 \dots$. As a consequence, if $M_f(x)$ is infinite, then $M_f(x) = f(x)$. It remains to show that this is the case when f is continuous at x . Suppose the contrary. Then, there exists i_0 such that for all $i \geq i_0$, the call to $\text{Next}_f(x[:i], v_0 \dots v_{i_0})$ returns No. Assume i_0 is the smallest index having this property, and let $d \in \mathbb{D}$ be such that $v_0 \dots v_{i_0} \cdot d < f(x)$. Then, for all $i \geq i_0$, there exists $y_i \in \mathbb{D}^\omega$ and $e \neq d$ such that $x[:i]y_i \in \text{dom}(f)$ and $v_0 \dots v_{i_0} \cdot e \leq f(x[:i]y_i)$. Clearly, the sequence $(f(x[:i]y_i))_{i \in \mathbb{N}}$, if it converges, does not converge to $f(x)$. Since $(x[:i]y_i)_i$ converges to x , this contradicts the continuity of f . We thus established item (i).

Consider now the case where f is Cauchy continuous. It implies that f admits a (unique) continuous extension $\bar{f}$ to $\overline{\text{dom}(f)}$ (see Fact 11.1). By item (i), that we just proved, it means that $M_{\bar{f}}$ computes $\bar{f}$. By definition

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

The i -th iteration lasts forever when the output only depends on the i -th first data values. This is the case e.g. for constant functions, for $i = 0$.

By convention, for any data ω -word w , we extend concatenation by letting $w \cdot \varepsilon = w$.

One can actually show that the following property holds: denote $m_x : \mathbb{N} \rightarrow \mathbb{N}$ the optimal modulus of continuity of f at x , i.e. the mapping which, to each $i \in \mathbb{N}$, associates $\min\{j \in \mathbb{N} \mid \forall y \in \text{dom}(f), |x \wedge y| \geq j \Rightarrow |f(x) \wedge f(y)| \geq i\}$. The minimum exists for all $i \in \mathbb{N}$ by definition of continuity (Definition 11.6, item (ii)). Then, the i -th data value of $f(x)$ is output at the $m_x(i)$ -th iteration of the for loop.

d is chosen as the $(|v_0| + \dots + |v_{i_0}| + 1)$ -th data value of $f(x)$.

of Cauchy computability, this means that M_f Cauchy computes f . Note that we have $\text{Next}_{\bar{f}} = \text{Next}_f$.

Finally, we show item (iv), which implies item (iii). Assume that f is m -continuous for some modulus of continuity $m : \mathbb{N} \rightarrow \mathbb{N}$. Let us show that M_f m -computes f . Since f is continuous, we already know that M_f computes f . Let $i \geq 0$, and $x \in \text{dom}(f)$. We have to show that $|M_f(x, m(i))| \geq i$. Let $j = m(i)$. Since the algorithm M_f calls $\text{Next}_f(x[:j], \cdot)$ until it returns No, we get that $v_0 v_1 \dots v_j$ is the longest output which can be safely output given only $x[:j]$, i.e. $v_0 \dots v_j = \bigwedge \{f(x[:j] \cdot y) \mid x[:j] \cdot y \in \text{dom}(f)\}$. If v_j is infinite, then $|M_f(x, j)| = +\infty$ and we are done. Suppose v_j is finite. Then, there exists $y \in \mathbb{D}^\omega$ such that $x[:j] \cdot y \in \text{dom}(f)$ and $f(x) \wedge f(x[:j] \cdot y) = v_0 v_1 \dots v_j = M_f(x, j)$. Since $|x \wedge x[:j] \cdot y| \geq j = m(i)$, by m -continuity of f , we get that $|M_f(x, j)| \geq i$, concluding the proof. $\square$

Exploration of Various Data Domains

12.

We now move to the study of various data domains. In all cases, the argument relies on characterising functionality, continuity and its related notions with a forbidden pattern that can be searched for by a decision procedure. The case of $(\mathbb{D}, =, \mathbb{C})$ is the most elementary, as the renaming property allows to reduce to the case of a finite alphabet (Theorem 12.11), itself characterised by a forbidden pattern [51, Figure 2 and Lemma 11]. We then move to the case of oligomorphic data domains, which encompasses that of $(\mathbb{D}, =, \mathbb{C})$. Intuitively, those domains are ‘almost finite’, in the sense that any tuple has finitely many orbits, i.e. there are finitely many valuations up to automorphisms of the domain. Since languages and relations recognised by register machines are closed under automorphisms, this property allows us to abstract their behaviour in a finite way, and define a notion of loop (Section 12.2.3) that allows to use pumping arguments to exhibit a forbidden pattern, and to further obtain a small witness property for such a pattern (Lemma 12.33). This setting supersedes $(\mathbb{D}, =, \mathbb{C})$, since then a valuation is characterised by the equalities between the content of each register. We chose to present the first case independently, as the reasonings are simpler. Finally, we move to the case of $(\mathbb{N}, <, 0)$, which is not oligomorphic (cf Example 12.3). It then implies refining the notion of loop to again get a decidable characterisation, this time using a Ramsey argument (Propositions 12.56 and 12.57). This section concludes with a method to transfer this result to similar structures, e.g. $(\mathbb{Z}, <)$, using the notion of quantifier-free interpretation.

The proof techniques we use have in common the following structure: first, we characterise non-functionality and non-continuity by structural patterns on non-deterministic register transducers and establish small-witness properties for the existence these patterns. Then, based on the small witness properties, we show how to decide whether given a register transducer, such patterns are matched or not. As this chapter consists in pushing further and further the decidability border, it might incur some redundancy between the different cases. However, it does not seem possible to factor all proofs using a reasonable amount of additional notations, except for $(\mathbb{D}, =, \mathbb{C})$, which is included for pedagogical purposes.

12.1. The Case of Data Domains with Equality

We start with the case of $(\mathbb{D}, =, \#)$. As stated above, the results in this section follow from the study of the oligomorphic case of Section 12.2, since data domains with equality are oligomorphic (cf Example 12.3). We prefer to treat this particular case in its own right, as the renaming property yields conceptually simpler proofs which present a pedagogical interest. This is moreover sufficient for the reader who is only interested in this case. The impatient reader can jump to Section 12.2, which presents standalone proofs for the oligomorphic case. Note that to lighten the argument, we defer the proof of some secondary results

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

Adding constants does not affect oligomorphicity, see Proposition 12.14.

to Section 12.2, namely equivalence between uniform continuity and Cauchy continuity (Theorem 12.29), and the characterisation and decision of uniform continuity, which is repetitive. We still give some hints on how to deduce them without diving into the oligomorphicity terminology.

The key property of data domains with equality is that machines with registers enjoy the renaming property (see Section 4.5.2 for a more detailed study of this property and its implications), even when non-deterministic reassignment is allowed:

Proposition 12.1 (Renaming property in the presence of non-deterministic reassignment [38, Proposition 14]) *Let A be a non-deterministic register automaton with k registers with non-deterministic reassignment over some data domain $(\mathbb{D}, =, \mathbb{C})$. Assume that A accepts some data word $w \in \mathbb{D}^\omega$. For all k -renaming sets $X \subseteq \mathbb{D}$, there exists a data word $w' \in X^\omega$ which is accepted by A .*

Moreover, it admits an accepting run whose valuations take their values in X .

Proof. The proof is the same as that of Proposition 4.15, and we do not redo it here. Observe that the valuations of the run we build indeed take their values in X , i.e. guessing can be restricted to the values in X . $\square$

Moreover, machines with registers over $(\mathbb{D}, =)$ behave like ω -automata over any finite subset of data values, again even when non-deterministic reassignment is allowed:

Proposition 12.2 ([38, Proposition 9]) *Let A be a register automaton with non-deterministic reassignment over data domain $(\mathbb{D}, =, \mathbb{C})$. For any finite subset $X \subseteq \mathbb{D}$, $L(A) \cap X^\omega$ is ω -regular.*

More precisely, $L(A) \cap X^\omega$ is recognised by an ω -automaton A_X with $n(\max(s, k) + c + 1)^k$ states, where n is the number of states of A , k its number of registers, $s = |X|$ is the size of X and $c = |\mathbb{C}|$. Moreover, if A is deterministic, so is A_X .

Proof. Note that contrary to the non-guessing case (Proposition 4.1), this property does not hold for all data domains (see Property 4.56 and Remark 10.2). However, in the case of $(\mathbb{D}, =, \mathbb{C})$, we can use the second part of the renaming property to obtain that if $w \in L(A)$, it admits an accepting run that takes its values in a finite renaming set.

Let us make the argument more precise: let A be a register automaton (with guessing) with states Q and registers R , and let $X \subset_f \mathbb{D}$ be some finite set of data values. Then, take a k -renaming set Y that contains X ; its size can be bounded by $\max(s, k) + c + 1$. By Proposition 12.1, we know that if $w \in L(A) \cap X^\omega$, then it admits an accepting run whose valuations take their values in Y . Thus, it suffices to simulate A on the finite set Y of data values, which can be done by an ω -automaton which stores valuations in memory using a state space $Q \times Y^R$. The construction is a straightforward generalisation of that of Proposition 4.1: the ω -automaton simply needs to additionally guess some value in Y for registers r such that $\text{regOp}(r) = \text{guess}$. $\square$

This result can easily be lifted to non-deterministic transducers, yielding

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

A k -renaming set is a set of the form $X = Y \sqcup \mathbb{C}$, where $Y \subseteq \mathbb{D} \setminus \mathbb{C}$ is of size $|Y| > k$ (Definition 4.21).

Recall that this property holds over any data domain when non-deterministic reassignment is disallowed (Proposition 4.1), but breaks otherwise (Remark 10.2).

Proposition 12.3 *Let T be a register transducer over data domain $(\mathbb{D}, =, \mathbb{C})$. For any finite subset $X \subseteq \mathbb{D}$, $\llbracket T \rrbracket \cap (X^\omega \times X^\omega)$ is ω -rational.*

More precisely, $\llbracket T \rrbracket \cap (X^\omega \times X^\omega)$ is recognised by an ω -transducer T_X with $n(\max(s, k) + c + 1)^k$ states, where n is the number of states of A , k its number of registers, $s = |X|$ is the size of X and $c = |\mathbb{C}|$.

As we demonstrate, those properties jointly allows to reduce functionality and continuity problems to the data-free case.

A relation is ω -rational when it can be recognised by a non-deterministic asynchronous transducer.

12.1.1. The Functionality Problem

We start with the functionality problem, as it is more elementary. We start by recalling that it is decidable in the data-free case, i.e. for functions defined by non-deterministic asynchronous transducers with no registers (Theorem 12.4). We then reduce the decision of functionality for register transducers to the data-free case by leveraging the renaming property: if a register transducer is not functional, then this is witnessed by an input data word and two distinct output data words that take their values in a finite set of data values, whose size only depends on the number of registers of the transducer (Proposition 12.7).

Functionality in the Data-Free Case

In this section, Σ and Γ denote two finite alphabets.

By [146, Proposition 3.3], functionality is decidable for transducers with no registers. By relying on the pattern logic of [147] designed for transducers of *finite* words, we can show that the problem more precisely belongs to $NLogSPACE$.

Theorem 12.4 ([146, Proposition 3.3], [147]) *Deciding whether a non-deterministic asynchronous transducer is functional is in $NLogSPACE$.*

Proof (Main Ideas). A non-deterministic asynchronous transducer T is not functional whenever a given input word admits at least two distinct outputs. Since both outputs are ω -words, they are distinct whenever there is a position at which they mismatch, i.e. an i such that their i -th letters are different. Using pumping arguments, one can show that if such a mismatch occurs at some point, there exists an input of the form $u \cdot v^\omega$ for words $u \in \Sigma^*$ and $v \in \Sigma^+$ whose length is polynomial in the size of T , and that has two accepting runs in T respectively yielding outputs $u_1(v_1)^\omega$ and $u_2(v_2)^\omega$ such that u_1 and u_2 are output on reading u and $u_1 \not\parallel u_2$. A non-deterministic algorithm can thus guess the accepting runs over the two inputs, which are lasso-shaped. Using a polynomially-bounded counter, it keeps track of the length of the output on the runs over the two mismatching outputs, and checks that there is indeed a mismatch. It finally checks that both runs can be jointly completed with the same input word to yield an accepting run. $\square$

[146]: Gire (1986), ‘Two Decidability Problems for Infinite Words’

[147]: Filiot, Mazzocchi, and Raskin (2018), ‘A Pattern Logic for Automata with Outputs’

A decision problem is in $NLogSPACE$ if there exists a non-deterministic Turing machine which decides it using only logarithmic space. It is well-known that those algorithms can be run deterministically in polynomial time, i.e. $NLogSPACE \subseteq PTIME$. The construction is the same as for showing $PSPACE \subseteq ExpTIME$.

A run is *lasso-shaped* if it is of the form $\rho = \pi\lambda^\omega$ for finite sequences of transitions π and λ .

Intermezzo: the Pattern Logic

Many problems on ω -transducers reduce to finding a pattern in a machine. This is the case for instance of determinisability, which has been characterised by the twinning property [148, Proposition 3.4]. As we point out in a few lines, this is also the case for functionality. Finally, continuity and uniform continuity also fall in this framework [51, Lemma 11], and our characterisation of these notions in the infinite alphabet case relies on an extension of these patterns. In [147, 149], the authors introduce the aptly named pattern logic, which allows for a uniform treatment of the decision of the existence of those patterns, and allows to abstract pumping arguments. We seize the occasion of this reasonably simple proof to present the logic, as we use it as a tool for our subsequent decision procedures. It is designed for the general setting of finite-state machines operating over some monoid. In the case of finite-state transducers, the model-checking problem for this logic is easily shown undecidable, but the authors introduce a decidable fragment that they call PL_{trans} [149, Theorem 9 and Section 5]. We present the syntax of this fragment, that we simply call PL or pattern logic, and provide its semantics on an intuitive level. We refer to [149] for a general presentation. Note that this logic is defined for transducers over *finite* words.

Definition 12.1 (Pattern logic [149, Definition 10]) Formulas of the *pattern logic* are of the following form, for some $n \in \mathbb{N}$:

$$\begin{aligned} \varphi &::= \exists \pi_1 : p_1 \xrightarrow{u_1|v_1} q_1, \dots, \exists \pi_n : p_n \xrightarrow{u_n|v_n} \cdot \mathcal{C} \\ \mathcal{C} &::= \neg \mathcal{C} \mid \mathcal{C} \vee \mathcal{C} \mid u \leq u' \mid u \in L \mid |u| \leq |u'| \mid \\ &\quad \text{init}(q) \mid \text{final}(q) \mid q = q' \mid \pi = \pi' \mid \\ &\quad t \not\leq t' \mid t \in N \mid |t| \leq |t'|, \text{ where:} \end{aligned}$$

- For all $1 \leq i < j \leq n$, $\pi_i \neq \pi_j$ and $v_i \neq v_j$, i.e. we ask that the path variables π and the output variables are all pairwise distinct, to disallow equality tests between outputs
- The set L (respectively N) are regular languages over some alphabet Σ (respectively Γ), given as a deterministic finite automaton
- $u, u' \in \{u_1, \dots, u_n\}$ are *input variables*
- $q, q' \in \{p_1, \dots, p_n, q_1, \dots, q_n\}$ are *state variables*
- $t, t' \in \text{Terms}(\{v_1, \dots, v_n\}, \cdot, \varepsilon)$ are *terms* over the *output variables*
- $\pi, \pi' \in \{\pi_1, \dots, \pi_n\}$ are *path variables*.
- We finally require that $t \not\leq t'$ does not occur under an odd number of negations.

We now give the semantics on an informal level: φ states that there exists n paths in the transducer that satisfy the condition $\mathcal{C}$. Then, negation and disjunction are interpreted as usual. The symbol $\leq$ is interpreted as the prefix relation, and $\in, |\cdot|, =$ are interpreted as expected (respectively set membership, word length and equality). $\text{init}(q)$ means that q is initial, i.e. $q \in I$, and $\text{final}(q)$ that it is final, i.e. $q \in F$.

Note that one can define syntactic sugar for non-equality of outputs $t \neq t' := t \not\leq t' \vee t' \not\leq t$, as well as for mismatch: $\text{mismatch}(t, t') := t \not\leq t' \wedge t' \not\leq t$.

As shown in [149, Theorem 11], the pattern logic has a decidable model-

[148]: Choffrut (1977), ‘Une Caractérisation des Fonctions Séquentielles et des Fonctions Sous-Séquentielles en tant que Relations Rationnelles’, in French. The twinning property is called « propriété de jumelage ».

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

[147]: Filiot, Mazzocchi, and Raskin (2018), ‘A Pattern Logic for Automata with Outputs’

[149]: Filiot, Mazzocchi, and Raskin (2020), ‘A Pattern Logic for Automata with Outputs’

Tests between outputs are the main cause of undecidability, as they allow to encode the Post Correspondence Problem.

This precaution is again to disallow the expression of equality tests between outputs.

Recall that conjunction are easily derived through De Morgan’s laws.

[149]: Filiot, Mazzocchi, and Raskin (2020), ‘A Pattern Logic for Automata with Outputs’

checking problem. More precisely:

Theorem 12.5 ([149, Theorem 11]) *The model checking of transducers against formulas in PL is PSPACE-complete. It is NLOGSPACE-complete when the formula is fixed.*

Example 12.1 Since we are operating over ω -words, we present as examples simple patterns that will serve as macros in the following. First, as can be established using a simple pumping argument, an ω -transducer accepts some input ω -word if and only if it accepts an input ω -word of the form $u \cdot v^\omega$ for some $u, v \in \Sigma^*$. This is characterised by the pattern of Figure 12.1.

$$\exists \pi_1 : p \xrightarrow{x|x'} q, \exists \pi_2 : q \xrightarrow{y|y'} q \cdot \text{init}(p) \wedge \text{final}(q)$$

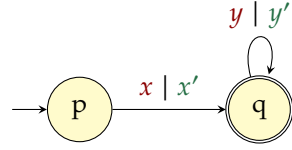


Figure 12.1: A pattern characterising non-emptiness of the domain of functions definable by a non-deterministic ω -transducer. We accompany the formula with its graphical depiction.

The attentive reader may have noticed that since the word can be decomposed in multiple ways, we do not necessarily have $u = x$ and $v = y$. Note also that we put no conditions on x' and y' .

This pattern also allows to specify that a given state is co-accessible, i.e. there exists an input word which yields a final run, starting from this state: $\text{coacc}(p)$ is defined as above, except that we do not ask $\text{init}(p)$.

Sometimes, we further require that two states are *jointly coaccessible*, i.e. that the same input word yields a final run from *both* states. Formally, p and q are jointly coaccessible, written $\text{jtcoacc}(p, q)$ whenever there exists an infinite input ω -word $x \in \Sigma^\omega$ such that $p \xrightarrow{x|x'}$ and $q \xrightarrow{x|x''}$ for some $x', x'' \in \Gamma^\omega$. A simple pumping argument establishes that this is characterised by the pattern of Figure 12.2. We make the argument precise, for the sake of completeness. p and q are jointly coaccessible whenever there exists two finite runs π_1 and π_2 of the following form, where the outputs have not been indicated as they do not matter for this property:

$$\begin{aligned} \pi_1 : p &\xrightarrow{u} p' \xrightarrow{v} p'' \xrightarrow{w} p' \\ \pi_2 : q &\xrightarrow{u} q' \xrightarrow{v} q'' \xrightarrow{w} q' \end{aligned}$$

such that p' and q'' are accepting states. Indeed, since we use a Büchi acceptance condition, if p and q are jointly co-accessible by a common ω -word $x \in \Sigma^\omega$, then we can find a loop in each run over x and exhibit two runs over some word $u \cdot (v \cdot w)^\omega$, on which there is a run from p with some accepting states p' and a run from q with some accepting state q'' , both occurring on the loop of the lasso. Therefore, $\text{jtcoacc}(p, q)$ holds whenever the pattern logic formula of Figure 12.2 is true against T (seen as a transducer over finite words).

We can now come back to the formal proof of the decidability and complexity of the functionality problem for ω -transducers.

Proof of Theorem 12.4. Let T be some NFT with input alphabet Σ and output alphabet Γ . By definition, $\llbracket T \rrbracket$ maps ω -words to ω -words (cf

$$\begin{aligned}
& \exists \pi_1 : p \xrightarrow{u} p' \quad \exists \pi'_1 : p' \xrightarrow{v} p'' \quad \exists \pi''_1 : p'' \xrightarrow{w} p' \\
& \exists \pi_2 : q \xrightarrow{u} q' \quad \exists \pi'_2 : q' \xrightarrow{v} q'' \quad \exists \pi''_2 : q'' \xrightarrow{w} q' \\
& \text{final}(p') \wedge \text{final}(q'')
\end{aligned}$$

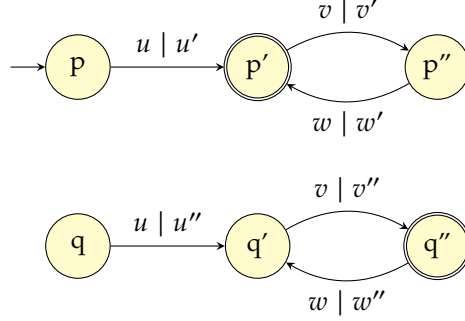


Figure 12.2.: A pattern characterising joint coaccessibility of two states p and q in a non-deterministic ω -transducer.

Assumption 10.3), so we have that T is *not* functional if and only if there exist three ω -words $x \in \Sigma^\omega$ and $y_1, y_2 \in \Gamma^\omega$ such that $(x, y_1) \in \llbracket T \rrbracket$, $(x, y_2) \in \llbracket T \rrbracket$ and $y_1 \neq y_2$, which implies that there is a mismatch between y_1 and y_2 , i.e. a position $i \in \mathbb{N}$ such that $y_1[i] \neq y_2[i]$. By taking a sufficiently long prefix $u \in \Sigma^*$ of x , the latter is equivalent to the existence of finite runs $\pi_1 : p_1 \xrightarrow{u|v_1} q_1$ and $\pi_2 : p_2 \xrightarrow{u|v_2} q_2$, where p_1, p_2 are initial, q_1, q_2 are jointly co-accessible (by a common ω -word) and there is a mismatch between v_1 and v_2 . This property is expressible in the pattern logic. More precisely, we model-check T against the following pattern formula:

$$\begin{aligned}
& \exists \pi_1 : p_1 \xrightarrow{u|v_1} q_1 \quad \exists \pi_2 : p_2 \xrightarrow{u|v_2} q_2 \\
& \text{init}(p_1) \wedge \text{init}(p_2) \wedge \text{jcoacc}(q_1, q_2) \wedge \text{mismatch}(v_1, v_2)
\end{aligned}$$

The predicate $\text{jcoacc}(q_1, q_2)$ is defined in Example 12.1. Recall that the pattern logic operates over finite words and paths, which is why we need to encode $\text{jcoacc}(q_1, q_2)$. This concludes the proof. $\square$

Reducing to the Data-Free Case

As established in Section 10.2, traces of runs of a non-deterministic register transducer can be recognised by a non-deterministic register automaton. Over data domains with equality, this model enjoys the renaming property, even when non-deterministic reassignment is allowed ([38, Proposition 14]; see also Proposition 12.1.). As we demonstrate, this allows to show that T is functional if and only if its restriction to some finite set of data values is functional.

More precisely, we show that if some mismatch happens between outputs, the input and output words can be renamed with finitely many distinct data values while preserving the mismatch, by applying the renaming property on some well-chosen non-deterministic register automaton. Indeed, if we know a bound on the length before the mismatch, we can recognise traces that correspond to runs whose outputs mismatch before this bound.

[38]: Kaminski and Zeitlin (2010), ‘Finite-Memory Automata with Non-Deterministic Reassignment’

Proposition 12.6 Let T be a register transducer with k registers, and let $i \in \mathbb{N}$. We define

$$M_i^T = \left\{ \rho_1 \otimes \rho_2 \mid \begin{array}{l} \rho_1 \text{ and } \rho_2 \text{ are accepting runs of } T \\ \text{in}(\rho_1) = \text{in}(\rho_2) \text{ and} \\ |\text{out}(\rho_1) \wedge \text{out}(\rho_2)| \leq i \end{array} \right\} \quad (12.1)$$

Then, M_i^T is recognised by a non-deterministic register automaton with $2k + 2$ registers.

Proof. Let T be a register transducer. We know by Proposition 10.2 that $L_{\otimes}(T)$ is recognised by a non-deterministic register automaton with $2k$ registers. Checking that $\text{in}(\rho_1) = \text{in}(\rho_2)$ can be done with a single register: store d_i , wait for the separator $\$,$ and check that $e_i = d_i$.

Checking that the outputs of the run mismatch before their i -th data value is done as follows: initially, the automaton guesses a position $j \leq i$ such that $\text{out}(\rho_1)[j] \neq \text{out}(\rho_2)[j]$ and stores it in an i -bounded counter. Assume without loss of generality that $\text{out}(\rho_1)[j]$ is produced first. The automaton keeps track of the length of the prefixes of $\text{out}(\rho_1)$ and $\text{out}(\rho_2)$ which have been output so far using two i -bounded counters. At some point, it reaches the output position j on $\text{out}(\rho_1)$ and stores $\text{out}(\rho_1)[j]$ in a register. Then, it keeps simulating ρ_1 and ρ_2 in parallel, while keeping track of the length of the prefix of $\text{out}(\rho_2)$ that has been produced. Once it reaches the output position j in ρ_2 , it checks that, indeed, $\text{out}(\rho_1)[j] \neq \text{out}(\rho_2)[j]$, by comparing the value with the one that it stored in its register. Finally, it continues to simulate ρ_1 and ρ_2 while checking that $\text{in}(\rho_1) = \text{in}(\rho_2)$ (cf above). $\square$

When comparing two runs ρ_1 and ρ_2 over the same input $w \in \mathbb{D}^\omega$, we say that an output data value at position j is produced first in ρ_1 if it is output in ρ_1 on reading the k -th input data value, while it is output in ρ_2 on reading the l -th input data value for $l \geq k$.

By applying the renaming property to such an automaton, we get the following:

Proposition 12.7 Let T be a non-deterministic register transducer with k registers. Then, for all $X \subset_f \mathbb{D}$ of size $|X| \geq 2k + 5$ such that $\#, \$ \in X$, we have that T is functional if and only if $\llbracket T \rrbracket \cap (X^\omega \times X^\omega)$ is functional.

One needs $2k + 3$ data values to rename, plus the 2 data values $\#$ and $\$$, hence $2k + 5$. Note that $\$$ could actually be simulated by adding labels.

Proof. The left-to-right direction is trivial. Now, assume that T is not functional, and let $x, y, z \in \mathbb{D}^\omega$ such that $y \neq z$ and $(x, y), (x, z) \in \llbracket T \rrbracket$. Correspondingly, let ρ_1 (respectively ρ_2) be an accepting run of T on input x yielding output y (respectively z). Let $i = |y \wedge z|$. Consider the data language M_i^T of Equation 12.1. We have that $\rho_1 \otimes \rho_2 \in M_i^T$, so $M_i^T \neq \emptyset$. We know by the above Proposition 12.6 that M_i^T is recognised by a non-deterministic register automaton with $2k + 2$ registers. By applying Proposition 12.1 to M_i^T , we know that there exists some $w \in M_i^T \cap X^\omega$. By definition of M_i^T , $w = \rho'_1 \otimes \rho'_2$ for accepting runs ρ'_1 and ρ'_2 such that $\text{in}(\rho'_1) = \text{in}(\rho'_2) = x'$ for some $x' \in X^\omega$, which respectively yields $\text{out}(\rho'_1) = y'$ and $\text{out}(\rho'_2) = z'$ such that $|y' \wedge z'| \leq i$. In particular, y' and z' are such that $y' \neq z'$, so this means that $(x', y'), (x', z') \in \llbracket T \rrbracket \cap (X^\omega \times X^\omega)$ are such that $y' \neq z'$, which means that $\llbracket T \rrbracket \cap (X^\omega \times X^\omega)$ is not functional, which concludes the proof. $\square$

Along with Proposition 12.3 and Theorem 12.4, we get:

[141]: Exibard, Filiot, and Reynier (2020), 'On Computability of Data Word Functions Defined by Transducers'

Theorem 12.8 ([141, Corollary 10]) *The functionality problem for functions defined by non-deterministic register transducers over data domain $(\mathbb{D}, =, \#)$ is PSPACE-complete.*

Proof. Let T be a non-deterministic register transducer. By the above Proposition 12.7, the functionality problem of T reduces to that of $\llbracket T \rrbracket \cap (X \times X)^\omega$ for any X of size $2k + 5$. The choice of X does not matter, e.g. if $\mathbb{D} = \mathbb{N}$, one can take $X = \{\#, 1, \dots, 2k + 5\}$. Then, $\llbracket T \rrbracket \cap (X \times X)^\omega$ is recognised by a non-deterministic ω -transducer with exponentially many states which can be constructed on-the-fly (Proposition 12.3). Since functionality is in NLOGSPACE for non-deterministic (register-free) asynchronous transducers, it can be decided in polynomial space whether T is functional.

For the hardness, it is known that the emptiness problem for non-deterministic register automata is PSPACE-hard [28, Theorem 5.1]. It can easily be reduced to a functionality problem, by constructing a register transducer which is functional if and only if its domain is empty. For instance, given a register automaton A , build a transducer which recognises the relation $\text{id}_{\mathbb{D}} \cup R_A$, where, by distinguishing $\$ \neq \pounds \in \mathbb{D}$, $R_A = \{(w \cdot \$^\omega, w \cdot \pounds^\omega) \mid w \in L(A)\}$. $\text{id}_{\mathbb{D}}$ is trivially recognisable by a NRT, and one can easily build a NRT recognising R_A (recall also that NRT are closed under union, see Proposition 10.4). Then, $\text{id}_{\mathbb{D}} \cup R_A$ is functional if and only if $R_A = \emptyset$, if and only if $L(A) = \emptyset$. $\square$

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

Note that R_A is in general a relation, since the decomposition $w \cdot \$^\omega$ is not unique in general, if $\$$ is a data of w .

12.1.2. ω -Computability and Continuity

We already know by Theorem 11.2 that ω -computability implies continuity. To show the converse, thanks to Theorem 11.3, it suffices to show that the next-letter problem is decidable.

Proposition 12.9 *Let T be a non-deterministic register transducer over $(\mathbb{D}, =, \mathbb{C})$ recognising some function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$. The next-letter problem for f is decidable.*

In other words, given $u, v \in \mathbb{D}^*$ one can decide whether there exists some $d \in \mathbb{D}$ such that for all $y \in \mathbb{D}^\omega$ with $u \cdot y \in \text{dom}(f)$, we have $v \cdot d \leq f(u \cdot y)$, and exhibit such a data value d if it exists.

Proof. The proof goes in two steps.

The next-letter decision problem We first solve the next-letter decision problem. We reduce it to the emptiness problem of some non-deterministic register automaton. The automaton simulates T on inputs which start with u and whose corresponding output starts with v , and checks whether there are two subsequent runs whose output do not match on the first data value after v .

Formally, consider the data language

$$L_{\otimes, u, v}^T = \left\{ \rho_1 \otimes \rho_2 \left| \begin{array}{l} \rho_1 \text{ and } \rho_2 \text{ are accepting runs of } T \\ u \leq \text{in}(\rho_1), \text{in}(\rho_2), v \leq \text{out}(\rho_1), \text{out}(\rho_2), \\ \text{out}(\rho_1)[|v|] \neq \text{out}(\rho_2)[|v|] \end{array} \right. \right\}$$

We have that $L_{\otimes, u, v}^T = \emptyset$ if and only if there exists $d \in \mathbb{D}$ such that for all $y \in \mathbb{D}^\omega$ with $u \cdot y \in \text{dom}(f)$, we have $v \cdot d \leq f(u \cdot y)$. Indeed, if $L_{\otimes, u, v}^T \neq \emptyset$, it means there exists two accepting runs over respective inputs $u \cdot y_1$ and $u \cdot y_2 \in \text{dom}(f)$ with respective outputs $v \cdot d_1 \cdot z_1$ and $v \cdot d_2 \cdot z_2$, with $d_1 \neq d_2$, which contradict the existence of d . Conversely, if $L_{\otimes, u, v}^T = \emptyset$, then take $d = f(u \cdot y)[|v|]$ for some $y \in \mathbb{D}^\omega$ with $u \cdot y \in \text{dom}(f)$ if such an y exists; one can check that it satisfies the required property. If no such y exists, then any $d \in \mathbb{D}$ is suitable.

We already know that $L_{\otimes}(T)$ is recognisable by some non-deterministic register automaton over $(\mathbb{D}, =, \$)$ (with $2k$ registers). One can hard-code u and v by adding constants $\text{elem}(u) \cup \text{elem}(v)$ to the data domain. Checking that $u \leq \text{in}(\rho_1), \text{in}(\rho_2)$ and $v \leq \text{out}(\rho_1), \text{out}(\rho_2)$ is then easily done through tests $\star = u[i]$ and $\star = v[j]$, with one $|u|$ -bounded counter i and one $|v|$ -bounded counter j . Finally, testing that $\text{out}(\rho_1)[|v|] \neq \text{out}(\rho_2)[|v|]$ is done using a single additional register and a $|v|$ -bounded counter. Overall, $L_{\otimes, u, v}^T$ is recognised by an NRA (with $2k + 1$ registers) over data domain $(\mathbb{D}, =, \{\$ \} \sqcup \text{elem}(u) \cup \text{elem}(v))$. The emptiness problem is decidable over this data domain (Theorem 4.20), which concludes the first part of the proof.

Exhibiting d If the next-letter decision problem answers No, we are done. Assume it answers Yes: we know there exists $d \in \mathbb{D}$ such that for all $y \in \mathbb{D}^\omega$ with $u \cdot y \in \text{dom}(f)$, we have $v \cdot d \leq f(u \cdot y)$. As pointed out above, there are two cases.

If $u\mathbb{D}^\omega \cap \text{dom}(f) = \emptyset$, then take any $d \in \mathbb{D}$. Note that $u\mathbb{D}^\omega$ and $\text{dom}(f)$ are both recognisable by an NRA, so checking whether $u\mathbb{D}^\omega \cap \text{dom}(f) = \emptyset$ reduces to checking the emptiness of some non-deterministic register automaton, which is decidable (again by Theorem 4.20).

If $u\mathbb{D}^\omega \cap \text{dom}(f) \neq \emptyset$, then one can iteratively test all $d \in \mathbb{D}$; the fact that $\mathcal{D}$ is representable entails that the d can be enumerated. For each $d \in \mathbb{D}$, the algorithm checks emptiness of the following data language, which can be shown recognisable by a NRA with a similar reasoning as above:

$$L_{u, v, d}^T = \left\{ \text{tr}(\rho) \mid \begin{array}{l} \rho \text{ is an accepting run of } T \text{ such that} \\ u \leq \text{in}(\rho) \text{ and } v \cdot d \leq \text{out}(\rho) \end{array} \right\}$$

The procedure is guaranteed to terminate since we know such a d exists. $\square$

Along with Theorem 11.3, this yields:

Theorem 12.10 ([141, Theorem 18]) *ω -Computability and continuity coincide for functions defined by non-deterministic register transducers over $(\mathbb{D}, =, \mathbb{C})$.*

This equivalence also holds for the refined notions of Cauchy, uniform and m -computability (respectively, continuity).

12.1.3. Deciding Continuity

In this section, we provide an effective characterisation of non-continuity, and show how to decide it, and hence how to decide ω -computability.

Recall that NRA are closed under intersection, see Proposition 4.2

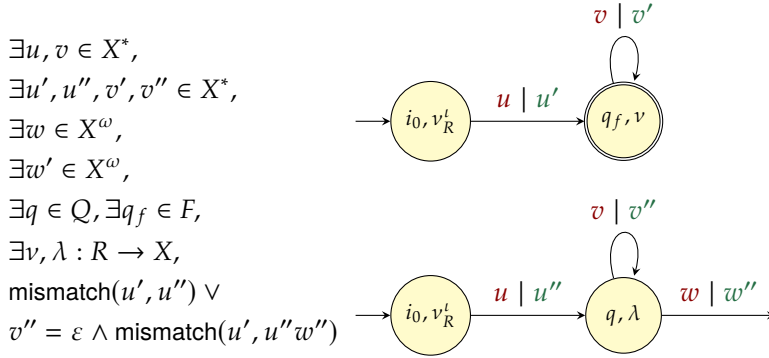
The reader might be surprised by the blatant inefficiency of the procedure we describe. Recall that we are only concerned about decidability here, since we only want to establish equivalence between ω -computability and continuity. A more efficient procedure would be to simulate T until it outputs the sought d . However, when guessing is involved, there are infinitely many distinct runs over a given input, so one has to abstract them in some way. This is done in a more general setting in the proof of Proposition 12.38.

[141]: Exibard, Filiot, and Reynier (2020), ‘On Computability of Data Word Functions Defined by Transducers’

In [51, Figure 2 and Lemma 11], continuity is characterised by a forbidden pattern. We extend this characterisation to the case of non-deterministic register transducers over $(\mathbb{D}, =)$, and show that this allows to reduce the decision of continuity to the finite alphabet case. The technique we use is very similar to the one for deciding functionality: we again show that a particular language of traces is recognisable by some non-deterministic register automaton, to which we apply the renaming property.

Theorem 12.11 ([141, Theorem 19]) *Let T be a functional NRT with k registers, and $X \subseteq \mathbb{D}$ of size $|X| \geq 2k + 3$. The following are equivalent:*

- (i) T is not continuous at some $x \in \mathbb{D}^\omega$
- (ii) T is not continuous at some $x \in X^\omega$
- (iii) T has the pattern of Figure 12.3.



[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

[141]: Exibard, Filiot, and Reynier (2020), ‘On Computability of Data Word Functions Defined by Transducers’

Figure 12.3.: A pattern characterising non-continuity of functions definable by a NRT over $(\mathbb{D}, =)$. Recall that $\text{mismatch}(u, v)$ is a macro expressing that u and v mismatch, i.e. $u \not\parallel v$.

Proof. The implication (ii) $\Rightarrow$ (i) is trivial. Note that (ii) $\Leftrightarrow$ (i) is sufficient for decidability, but (iii) is useful to prove (i) $\Rightarrow$ (ii). It also yields another explanation of the decision procedure, which actually consists in looking for this pattern under the hood, as it is the way to decide non-continuity in the data-free case [51, Figure 2 and Lemma 11]. We now move to (iii) $\Rightarrow$ (ii).

The pattern yields a discontinuity point Let T be a non-deterministic register transducer. Assume that T has the pattern of Figure 12.3 for some $X \subseteq \mathbb{D}$. Consider the sequence $(x_n)_{n \in \mathbb{N}}$ defined, for all $n \in \mathbb{N}$, by $x_n = u \cdot v^n \cdot w$. It converges to $x = u \cdot v^\omega$, and $f(x) = u' \cdot (v')^\omega$. For all $n \in \mathbb{N}$, we have $f(x_n) = u'' \cdot (v'')^n \cdot w''$. There are two cases:

- If $u' \not\parallel u''$, then for all $n \in \mathbb{N}$, $f(x_n) \wedge f(x) = u' \wedge u''$, so $(f(x_n))_{n \in \mathbb{N}}$ does not converge to $f(x)$.
- Otherwise, we have $v'' = \varepsilon$, so for all $n \in \mathbb{N}$ $f(x_n) = u \cdot w''$. Since in that case $u' \not\parallel u'' \cdot w''$, this again means that $(f(x_n))_{n \in \mathbb{N}}$ does not converge to $f(x) = u'(v')^\omega$.

In both cases, f is not continuous at $x = u \cdot v^\omega \in X^\omega$, and item (ii) holds.

The pattern is necessary for non-continuity We now move to the hardest implication, namely (i) $\Rightarrow$ (iii).

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

Outline To that end, we need to build two data words u and v which take their data values in X and which coincide on a sufficiently long prefix to allow for pumping. We use a similar proof technique as for Proposition 12.7: we show that a non-deterministic register automaton can recognise the data language of interleaved traces of runs whose inputs coincide on a sufficiently long prefix, while their respective outputs mismatch before a given position. We then apply the renaming property. Actually, we also ask that one run presents sufficiently many occurrences of a final state q_f , so that we can ensure the existence of a pair of configurations containing q_f which repeats in both runs.

On reading such u and v , the automaton behaves as an ω -automaton, since the number of distinct data values is finite (Proposition 12.2). By analysing the respective runs, we can, using pumping arguments, bound the position on which the mismatch appears in u , then show the existence of a synchronised loop over u and v after such position, allowing us to build the sought pattern.

Relabel over X Let us prove the result formally. Let T be a functional non-deterministic transducer with n states and k registers, which recognises a function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$. Assume that f is not continuous at some $x \in \mathbb{D}^\omega$ and let $X \subseteq \mathbb{D}$ such that $|X| \geq 2k + 3$. Let ρ be an accepting run of T yielding output $f(x)$, and let $q_f \in \text{Inf}(\text{states}(\rho)) \cap F$ be a final state which appears infinitely often in ρ . As f is not continuous at x , there exists some $i \geq 0$ such that for all $j \geq 0$, there exists $y \in \text{dom}(f)$ such that $|x \wedge y| \geq j$ but $|f(x) \wedge f(y)| \leq i$. Pick such an i . Then, take $K > |Q| \cdot |X|^k$, and define $m = K(i + 1) + K^2(i + 1) + K^2$. Finally, choose $j \in \mathbb{N}$ such that $\text{states}(\rho)[:j]$ contains at least m occurrences of q_f . Consider the data language

$$L_{i,j,m}^T = \left\{ \rho_1 \otimes \rho_2 \left| \begin{array}{l} \rho_1 \text{ and } \rho_2 \text{ are accepting runs of } T \\ |\text{in}(\rho_1) \wedge \text{in}(\rho_2)| \geq j, |\text{out}(\rho_1) \wedge \text{out}(\rho_2)| \leq i \text{ and} \\ q_f \text{ occurs at least } m \text{ times in } \text{states}(\rho_1)[:j] \end{array} \right. \right\}$$

By Proposition 10.2, we know the language of interleaved traces of accepting runs of T is recognised by a non-deterministic register automaton with $2k$ registers. Checking that $|\text{in}(\rho_1) \wedge \text{in}(\rho_2)| \geq j$ can be done with a single additional register and a j -bounded counter, by successively storing each data value $\text{in}(\rho_1)[l]$ for $0 \leq l \leq j$, going to the next $\$,$ and checking that indeed $\text{in}(\rho_1)[l] = \text{in}(\rho_2)[l]$. Testing whether $|\text{out}(\rho_1) \wedge \text{out}(\rho_2)| \leq i$ is done with two i -bounded counters and another register: the automaton guesses the position of the mismatch, stores the corresponding position in its state using a third i -bounded counter along with its data value, goes to the same position in the output of the other run and checks that they indeed mismatch. Finally, checking that there are at least m occurrences of q_f in $\text{states}(\rho_1)[:j]$ is a regular property, necessitating yet another j -bounded counter and an m -bounded counter. Overall, $L_{i,j,m}^T$ is recognised by a non-deterministic register automaton with $2k + 2$ registers.

It remains to show that $L_{i,j,m}^T \neq \emptyset$, so that we can apply the renaming property and pump to get the sought pattern. Since f is not continuous at x , there exists some $y \in \text{dom}(f)$ such that $|x \wedge y| \geq j$ but $|f(x) \wedge f(y)| \leq i$. Pick such a y , and let ρ_2 be an accepting run of T over y yielding output

Note that the constants involved might be incredibly large, as we have no bound on i , nor on j . Thus, the NRA we build can have a myriad of states. However, we do not care about that, as the renaming property only depends on the number of registers. Evaluating the number of states is an easy exercise, which may (or may not) bring some fun to the reader.

$f(y)$. We then have that $\rho_1 \otimes \rho_2 \in L_{i,j,m}^T$. By the renaming property (Proposition 12.1) we get two runs ρ'_1 and ρ'_2 such that $\rho'_1 \otimes \rho'_2 \in L_{i,j,m}^T \cap X^\omega$. Thus, they are both accepting, and their respective inputs $x' = \text{in}(\rho'_1) \in X^\omega$ and $x'' = \text{in}(\rho'_2) \in X^\omega$ are such that $|x' \wedge x''| \geq j$ but $|f(x') \wedge f(x'')| \leq i$. Moreover, $\text{states}(\rho'_1)[j]$ contains at least m occurrences of q_f . Note that we cannot stop here and right away conclude to non-continuity x' , since x' and x'' depend on j . We thus need pumping arguments to exhibit the pattern, which is a witness of non-continuity at some x''' .

Decompose the runs and pump Write ρ'_1 as $\rho'_1 = \lambda^{\text{pr}} \cdot \lambda^{\text{np}} \cdot \lambda^{\text{mm}} \cdot \lambda^{\text{sf}}$, where (this decomposition is depicted in Figure 12.4):

- λ^{pr} is the partial run containing the first $K(i+1)$ occurrences of q_f and ending with q_f . We write $\lambda^{\text{pr}} = (q^t, v_R^t) \xrightarrow{u^{\text{pr}} | v^{\text{pr}}} (q_f, v^{\text{pr}})$, where $u^{\text{pr}} = \text{in}(\lambda^{\text{pr}})$ and $v^{\text{pr}} = \text{out}(\lambda^{\text{pr}})$
- λ^{np} is the subsequent partial run containing $K^2(i+1)$ occurrences of q_f and ending with q_f . We write $\lambda^{\text{np}} = (q_f, v^{\text{pr}}) \xrightarrow{u^{\text{np}} | v^{\text{np}}} (q_f, v^{\text{np}})$, where $u^{\text{np}} = \text{in}(\lambda^{\text{np}})$ and $v^{\text{np}} = \text{out}(\lambda^{\text{np}})$
- λ^{mm} is the following partial run containing K^2 occurrences of q_f and ending with q_f . We write $\lambda^{\text{mm}} = (q_f, v^{\text{np}}) \xrightarrow{u^{\text{mm}} | v^{\text{mm}}} (q_f, v^{\text{mm}})$, where $u^{\text{mm}} = \text{in}(\lambda^{\text{mm}})$ and $v^{\text{mm}} = \text{out}(\lambda^{\text{mm}})$
- λ^{sf} is the remaining run. We write $\lambda^{\text{sf}} = (q_f, v^{\text{mm}}) \xrightarrow{t_1 | t'_1}$, where $t_1 = \text{in}(\lambda^{\text{sf}})$ and $t'_1 = \text{out}(\lambda^{\text{sf}})$.

We pararelyly decompose (cf again Figure 12.4) $\rho'_2 = \tau^{\text{pr}} \cdot \tau^{\text{np}} \cdot \tau^{\text{mm}} \cdot \tau^{\text{sf}}$, where $|\lambda^*| = |\tau^*|$ for $* \in \{\text{pr}, \text{np}, \text{mm}\}$. Note that this condition implies that $\text{in}(\rho'_1) = \text{in}(\rho'_2)$ (again for $* \in \{\text{pr}, \text{np}, \text{mm}\}$), since we know that $|\text{in}(\rho'_1) \wedge \text{in}(\rho'_2)| \geq j$, so the inputs match at least until the m -th occurrence of q_f . Thus, we can write:

- $\tau^{\text{pr}} = (p^t, v_R^t) \xrightarrow{u^{\text{pr}} | w^{\text{pr}}} (p, \mu^{\text{pr}})$
- $\tau^{\text{np}} = (p^{\text{pr}}, \mu^{\text{pr}}) \xrightarrow{u^{\text{np}} | w^{\text{np}}} (p^{\text{np}}, \mu^{\text{np}})$
- $\tau^{\text{mm}} = (p^{\text{np}}, \mu^{\text{np}}) \xrightarrow{u^{\text{mm}} | w^{\text{mm}}} (p^{\text{mm}}, \mu^{\text{mm}})$
- $\tau^{\text{sf}} = (p^{\text{mm}}, \mu^{\text{mm}}) \xrightarrow{t_2 | t'_2}$

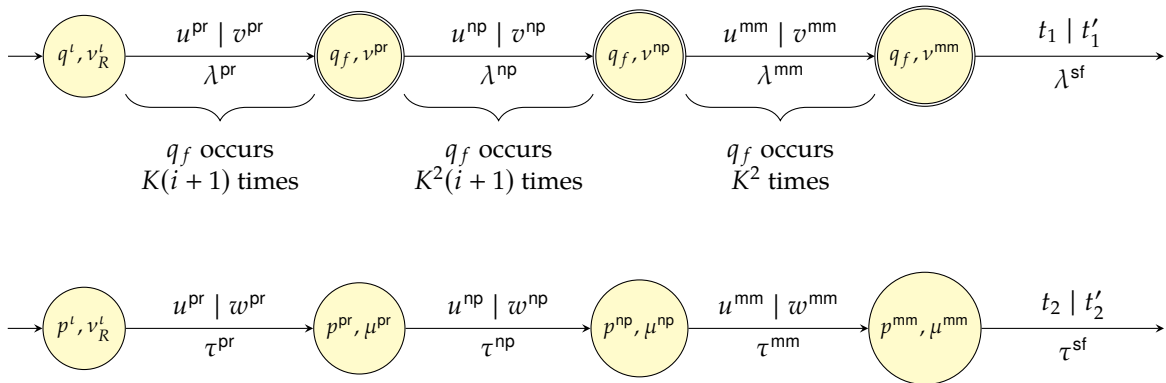


Figure 12.4.: Decomposition of the runs ρ'_1 and ρ'_2 of T .

Still, the above reasoning suffices in the case of uniform continuity, since then the witnessing x' and x'' can depend on j , due to uniformity: for each $i \geq 0$, one just need to find x' and x'' such that $|x' \wedge x''| \geq i$ but $|f(x') \wedge f(x'')| \leq j$.

We removed the primes and indices in the decomposition to make the notation less cluttered. pr stands for 'prefix', np for 'non-productive' (it is called this way because this is where we pump in the case $v'' = \varepsilon$), mm for 'mismatch' and sf for 'suffix'. Note also that we do not need to assume that the partial runs decomposing ρ'_1 end in q_f , but this is a way to make the decomposition unambiguous.

Note that λ^{sf} is the only infinite sequence, the others are partial runs. Similarly, w and w' are data ω -words, while the others are finite data words.

Repeating configurations We can assume that all valuations take their values in X , even when non-deterministic reassignment is allowed (cf Proposition 12.1). We can see from Figure 12.4 that we are getting closer to the pattern. First, let us observe that in ρ'_1 , any sequence of transitions which contains more than $|Q| \cdot |X|^k$ occurrences of q_f contains a repeating valuation associated with q_f , by the pigeonhole principle. In other words, there exists some valuation $v : R \rightarrow X$ such that (q_f, v) appears at least twice. As a consequence, there exists at least one productive transition among those transitions. Otherwise, it means that there exists a partial run $\lambda = (q_f, v) \xrightarrow[T]{w|\varepsilon} (q_f, v)$ for some finite input data word $w \in \mathbb{D}^*$, which can be pumped to yield an accepting run with finite output, which contradicts the assumption that register transducers only produce infinite outputs. For the sake of completeness, we make the argument more precise: since (q_f, v) is a configuration of ρ'_1 , it means in particular that it is accessible, i.e. there exist $w_0, w'_0 \in \mathbb{D}^*$ and a partial run π such that $(q^t, v_R^t) \xrightarrow[T]{w_0|w'_0} (q_f, v)$. By pumping λ , we get that $\pi \cdot \lambda^\omega = (q^t, v_R^t) \xrightarrow[T]{w_0|w'_0} (q_f, v) \xrightarrow[T]{w|\varepsilon} (q_f, v) \xrightarrow[T]{w|\varepsilon} \dots$ is a run over input $w_0 \cdot w^\omega$ which yields finite output $w'_0 \in \mathbb{D}^*$, which contradicts Assumption 10.3.

A *productive transition* is a transition whose output is $o \neq \varepsilon$.

By a slight abuse of notation, we write $(q_f, v) \xrightarrow[T]{w|\varepsilon} (q_f, v)$ to say that we call λ (for 'loop') the partial run which loops over (q_f, v) .

Locating the mismatch The first consequence of this is that $|v^{\text{pr}}| > i$, since λ^{pr} contains $K(i+1)$ occurrences of q_f , and $K > |Q| \cdot |X|^k$. This means that the position of the mismatch between $f(x')$ and $f(x'')$ lies in v^{pr} . Now, the argument is a bit technical, due to the necessity to decompose the runs further, but as we demonstrate, the number of occurrences of q_f in each part of ρ'_1 allows to pump so as to exhibit the pattern of Figure 12.3 on page 235. There are two cases, depending on whether the mismatch occurs early or late. In the latter case, we then show that it implies the existence of unproductive transitions (this corresponds to the right-hand-side of the disjunction in the pattern, where $v'' = \varepsilon$). We formalise the argument:

- Assume the mismatch occurs early in ρ'_2 , i.e. $v^{\text{pr}} \cdot v^{\text{np}} \not\leq w^{\text{pr}} \cdot w^{\text{np}}$. Since there are $K^2 > |Q|^2 \cdot |X|^{2k}$ occurrences of q_f in λ^{mm} , there is a *pair* of configurations which contains q_f and repeats synchronously. In other words, we can write

$$\lambda^{\text{mm}} = (q_f, v^{\text{np}}) \xrightarrow[t_0|t'_0]{} (q_f, v) \xrightarrow[v|v']{} (q_f, v) \xrightarrow[t_1|t'_1]{} (q_f, v^{\text{mm}})$$

and simultaneously

$$\tau^{\text{mm}} = (p^{\text{np}}, \mu^{\text{np}}) \xrightarrow[t_0|t''_0]{} (p, \mu) \xrightarrow[v|v'']{} (p, \mu) \xrightarrow[t_1|t''_1]{} (p^{\text{mm}}, \mu^{\text{mm}})$$

By taking $u = u^{\text{pr}} \cdot u^{\text{np}}$, $u' = v^{\text{pr}} \cdot v^{\text{np}}$, $u'' = w^{\text{pr}} \cdot w^{\text{np}}$ and $w = t_1 \cdot t_2$, $w'' = t'_1 \cdot t'_2$, we get the pattern of Figure 12.3.

- Otherwise, the mismatch occurs late in ρ'_2 , i.e. $w^{\text{pr}} \cdot w^{\text{np}} \leq v^{\text{pr}} \cdot v^{\text{np}}$. This implies that $|w^{\text{pr}} \cdot w^{\text{np}}| \leq i$, so, in particular, $|w^{\text{np}}| \leq i$. We know there are $K^2(i+1)$ occurrences of q_f in λ^{np} , so, among the pairs of configurations containing q_f that repeat, there is at least

one which is improductive. In other words, we can write τ^{np} as

$$(p^{\text{np}}, \mu^{\text{np}}) \xrightarrow{t_0 | t_0''} (q, \mu) \xrightarrow{v | \varepsilon} (q, \mu) \xrightarrow{t_1 | t_1''} (p^{\text{mm}}, \mu^{\text{mm}})$$

and synchronously decompose λ^{np} as

$$(q_f, v^{\text{np}}) \xrightarrow{t_0 | t_0'} (q_f, v) \xrightarrow{v | v'} (q_f, v) \xrightarrow{t_1 | t_1'} (q_f, v^{\text{mm}})$$

Take $u = u^{\text{pr}} \cdot t_0$, $u' = v^{\text{pr}} \cdot t_0'$, $u'' = w^{\text{pr}} \cdot t_0''$, $w = t_1 \cdot u^{\text{mm}} \cdot t_2$ and $w'' = t_1'' \cdot w^{\text{mm}} \cdot t_2'$. Since $|u'| > i$, we know that $u' \not\parallel u'' \cdot w''$, as $|f(x') \wedge f(x'')| \leq i$, so we exhibited the pattern of Figure 12.3.

This concludes the proof. $\square$

The above characterisation allows to decide continuity for functions defined by non-deterministic register transducers. We can further pinpoint its complexity:

Theorem 12.12 ([141, Corollary 20]) *The continuity problem for functions defined by non-deterministic register transducers is PSPACE-complete.*

Proof. Let T be a non-deterministic register transducer. We know by Theorem 12.11 that T is not continuous if and only if it is not continuous over X^ω for any $X \subseteq \mathbb{D}$ of size $|X| \geq 2k + 5$. Pick some $X \subset_f \mathbb{D}$ of size $2k + 3$. By Proposition 12.3, we know that $T \cap (X^\omega \times X^\omega)$ is recognised by a non-deterministic ω -transducer with exponentially many states, which can moreover be constructed on-the-fly. Since continuity is in NLogSPACE for this model [51, Theorem 12], this yields an algorithm for deciding continuity of T which runs in polynomial space.

To show that the problem is PSPACE-hard, we again reduce from the emptiness problem for non-deterministic register automata over *finite data words*, which is PSPACE-complete [28, Theorem 5.1]. Let A be such a register automaton, over $(\mathbb{D}, =)$. We construct a transducer T which is continuous if and only if $L(A) = \emptyset$, iff the domain of T is empty. Let f be a function realised by some NRT H which is not continuous at some $x \in \mathbb{D}^\omega$ (take, e.g., the function f_{again} of Example 11.2). Let $\$ \notin \mathbb{D}$ be a fresh symbol, and define the function g as follows: it takes as input a data word of the form $w\$w'$, where $w \in L(A)$, and outputs $w\$f(w')$. Then, g is continuous if and only if $L(A) = \emptyset$. Indeed, if $L(A) = \emptyset$, then $\text{dom}(g) = \emptyset$ and g is trivially continuous. Conversely, if $L(A) \neq \emptyset$, then g is not continuous at $w\$x$ for any $w \in L(A)$. It remains to show that g is recognisable by a non-deterministic register transducer. This transducer starts by simulating A on its input, and simultaneously outputs it. When it reads $\$$, if it was in some accepting state of A , it branches to some initial state of H and proceeds executing H . This concludes the proof. $\square$

Remark 12.1 (Uniform continuity) Over finite alphabets, uniform continuity is also characterised by a forbidden pattern [51, Figure 3 and Lemma 11], for any set X of size $|X| \geq 2k + 3$. The above reasoning can be adapted to show that uniform continuity is characterised by the same forbidden pattern over $(\mathbb{D}, =)$ (Figure 12.5 on the following page). It is thus decidable, and more precisely PSPACE-complete. We do not formally show it here, as it does not bring much to the presentation and can be

[141]: Exibard, Filiot, and Reynier (2020), ‘On Computability of Data Word Functions Defined by Transducers’

Note that the result can equivalently be obtained by directly deciding the pattern of Figure 12.3 with the pattern logic of [147]: Filiot, Mazzocchi, and Raskin (2018), ‘A Pattern Logic for Automata with Outputs’.

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

deducted from the study of the oligomorphic case (see Section 12.2, and in particular Theorem 12.41). Besides, as pointed out in the proof of Theorem 12.11, one does not actually need to exhibit the pattern to get decidability, as we can directly establish that non-uniform continuity at some $x \in \mathbb{D}^\omega$ is equivalent with non-uniform continuity at some $x \in X^\omega$ for any X of size $|X| \geq 2k + 3$. It then suffices to reduce to the data-free case, as is done in the above proof.

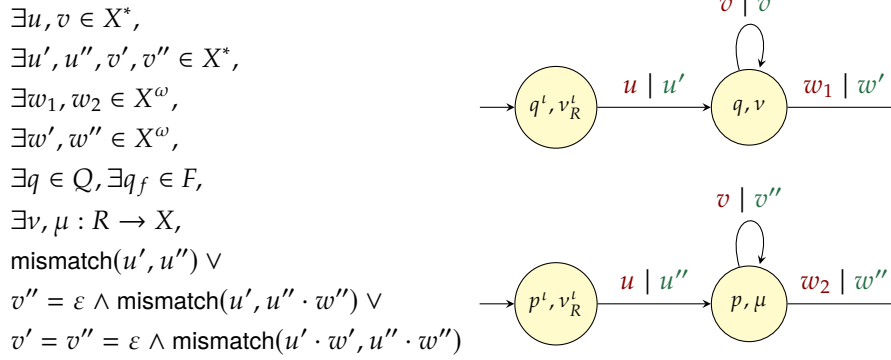


Figure 12.5.: A pattern characterising non-uniform continuity of functions definable by a NRT over $(\mathbb{D}, =)$.

Finally, note that, as in the finite alphabet case, uniform continuity and Cauchy continuity coincide over $(\mathbb{D}, =)$ by Theorem 12.29.

The proof of Theorem 12.29 is done in the more general oligomorphic setting, but the reader who is only interested in the case of $(\mathbb{D}, =)$ should be able to derive from it a direct proof for this particular case.

12.2. The Case of Oligomorphic Data Domains

As pointed out in Property 4.33, the renaming property does not hold anymore over $(\mathbb{Q}, <, 0)$, so the proof techniques of Section 12.1 do not apply. However, as witnessed by our study of register automata over $(\mathbb{Q}, <, 0)$ (Section 4.6), this data domain presents some form of finiteness: its k -tuples have finitely many distinct types. These types can thus be stored in the states, and this allows to show that in $(\mathbb{Q}, <, 0)$, the projection over labels of non-deterministic register automata is ω -regular (Proposition 4.35). It also makes it possible to turn them into locally concretisable devices (Proposition 4.37). As we demonstrate, this property (as well as reasonable computability assumptions) is sufficient for the decidability of functionality, continuity and its refined notions. It also yields decidability of the next-letter problem, which entails the decidability of ω -computability (and, again, its refined notions).

Since this part of our study has a more theoretical aim, we climb one level of abstraction and study the class of data domains whose k -uples have finitely many types. To that end, we shift to the setting of nominal sets, where this property bears the name of oligomorphicity. This class of domains contains a wide range of data domains, and in particular $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$ (cf Example 12.3). Distinguishing constants preserves oligomorphicity, so this also encompasses $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, \mathbb{C}')$ for arbitrary finite sets $\mathbb{C}$ and $\mathbb{C}'$ of constant symbols which are interpreted with data values in $\mathbb{D}$ (respectively $\mathbb{Q}$), see Proposition 12.15. More generally, any data domain which admits quantifier elimination falls in this study, as types then correspond to quantifier-free formulas. Note

The reader who is less mathematically inclined will find some remarks which help to derive proofs for the particular case of $(\mathbb{Q}, <)$ from the general setting.

that, on the contrary, $(\mathbb{N}, <, 0)$ is not oligomorphic, and is the topic of the dedicated Section 12.3.

As was already pointed out earlier, the languages recognised by register automata are closed under automorphisms (Proposition 4.10 in Section 4.4.6). The proofs are easily adapted to show that this is also the case for register transducers, even in the presence of non-deterministic reassignment (see Proposition 12.13). In nominal sets, this closure property bears the name of equivariance, and plays a central role. In [25], the authors established an elegant correspondence between nominal sets and register automata; the study has been furthered in particular in [26]. We lean on this work to develop our argument. In the following, we recall what an automorphism is, and subsequently define the notions of orbit. As we will see, it is closely related to the notion of type (see Proposition 12.17). Note that what follows is by no means a comprehensive introduction to the theory of nominal sets and its links with automata; we refer to [25, 26] for a more thorough presentation.

Let us recall the notion of automorphism.

Definition 4.18 (Automorphism of a data domain) Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ be a data domain. A function $\mu : \mathbb{D} \rightarrow \mathbb{D}$ is an *automorphism* of $\mathcal{D}$ if it is bijective and preserves relations and constants, i.e.:

- for all constants $c \in \mathcal{C}$, $\mu(c) = c$
- for all relation $P \in \mathcal{R}$ of arity k and for all tuples $(d_1, \dots, d_k) \in \mathbb{D}^k$, we have $(d_1, \dots, d_k) \in P$ if and only if $(\mu(d_1), \dots, \mu(d_k)) \in P$.

The set of automorphisms of $\mathcal{D}$ is denoted $\text{Aut}(\mathcal{D})$. Note that it has a group structure for the composition operation.

Automorphisms are extended to labelled data in the obvious way: $\mu(\sigma, d) = (\sigma, \mu(d))$ for all $(\sigma, d) \in \Sigma \times \mathbb{D}$.

Let us show that runs of a register automaton are closed under the application of some automorphism. (see also Proposition 4.10). In the setting of nominal sets, this property is called equivariance:

Definition 12.2 Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ be a data domain. A set $X \subseteq \mathbb{D}$ is *equivariant* if it is closed under automorphisms, i.e. $\mu(X) \subseteq X$ for all morphisms $\mu \in \text{Aut}(\mathcal{D})$. This property extends to sets that can be obtained from $\mathbb{D}$ using product, sum, powerset, Kleene star, etc by accordingly extending morphisms through pointwise application.

Proposition 12.13 Let A be a register automaton over some data domain $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ (with non-deterministic reassignment).

For a run ρ of A and a morphism $\mu \in \text{Aut}(\mathcal{D})$, we define $\mu(\rho)$ by applying it pointwise to its configurations: $\text{configs}(\mu(\rho)) = \mu(C_0)\mu(C_1) \dots$, and $\text{trans}(\mu(\rho)) = \text{trans}(\rho)$.

Then, for all runs ρ of A over some data word $w \in \mathbb{D}^\omega$, we have that $\mu(\rho)$ is a run of A over $\mu(w)$. Moreover, if ρ is initial (respectively, final, accepting), then so is $\mu(\rho)$.

The result extends to partial runs in the expected way. It also holds over transducers, by applying μ to the input and output word.

[25]: Bojańczyk, Klin, and Lasota (2014), ‘Automata theory in nominal sets’

[26]: Bojańczyk (2019), *Atom Book*

For simplicity, we confuse a symbol of the signature with its interpretation.

Composition of functions is associative, and it is routine to check that the composition of two automorphisms again preserves relations and constants. The neutral element of $\text{Aut}(\mathcal{D})$ is the identity function $\text{id}_{\mathbb{D}}$, and the inverse of an automorphism is its inverse function.

Applying a morphism to a configuration consists in applying it to its valuation: for all $(q, v) \in \text{Configs}(A)$, $\mu(q, v) = (q, \mu(v))$, with $\mu(v) : r \in R \mapsto \mu(v(r))$.

$\mu(w)$ is the pointwise application of μ to w , i.e. for all $i \in \mathbb{N}$, $\mu(w)[i] = \mu(w[i])$.

Proof. We prove the result over a single transition, the result then follows by a direct induction. The proof essentially consists in observing that the successor relation over configurations is invariant under applying an automorphism.

We treat the case of non-deterministic register transducers; the case of register automata is obtained by ignoring the outputs. Let T be a register transducer over data domain $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathbb{C})$. Let $(p, \nu), (q, \lambda) \in \text{Configs}(T)$, and assume there exists $t = p \xrightarrow{\phi, \text{regOp}, \nu} q \in \Delta$ and $d \in \mathbb{D}$, $w \in \mathbb{D}^*$ such that $C \xrightarrow[T]{d|w} D$. By definition, this means that $\nu, d \models \phi$, that λ is updated as follows

- for all $r \in R$, if $\text{regOp}(r) = \text{keep}$, then $\lambda(r) = \nu(r)$
- for all $r \in R$, if $\text{regOp}(r) = \text{set}$, then $\lambda(r) = d$
- for all $r \in R$, if $\text{regOp}(r) = \text{guess}$, then $\lambda(r) \in \mathbb{D}$ can take any data value

and that $w = \lambda(v)$.

Since μ preserves the predicates and constants of $\mathcal{D}$, we get that $\mu(\nu), \mu(d) \models \phi$, as $\nu, d \models \phi$. Moreover, $\mu(\lambda)$ indeed corresponds to some update of $\mu(\nu)$ according to regOp . Finally, $\mu(w) = \mu(\lambda(v)) = (\mu(\lambda))(v)$.

Overall, this means that $\mu(C) \xrightarrow[T]{\mu(d)|\mu(w)} \mu(D)$.

Finally, the character of being initial, final, accepting is preserved, since applying a morphism to a configuration does not affect its state.

Note that by considering μ^{-1} , we could even get the converse of Proposition 12.13, i.e. that for any sequence $\rho \in (\text{Configs}(T)\Delta)^\infty$, ρ is a (partial) run of T if and only if $\mu(\rho)$ is a (partial) run of T . $\square$

Orbits

In this study, we are interested in data domains which have a lot of symmetry. The reader can refer to the notion of data symmetry in [25, Part 1] for a formal definition. Oligomorphic data domains are data domains which are characterised by their group of automorphisms. For instance $(\mathbb{D}, =)$ is characterised by the group of (unrestricted) bijections over $\mathbb{D}$, as the only predicate that is preserved by all bijections is equality. Adding a finite number of constants then corresponds to restricting to bijections that preserve those elements. Then, $(\mathbb{Q}, <)$ is characterised by the group of increasing bijections, which preserve the dense order and equality. On the contrary, $(\mathbb{N}, <)$ does not fall in this category, as the only bijection which preserve the discrete order is the identity. Yet, a dedicated study yields results that are similar to the oligomorphic case, although the proof is more involved. The key property that we use is that $(\mathbb{N}, <)$ is a substructure of $(\mathbb{Q}, <)$, which is oligomorphic. Finally, $(\mathbb{Z}, +)$ is not suitable for our purpose: already, emptiness of register automata over this domain is undecidable, as they can simulate Minsky machines (cf L_M in Example 4.1).

We now introduce the notion of orbit, which characterises the structure of a data domain.

We say ‘some update’ because the updated valuation is not unique in presence of non-deterministic reassignment.

[25]: Bojańczyk, Klin, and Lasota (2014), ‘Automata theory in nominal sets’

L_M is defined over $(\mathbb{N}, <, 0)$, but 0 can be simulated by assimilating it with the first data value that is read.

Definition 12.3 (Orbit) Let $r \in \mathbb{N}$. For $\mathbf{d} \in \mathbb{D}^r$, the *orbit* of $\mathbf{d}$ under the action of $\text{Aut}(\mathcal{D})$ is the set $\text{orbit}(\mathbf{d}) = \{\mu(\mathbf{d}) \mid \mu \in \text{Aut}(\mathcal{D})\}$.

Example 12.2 In $(\mathbb{D}, =)$, all 1-uple have the same orbit $\mathbb{D}$ under the action of $\text{Aut}((\mathbb{D}, =))$, which is the set of all bijections. 2-uple $(d_1, d_2) \in \mathbb{D}^2$ are of two types: either $d_1 = d_2$, in which case the orbit of (d_1, d_2) is the diagonal $\{(d, d) \mid d \in \mathbb{D}\}$, or $d_1 \neq d_2$ and the corresponding orbit is the complement of the diagonal $\{(d, e) \mid d \neq e\}$. More generally, one can check that the orbit of a tuple is characterised by the equalities between its elements (more on that later; see also [25, Lemma 6.1]).

Similarly, over $(\mathbb{Q}, <)$, the orbit of a pair (d_1, d_2) (under the action of $\text{Aut}((\mathbb{Q}, <))$, which is the set of all increasing bijections) depends on whether $d_1 < d_2$ (it is then $\{(d, e) \mid d < e\}$), $d_1 = d_2$ or $d_1 > d_2$.

On the other hand, consider $(\mathbb{Z}, <)$. Automorphisms of $(\mathbb{Z}, <)$ are exactly translations $t_c : n \mapsto n + c$ for some $c \in \mathbb{Z}$. Thus, the orbit of a pair $(m, n) \in (\mathbb{Z}^2)$ depends on the distance between m and n , as it is preserved by applying a translation. Thus, the orbit of (m, n) is $\{(k, l) \mid k - l = m - n\}$.

Oligomorphicity

We are now ready to define the notion of oligomorphicity (see also [26, Chapter 3]), which separates $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$ from $(\mathbb{N}, <)$ and $(\mathbb{Z}, <)$ (and, a fortiori, $(\mathbb{Z}, +)$).

Definition 12.4 (Oligomorphicity) A data domain $\mathcal{D}$ is *oligomorphic* if for all $k \in \mathbb{N}$, the set $\mathbb{D}^k$ has finitely many orbits under the action of $\text{Aut}(\mathcal{D})$.

When $\mathcal{D}$ is oligomorphic, we denote $\mathcal{R}_{\mathcal{D}} : \mathbb{N} \rightarrow \mathbb{N}$ its *Ryll-Nardzewski function*, which maps $k \in \mathbb{N}$ to the number of orbits of k -tuples of data values.

Example 12.3 As explained in Example 12.2, data domains with equality only $(\mathbb{D}, =)$ are oligomorphic. Indeed, the orbit of a k -uple is characterised by the equalities between its components. It is thus characterised by an equivalence relation over k elements, so the Ryll-Nardzewski function of $(\mathbb{D}, =)$ maps k to the k -th Bell number B_k .

Similarly, $(\mathbb{Q}, <)$ is oligomorphic, as the orbit of a k -uple $\mathbf{x}$ is characterised by the set of relations $x_i \bowtie x_j$ for all $0 \leq i, j \leq |x|$, where $\bowtie$ is selected in $\{<, >, =\}$. It can thus be represented by a function $X \times X \rightarrow \{<, >, =\}$, where X is a set of variables of size k . Overall, $\mathcal{R}_{(\mathbb{Q}, <)}$ is bounded by 2^{k^2} .

This is also the case of $(1(0 + 1)^*, \otimes)$ which consists in bit vectors, where $\otimes$ denotes the bitwise xor operation. Indeed, it forms a vector space, and an n -tuple is either linearly independent or not. A tuple is then characterised by the linear equations over n variables that it satisfies, and there are exponentially many.

Oligomorphic data domains can be thought of as ‘almost finite’, in the sense that in many cases, it suffices to consider the finitely many distinct types of orbits, instead of the data values themselves.

As a final note, recall that $(\mathbb{N}, <)$ is not oligomorphic, as explained above.

Remark that the adding finitely many constants to the data domain preserves oligomorphicity.

The use of the term ‘type’ is not an accident. As we will see, the notions of orbits and types are tightly related (Proposition 12.17).

[25]: Bojańczyk, Klin, and Lasota (2014), ‘Automata theory in nominal sets’

[26]: Bojańczyk (2019), *Atom Book*

We omit $\mathcal{D}$ and simply write $\mathcal{R}$ when $\mathcal{D}$ is clear from the context.

Proposition 12.14 Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ be an oligomorphic data domain. Let K be a set of fresh data values, i.e. $\mathbb{D} \cap K = \emptyset$.

We define the addition of K to data domain as follows: $\mathcal{D} \sqcup K = (\mathbb{D} \sqcup K, \mathcal{R}, \mathcal{C} \sqcup \mathcal{C}')$, where $\mathcal{C}'$ is some set of fresh constant symbols that bijectively represents K . Then, $\mathcal{D} \sqcup K$ is oligomorphic. Moreover, their Ryll-Nardzewski functions are equal.

Formally, $\mathcal{C}'$ bijectively represents K if $|\mathcal{C}'| = |K|$ and each element of K is uniquely represented by some element of $\mathcal{C}'$. In practice, we confuse K with $\mathcal{C}'$.

Proof. Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ be an oligomorphic data domain. We show how to add one constant, the result is then obtained by induction. Let $c_0 \notin \mathcal{C}$ be some fresh constant symbol. The result follows from the definition of oligomorphicity, we include the proof for completeness. Assume that c_0 interpreted as some fresh data value $\$ \notin \mathbb{D}$. The automorphism group $\text{Aut}(\mathcal{D} \sqcup \{\$\})$ is in bijection with $\text{Aut}(\mathcal{D})$, as the former is obtained by uniquely extending all $\mu \in \text{Aut}(\mathcal{D})$ by setting $\mu(\$) = \$$. Consequently, for all $k \in \mathbb{N}$, $\mathbb{D}^k$ has finitely many orbits under the action of $\text{Aut}(\mathcal{D})$ if and only if $(\mathbb{D} \sqcup \{\$\})^k$ has finitely many under the action of $\text{Aut}(\mathcal{D} \sqcup \{\$\})$, and the number of distinct orbits is the same. $\square$

Remark 12.2 This easy result entails that our assumption of the existence of some special constant $\#$ that we use to initialise the registers is again benign for oligomorphic data domains (Assumption 10.1). This is also the case when we suppose that there is a value that is used as a separator (Assumption 10.5).

We now show that it is also possible to distinguish some finite set of constants, i.e. to add constant symbols that are interpreted as data values that *already belong* to the data domain. The difference lies in the fact that the predicates of the domain act on these constants, e.g. in $(\mathbb{Q}, <, \{0, 1\})$, we have $0 < 1$. Contrary to the above, the Ryll-Nardzewski function is in general affected by this distinction process.

Proposition 12.15 Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ be an oligomorphic data domain. Let $K \subset_f \mathbb{D}$ be a finite set of distinguished data values.

We define the distinction of K in data domain as follows: $\mathcal{D}^K = (\mathbb{D}, \mathcal{R}, \mathcal{C} \sqcup \mathcal{C}')$, where $\mathcal{C}'$ is some finite set of fresh constant symbols that bijectively represents K . Then, $\mathcal{D}^K$ is oligomorphic. We have, for all $n \in \mathbb{N}$, $\mathcal{R}_{\mathcal{D}^K}(n) \leq \mathcal{R}_{\mathcal{D}}(n + |K|)$.

Proof. This result is slightly less immediate, since the number of orbits increases in general. We again show how to distinguish a single constant, the result directly follows by induction on the number of distinguished constants. Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathcal{C})$ be an oligomorphic data domain, and let $c \in \mathbb{D}$. Let $c_0 \notin \mathcal{C}$ be some fresh constant symbol, interpreted as c . We have $\text{Aut}(\mathcal{D}^c) = \{\mu \in \text{Aut}(\mathcal{D}) \mid \mu(c) = c\}$. Consider an element $\mathbf{d} = (d_1, \dots, d_n) \in \mathbb{D}^n$ for some $n \in \mathbb{N}$. Its orbit in $\mathcal{D}^c$ is

$$\begin{aligned} \text{orbit}_{\mathcal{D}^c}(\mathbf{d}) &= \{\mu(\mathbf{d}) \mid \mu \in \text{Aut}(\mathcal{D}^c)\} \\ &= \{\mu(\mathbf{d}) \mid \mu \in \text{Aut}(\mathcal{D}), \mu(c) = c\} \end{aligned}$$

Consider now the orbit of $(d_1, \dots, d_n, c)$ in $\mathcal{D}$. It is $\text{orbit}_{\mathcal{D}}((d_1, \dots, d_n, c)) = \{\mu(d_1, \dots, d_n, c) \mid \mu \in \text{Aut}(\mathcal{D})\}$. Let $H_{n+1}^c = \{(d_1, \dots, d_{n+1}) \in \mathbb{D}^{n+1} \mid d_{n+1} = c\}$ be the set of $(n+1)$ -uples whose last component is equal to c . We have that

$$\text{orbit}_{\mathcal{D}}((d_1, \dots, d_n, c)) \cap H_{n+1}^c = \{\mu(d_1, \dots, d_n, c) \mid \mu \in \text{Aut}(\mathcal{D}), \mu(c) = c\}$$

Technically, H_{n+1}^c is called an hyperplane, hence its name.

Thus, projecting on the n first elements yields

$$\pi_{\mathbb{D}^n}(\text{orbit}_{\mathcal{D}}((d_1, \dots, d_n, c)) \cap H_{n+1}^c) = \text{orbit}_{\mathcal{D}^c}(\mathbf{d})$$

As a consequence, the function $\text{Orbits}_{\mathcal{D}}(\mathbb{D}^{n+1}) \rightarrow \text{Orbits}_{\mathcal{D}^c}(\mathbb{D}^n)$ that maps any orbit $\Omega \in \text{Orbits}_{\mathcal{D}}(\mathbb{D}^{n+1})$ to $\pi_{\mathbb{D}^n}(\Omega \cap H_{n+1}^c)$ is surjective, which implies that $\text{Orbits}_{\mathcal{D}^c}(\mathbb{D}^n)$ is finite, since $\text{Orbits}_{\mathcal{D}}(\mathbb{D}^{n+1})$ is itself finite, as $\mathcal{D}$ is oligomorphic. More precisely, we have $\mathcal{R}_{\mathcal{D}^c}(n) \leq \mathcal{R}_{\mathcal{D}}(n+1)$. This concludes the proof. $\square$

Decidable Data Domains

Definition 12.5 We say that $\mathcal{D}$ is *decidable* if its Ryll-Nardzewski function is computable and $\text{FO}[\leq]$ has decidable satisfiability problem over $\mathcal{D}$. Moreover, we say that $\mathcal{D}$ is *polynomially decidable* if any orbit of $\mathbb{D}^k$ can be expressed by some FO formula whose size is polynomial in k and the FO satisfiability problem is decidable using polynomial space.

Remark 12.3 If $\mathcal{D}$ is polynomially decidable, then $\mathcal{R}_{\mathcal{D}}$ is exponential, since there are exponentially many distinct formula of polynomial size.

Example 12.4 The data domain $(\mathbb{D}, =)$ is decidable. First, as stated in Example 12.3, $\mathcal{R}(k)$ is the k -th Bell number for each $k \in \mathbb{N}$, so it is computable. Moreover, satisfiability of $\text{FO}[=]$ is decidable, and more precisely in PSPACE. Finally, an orbit of $\mathbb{D}^k$ can be expressed as $\bigwedge_{i=1}^k \bigwedge_{j=1}^k x_i \bowtie_{i,j} x_j$ for variables $x_1, \dots, x_k$ where $\bowtie_{i,j}$ is chosen in $\{=, \neq\}$ for each $(1 \leq i \leq j \leq k)$. Overall, we can see that $(\mathbb{D}, =)$ is actually polynomially decidable.

The same goes for $(\mathbb{Q}, <)$; simply replace $\bowtie_{i,j} \in \{=, \neq\}$ with $\bowtie_{i,j} \in \{=, <, >\}$, so $(\mathbb{Q}, <)$ is again polynomially decidable.

One can show that $(1\{0, 1\}^*, \otimes)$ is not polynomially decidable. However, it is decidable: one characterises a tuple by the set of equations over n variables that it satisfies (there are exponentially many), which can be manipulated by an SMT-solver.

It can be established that $(\{0, 1\}^*, \otimes)$ is exponentially decidable, in the sense of Remark 12.5.

Remark 12.4 One can also establish that orbits of $(\mathbb{Q}, <)$ (and, a fortiori, $(\mathbb{D}, =)$) can be defined with a polynomial formula by using the fact that this domain admits quantifier-elimination. Thus, any orbit can be defined by a quantifier-free formula; taking the minimum such formula yields a formula of linear size. Actually, a type is simply given by the linear order between the free variables (with equalities allowed).

Note that $(\mathbb{Q}, <)$ is homogeneous [26, Definition 7.1 and Example 7.2], which also yields the result by [26, Lemma 7.5].

[26]: Bojańczyk (2019), *Atom Book*

It is a direct consequence of Propositions 12.14 and 12.15 that adding and distinguishing constants preserve decidability and polynomial decidability. Formally,

Proposition 12.16 Let $\mathcal{D} = (\mathbb{D}, \mathcal{R}, \mathbb{C})$ be an oligomorphic data domain. For any finite set of fresh constants K (i.e., $K \cap \mathbb{D} = \emptyset$), $\mathcal{D} \sqcup K$ is decidable. If $\mathcal{D}$ is polynomially decidable, then so is $\mathcal{D} \sqcup K$, if $|K|$ is considered fixed.

Similarly, for any finite set of distinguished constants $K \subset_f \mathbb{D}$, $\mathcal{D}^K$ is decidable, and polynomially decidable if $\mathcal{D}$ is.

Remark 12.5 One could easily define analogous notion of *exponentially decidable* data domains, and more generally *f-decidable* for some fixed complexity function f . We only study the case of decidable and polynomially decidable data domains since it covers $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$. The reader can check that the complexity analysis transfers to the more general setting.

Strictly speaking, we need f to be at least polynomial, but even the simplest data domain $(\mathbb{D}, =)$ already requires a polynomial f .

Roughly speaking, the problems that we consider (emptiness, functionality and continuity) are PSPACE (respectively, decidable) whenever the data domain $\mathcal{D}$ is polynomially decidable (resp., decidable).

12.2.1. Register Automata over Oligomorphic Data Domains

We start by carrying out a brief study of the behaviour of register automata over oligomorphic data domains, as it gives an intuition of the structure induced by oligomorphicity. This also yields results that prove useful later on.

Orbits, Types and First-Order Logic

Orbits and Types Recall that a type is defined as follows.

Definition 4.30 (Type) Let $\nu : X \rightarrow \mathbb{D}$ be a valuation over X . The *type* of ν in $\mathcal{D}$ is the set $\text{type}_{\mathcal{D}}(\nu)$ of *first-order* formulas $\varphi(X)$ (i.e. we allow the use of quantifiers) that ν satisfies over domain $\mathcal{D}$.

Note that in general, $\text{type}(\nu)$ is infinite.

We simply write $\text{type}(\nu)$ when $\mathcal{D}$ is clear from the context.

The set of types over $\mathcal{D}$ is denoted $\text{Types}(\mathcal{D})$.

At this point, the reader who is familiar with the notion should feel a connection between this notion and that of orbit. And indeed, by [26, Claim 4.12], we get:

For instance by reading Section 4.5.5.

Proposition 12.17 (Orbits and Types [26, Claim 4.12]) *For an oligomorphic data domain and for any $k \in \mathbb{N}$, two k -uples have the same orbit whenever they have the same type.*

[26]: Bojańczyk (2019), *Atom Book*

Recall that by definition, this is the case whenever they satisfy the same set of first-order formulas.

In our study, we are mainly interested in valuations and configurations, so we make precise what we mean when we mention their type. In the following, we assume that the set R of registers is of the form $R = \{r_1, \dots, r_k\}$ for some $k \geq 1$. This is without loss of generality. Note that in particular, this induces an ordering over registers: $r_1 < \dots < r_k$. We use this fact to interpret valuations over R as k -uples.

Definition 12.6 The type of a valuation $\nu : R \rightarrow \mathbb{D}$ is $\text{type}((r_1, \dots, r_k))$. For $C = (q, \nu)$ a configuration, its *type* is defined as $\text{type}(C) = (q, \text{type}(\nu))$.

An immediate corollary of Proposition 12.17 is that when two configurations have the same type, one can be mapped to the other through an automorphism (and conversely):

Corollary 12.18 *Let $\nu, \nu' : R \rightarrow \mathbb{D}$ be two valuations. $\text{type}(\nu) = \text{type}(\nu')$ if and only if there exists some morphism $\mu \in \text{Aut}(\mathcal{D})$ such that $\mu(\nu) = \nu'$.*

Let A be a register automaton. This property is extended to configurations of A by considering morphisms over $\mathcal{D} \sqcup Q$, where Q is seen as a finite data domain with $\text{Aut}(Q) = \{\text{id}_Q\}$ (or, equivalently, as a finite set of constants).

We assume without loss of generality that $Q \cap \mathfrak{S} = \emptyset$, where $\mathfrak{S}$ is the signature of the data domain.

Automorphisms of $\mathcal{D} \sqcup Q$ are exactly automorphisms $\mu \in \text{Aut}(\mathcal{D})$ extended by letting $\mu(q) = q$ for all $q \in Q$. We can formulate the property as follows: let $C = (p, v)$ and $C' = (q, v')$ be two configurations of A . $\text{type}(C) = \text{type}(C')$ if and only if there exists some morphism $\mu \in \text{Aut}(\mathcal{D} \sqcup Q)$ such that $\mu(C) = C'$.

In the following, this extension will be left implicit to lighten the notations. Thus, we pick $\mu \in \text{Aut}(\mathcal{D})$ and implicitly extend it to configurations by first letting $\mu(q) = q$ for all $q \in Q$ and then applying it pointwise: $\mu(q, v) = (q, \mu(v))$.

Expressing Transitions in First-Order Logic By definition of a type, we also have that the ability to take a transition from a configuration only depends on the type of the configuration.

Lemma 12.19 *Let A be some register automaton. Let C, D be two configurations of A such that $\text{type}(C) = \text{type}(D)$. For any transition t of A , t is enabled from C if and only if it is enabled from D .*

The proof is merely an observation, since enabling can be expressed with a first-order formula. First, notice that a transition can be represented by a formula.

Definition 12.7 Let $R = \{r_1, \dots, r_k\}$ be a set of k registers (for some $k \geq 1$), and $(\phi, \text{regOp}) \in \text{Actions}_{\mathcal{D}}(R)$. To (ϕ, regOp) , we associate the quantifier-free formula

$$\psi_{\phi, \text{regOp}}(r_1, \dots, r_k, \star, r'_1, \dots, r'_k) := \phi(r_1, \dots, r_k, \star) \wedge v_{\text{regOp}}(r_1, \dots, r_k, \star, r'_1, \dots, r'_k)$$

where

$$v_{\text{regOp}}(r_1, \dots, r_k, \star, r'_1, \dots, r'_k) := \bigwedge_{i=1}^k \bigwedge_{\text{regOp}(r_i)=\text{keep}} r'_i = r_i \wedge \bigwedge_{\text{regOp}(r_i)=\text{set}} r'_i = \star$$

Let A be some register automaton. By a slight abuse of notation, for any transition $t = p \xrightarrow{\phi, \text{regOp}} q$ (for some states p and q), we denote $\psi_t := \psi_{\phi, \text{regOp}}$.

Observation 12.20 Let A be some register automaton. It follows from the semantics of register automata that for any configurations $C = (p, v)$ and $C' = (q, \lambda)$, and for all transitions $p \xrightarrow{\phi, \text{regOp}} q$ and all data values $d \in \mathbb{D}$, $C \xrightarrow{d} C'$ if and only if $\psi_t(C, d, C')$ holds.

We can now establish Lemma 12.19, which follows from this observation.

Proof of Lemma 12.19. Let A be some register automaton. Let $C, D \in \text{Configs}(A)$ be two configurations of A such that $\text{type}(C) = \text{type}(D)$. Write $C = (p, v)$ and $D = (p, \lambda)$. Let $t = p \xrightarrow{\phi, \text{regOp}} q$ be a transition, and assume that t is enabled from C . Let $d \in \mathbb{D}$ and C' be such that $C \xrightarrow{d, \phi, \text{regOp}} C'$. By Observation 12.20, this means that $\varphi(r_1, \dots, r_k) := \exists d, \exists r'_1, \dots, \exists r'_k \cdot \psi_t(r_1, \dots, r_k, d, r'_1, \dots, r'_k)$ is satisfied by v . Since C and D have the same type, they satisfy the same formulas, so in particular

In particular, $\text{type}(C) = \text{type}(C')$ imply that they have the same state.

t is enabled from C whenever there exists d such that $C \xrightarrow{d} C'$ for some target configuration C' .

Note that we impose no condition on r'_i when $\text{regOp}(r_i) = \text{guess}$, as it is then allowed to take any data value.

By the formula $\psi_t(C, d, C')$, we mean $\psi_t(v(r_1), \dots, v(r_k), d, \lambda(r_1), \dots, \lambda(r_k))$.

By definition of our notion of type for configurations, having the same type implies having the same state (Definition 12.6).

When we say that $\varphi(r_1, \dots, r_k)$ is satisfied by v , we naturally mean that it is true when each variable r_j for $1 \leq j \leq k$ is replaced by $v(r_j)$.

$\exists d, \exists r'_1, \dots, r'_k \cdot \psi_t(D, d)$ is satisfied by v . This precisely means that t is also enabled from D . $\square$

Orbits and First-Order Logic A key property of oligomorphic data domains is that their orbits can be defined by first order formulas. In the following, we denote by $\text{FO}[\mathfrak{S}]$ the set of first-order formulas over signature $\mathfrak{S} = \mathfrak{R} \sqcup \mathfrak{C}$, or simply FO when $\mathfrak{S}$ is clear from the context. As demonstrated in [26], we have:

Proposition 12.21 ([26, Lemma 4.11]) *Let $\mathcal{D}$ be an oligomorphic data domain, and let k be a natural number. Any orbit of an element of $\mathbb{D}^k$ is first-order definable.*

As a consequence, we can refine Lemma 12.19 by requiring that the types of the target configurations also match.

Proposition 12.22 *Let A be some register automaton. Let C, D be two configurations of A such that $\text{type}(C) = \text{type}(D)$. For any transition t of A , if $C \xrightarrow{d}_t C'$ for some $d \in \mathbb{D}$ and some configuration C' , then there exists $d' \in \mathbb{D}$ and a configuration D' such that $D \xrightarrow{d'}_t D'$ and $\text{type}(C') = \text{type}(D')$.*

Proof. Let A be some register automaton, and C, D be two configurations of A such that $\text{type}(C) = \text{type}(D)$. Let t be a transition of A , and assume that $C \xrightarrow{d}_t C'$ for some $d \in \mathbb{D}$ and some configuration C' . Let $\tau' = \text{type}(C')$ be given as a first-order formula (thanks to Propositions 12.17 and 12.21). Since $C \xrightarrow{d}_t C'$ and $\text{type}(C') = \tau'$, we know that $\exists d, \exists r'_1, \dots, r'_k \cdot \psi_{C,d} \wedge \tau'(r'_1, \dots, r'_k)$ is satisfiable (see Observation 12.20). Then, C and D have the same type, so we know that $\exists d', \exists r'_1, \dots, r'_k \cdot \psi_{D,d} \wedge \tau'(r'_1, \dots, r'_k)$ is satisfiable as well. Let $d', r'_1, \dots, r'_k \in \mathbb{D}$ be some data values that satisfy it, and define λ as $\lambda(r_j) = r'_j$ for all $1 \leq j \leq k$. By taking q the state of C' , we get that $D' = (q, \lambda)$ is such that $D \xrightarrow{d'}_t D'$ and $\text{type}(D') = \text{type}(C')$. $\square$

Fugue: Projection over Labels

To give a flavour of the behaviour of register automata over oligomorphic data domains, we show that the set of feasible action sequences is ω -regular (provided we take a finite set of tests), which implies that the projection over labels is also ω -regular. This yields a proof that emptiness is decidable for register automata over decidable data domains. This section is not strictly necessary for our argument on deciding functionality and continuity as we provide a standalone proof of this fact later on (Theorem 12.26), but it completes the picture of register automata that we started in Chapter 4 and might give a better understanding of Theorem 12.26.

We first recall the necessary definitions:

Definition 4.26 (Action Sequence) An (infinite) *action sequence* over data domain $\mathcal{D}$ and set of registers R is an infinite sequence

$$\alpha = \phi_1 \text{asgn}_1 \phi_2 \text{asgn}_2 \cdots \in (\text{Tests}_{\mathcal{D}}(R) 2^R)^\omega$$

We denote by $\text{ActSeq}_{\mathcal{D}}(R)$ the set of action sequences over $\mathcal{D}$ and R .

[26]: Bojańczyk (2019), *Atom Book*

Feasible action sequences are defined in Section 4.5.4, but we recall the necessary definitions.

We banned labels from this part, but it should be obvious how to restore them. Otherwise, simply consider the result in the case of a unary alphabet; this suffices to reduce the emptiness problem of register automata to its analogous for ω -automata.

The impatient reader should feel free to jump to Section 12.2.2.

We did not study register automata over oligomorphic data domains in Chapter 4 since the case of $(\mathbb{Q}, <)$ was then sufficient for our purpose, and introducing the terminology would have been cumbersome.

Definition 4.27 (Compatible Data Word) A data word $x = d_0 d_1 \dots \in \mathbb{D}^\omega$ is *compatible* with α whenever there exists an infinite sequence of valuations $(v_i)_{i \in \mathbb{N}} \in (\text{Val}_{\mathbb{D}}(R))^\omega$ such that $v_0 = v_R^l$ and for all $i \in \mathbb{N}$, $v_i\{\star \leftarrow d_i\} \models \phi_i$ and $v_{i+1} = v_i\{\text{asgn}_i \leftarrow d_i\}$. In other words, $x \in \mathbb{D}^\omega$ is compatible with α whenever there is a path over $x \otimes \alpha$ in the data processing transition system $\mathcal{T}_R^{\mathbb{D}}$.

Definition 4.28 (Feasible Action Sequence) Given an action sequence α , we then define the set of compatible data words as:

$$\text{Comp}(\alpha) = \{x \in \mathbb{D}^\omega \mid x \text{ is compatible with } \alpha\}$$

An action sequence α is *feasible* if $\text{Comp}(\alpha) \neq \emptyset$. The set of feasible action sequences over R is denoted as:

$$\text{Feasible}_{\mathbb{D}}(R) = \{\alpha \in \text{ActSeq}_{\mathbb{D}}(R) \mid \text{Comp}(\alpha) \neq \emptyset\}$$

$\mathbb{D}$ may be omitted when it is clear from the context.

The following result generalises Proposition 4.29 and Proposition 4.34.

Proposition 12.23 Let $\mathbb{D}$ be a decidable oligomorphic data domain, and let R be a set of registers.

For any finite subset $\Phi \subset_f \text{Tests}(R)$, there exists an ω -automaton A_Φ^R over the alphabet $\Phi \times \text{RegOp}_R$ with states $\text{Types}(\mathbb{D})$ and a trivial acceptance condition which accepts $\text{Feasible}_{\mathbb{D}}(R) \cap (\Phi \times \text{RegOp}_R)^\omega$.

If $\mathbb{D}$ is polynomially decidable, then A_Φ^R can be represented in space polynomial in $|\Phi|$ and $|R|$.

This condition is necessary to get PSPACE complexity later on. See Section B.2.1 for a definition of the representation of an ω -automaton.

Proof. This is essentially a corollary of Proposition 12.22: the core property that we use is that given a transition, the existence of a successor configuration of a given type only depends on the type of the current configuration, since having the same type implies satisfying the same set of first-order formulas. We develop the argument as it makes the proof of Theorem 12.26 (on which we heavily rely) more precise. Note that the standalone proof that we give later on (see Section Subsection 12.2.3 on page 255) essentially consists in checking emptiness of an ω -automaton recognising accepting runs of a given register automaton, although we present it in a different way.

Let $\mathbb{D}$ be a decidable oligomorphic data domain, and let R be a set of registers. Finally, let $\Phi \subset_f \text{Tests}(R)$ be a finite set of tests. Define $A_\Phi^R = (\text{Types}(\mathbb{D}), \Phi \times \text{RegOp}_R, \tau^l, \Delta, \Omega)$, where $\tau^l = \bigwedge_{r \in R} r = \#$ and $\Omega = \text{Types}(\mathbb{D})^\omega$ is a trivial acceptance condition. Let $\tau, \tau' \in \text{Types}(\mathbb{D})$ and $(\phi, \text{regOp}) \in \Phi \times \text{RegOp}_R$. There is a transition $\tau \xrightarrow[\text{A}_\Phi^R]{\phi, \text{regOp}} \tau'$ whenever the formula $\psi(x_1, \dots, x_k, d, y_1, \dots, y_k) := \tau(x_1, \dots, x_k) \wedge \tau'(y_1, \dots, y_k) \wedge \phi(d, x_1, \dots, x_k) \wedge v_{\text{regOp}}(x_1, \dots, x_k, d, y_1, \dots, y_k)$ is satisfiable.

Correctness of the construction We start by showing that this automaton accepts the set of feasible action sequences over R in $\mathbb{D}$ and defer the complexity analysis to the end of the proof. First, let $\alpha = (\phi_0, \text{regOp}_0)(\phi_1, \text{regOp}_1) \dots \in (\Phi \text{RegOp}_R)^\omega$ be a feasible action sequence, and let $w = d_0 d_1 \dots \in \text{Comp}(\alpha)$. By definition, there exists a sequence

of valuations $(v_i)_{i \in \mathbb{N}}$ such that $v_0 = v^l$ and for all $i \in \mathbb{N}$, $v_i, d_i \models \phi_i$ and $v_{i+1} = \text{update}(v_i, \text{regOp}_i, d_i)$. For all $i \in \mathbb{N}$, we let $\tau_i = \text{type}(v_i)$. We build by induction a run ρ of A_Φ^R such that for all $i \in \mathbb{N}$, $\text{states}(\rho)[i] = \tau_i$.

Initially, we have $\text{type}(v_0) = \text{type}(v^l) = \tau^l$. Assume that this is the case at step i . We know that $v_i, d_i \models \phi_i$, $v_{i+1} = \text{update}(v_i, \text{regOp}_i, d_i)$ and $\text{type}(v_i) = \tau_i$. We then have that $\tau_i(x_1, \dots, x_k) \wedge \tau_{i+1}(y_1, \dots, y_k) \wedge \phi(d, x_1, \dots, x_k) \wedge v_{\text{regOp}_i}(x_1, \dots, x_k, d, y_1, \dots, y_k)$ is satisfied by taking, for all $j \in R$ $x_j = v_i(r_j)$, $y_j = v_{i+1}(r_j)$ and $d = d_i$. Thus, $\tau_i \xrightarrow{\phi_i, \text{regOp}_i} \tau_{i+1}$, where $\tau_{i+1} = \text{type}(v_{i+1})$, so the inductive invariant holds at step $i + 1$.

Conversely, let ρ be a run of A_Φ^R over some action sequence $\alpha = (\phi_0, \text{regOp}_0)(\phi_1, \text{regOp}_1) \dots \in (\Phi \text{RegOp}_R)^\omega$. We need to show that α is feasible. To that end, we build by induction a sequence of valuations $(v_i)_{i \in \mathbb{N}}$ such that $v_0 = v^l$ and for all $i \in \mathbb{N}$, $v_i, d_i \models \phi_i$ and $v_{i+1} = \text{update}(v_i, \text{regOp}_i, d_i)$. We moreover ask that $\text{type}(v_i) = \text{states}(\rho)[i] := \tau_i$ for all $i \in \mathbb{N}$.

Initially, take $v_0 = v^l$; we have $\text{type}(v^l) = \tau_0$. Now, assume that v_i has been built. Since $\tau_i \xrightarrow{\phi_i, \text{regOp}_i} \tau_{i+1}$, we know that

$$\begin{aligned} \psi_i(x_1, \dots, x_k, d, y_1, \dots, y_k) := & \tau_i(x_1, \dots, x_k) \wedge \tau_{i+1}(y_1, \dots, y_k) \wedge \\ & \phi(d, x_1, \dots, x_k) \wedge \\ & v_{\text{regOp}_i}(x_1, \dots, x_k, d, y_1, \dots, y_k) \end{aligned}$$

is satisfiable. Let $x_1, \dots, x_k \in \mathbb{D}$ which, together with some $d' \in \mathbb{D}$ and some $y'_1, \dots, y'_k \in \mathbb{D}$, satisfy ψ_i . Then, the first-order formula $\exists d, \exists y_1, \dots, y_k, \psi_i(x_1, \dots, x_k, d, y_1, \dots, y_k)$ is satisfiable. We also know that $\tau_i(x_1, \dots, x_k)$. Since $\text{type}(v_i) = \tau_i$, it means that v_i has the same type τ_i as $(x_1, \dots, x_k)$, so, by definition, they satisfy the same set of formula. In particular, it means that $\exists d, \exists y_1, \dots, y_k, \psi_i(v(r_1), \dots, v(r_k), d, y_1, \dots, y_k)$ is satisfiable; let $d_i = d$ and, for all $1 \leq j \leq k$, $v_{i+1}(r_j) = y_j$. We then know that $\text{type}(v_{i+1}) = \tau_{i+1}$ and $v_{i+1} = \text{update}(v_i, \text{regOp}_i, d_i)$, so the inductive invariant holds at step $i + 1$.

Complexity of the construction Now, assume that $\mathcal{D}$ is polynomially decidable. We need to show that A_Φ^R is representable in polynomial space. First, its states can be represented in polynomial space as first-order formulas. The initial state can be represented in space linear in $|R|$, and the acceptance condition is trivial. It remains to show that given $\tau, \tau' \in \text{Types}(\mathcal{D})$ and $\phi \in \Phi$, $\text{regOp} \in \text{RegOp}_R$, one can decide in polynomial space whether there exists a transition $\tau \xrightarrow[\text{A}_\Phi^R]{\phi, \text{regOp}} \tau'$. This amounts to deciding satisfiability of some first-order formula of size linear in $|R|$, which is doable in polynomial space by definition of polynomial decidability. $\square$

As a consequence, we get:

Theorem 12.24 *Let $\mathcal{D}$ be a decidable oligomorphic data domain. Let A be a register automaton with labels over $\mathcal{D}$. The projection over labels $\text{lab}(A)$ is effectively ω -regular.*

The notion of representation of a non-deterministic Büchi automaton is formally defined in Section B.2.1.

More precisely, is is recognised by an ω -automaton with states $Q \times \text{Types}(\mathcal{D})$ (where Q is the state space of A). If $\mathcal{D}$ is polynomially decidable, this automaton can be represented using polynomial space.

Proof. The proof is the same as for Proposition 4.29 and Proposition 4.34: see A as an ω -automaton, and construct the product of A and A_Φ^R (defined in Proposition 12.23) that recognises the intersection of their languages, where R is the set of registers of A , and Φ its set of tests. Such an ω -automaton recognises the language of accepting runs of A . Finally, project away the alphabet $\Phi \times \text{RegOp}_R$, keeping only labels. The acceptance condition is obtained by co-projecting on the Q component. One can check that all those operations can be encoded using polynomial space. $\square$

Since $(\mathbb{Q}, <, 0)$ is polynomially decidable, we get:

Corollary 12.25 *The projection over labels of a non-deterministic register automaton (with guessing) over $(\mathbb{Q}, <, 0)$ is ω -regular.*

Emptiness Problem An immediate corollary of Theorem 12.24 is the decidability of the emptiness problem, as $\text{lab}(A) = \emptyset$ if and only if $L(A) = \emptyset$. Moreover, when $\mathcal{D}$ is polynomially decidable, we obtain membership of PSPACE since $\text{lab}(A)$ is recognised by an ω -automaton that can be represented using polynomial space.

Theorem 12.26 *Let $\mathcal{D}$ be a decidable (respectively, polynomially decidable) oligomorphic data domain. The emptiness problem for non-deterministic register automata over $\mathcal{D}$ is decidable (respectively in PSPACE).*

We later on provide a standalone proof by describing a non-deterministic algorithm that checks emptiness using polynomial space. It essentially consists in checking emptiness of $\text{lab}(A)$, but we feel that this presentation sheds a different and useful light on the problem (see Section 12.2.3).

Note that since $(\mathbb{Q}, <, 0)$ is polynomially decidable (Example 12.4), we get the following result, that is already known from [39, Theorem 14]:

Theorem 12.27 ([39]) *The emptiness problem for register automata over $(\mathbb{Q}, <, 0)$ is PSPACE -complete.*

12.2.2. Uniform Continuity and Cauchy Continuity

As we have seen by our short study of register automata over oligomorphic data domains, these domains enjoy some form of finiteness. Correspondingly, oligomorphicity allows to recover some form of compactness, which implies that uniform continuity and Cauchy continuity coincide, as in the finite alphabet case.

Proposition 12.28 *Let $\mathcal{D}$ be an oligomorphic data domain. The metric space $\mathbb{D}^\infty / \text{Aut}(\mathbb{D})$ is compact, equipped with the distance*

$$d([x], [y]) = \min\{d(u, v) \mid u \in [x], v \in [y]\}$$

Formally, the co-projection of $\Omega \in Q^\omega$ is defined as $(\pi_Q^X)^{-1}(\Omega) = \{(q_0, x_0)(q_1, x_1) \cdots \in (Q \times X)^\omega \mid q_0 q_1 \cdots \in \Omega\}$, where π_Q^X denotes the projection of $Q \times X$ to Q .

Recall that in this part, automata and transducers natively handle guessing.

As usual, PSPACE -hardness results from the fact that the problem is already PSPACE -hard over $(\mathbb{Q}, =)$, see Theorem 4.20.

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

Note that $y = \mu(x)$ implies that x and y have the same length, with the convention that two infinite words have the same length.

Proof. Let $\mathcal{D}$ be an oligomorphic data domain. By definition, the quotient $\mathbb{D}^\omega / \text{Aut}(\mathbb{D})$ is constructed as follows: given two words $x, y \in \mathbb{D}^\omega$, $x \sim y$ if there exists an automorphism $\mu \in \text{Aut}(\mathbb{D})$ such that $y = \mu(x)$. It is well-known that $\sim$ is an equivalence relation. Then, elements of $\mathbb{D}^\omega / \text{Aut}(\mathbb{D})$ are the equivalence classes of $\sim$. Given $x \in \mathbb{D}^\omega$, we denote by $[x]$ the equivalence class of x .

The main idea of the proof is that since $\mathcal{D}$ is oligomorphic, we have that for all $i \in \mathbb{N}$, $\mathbb{D}^i$ only has finitely many orbits. As a consequence, one can successively extract subsequences from $(x_n)_{n \in \mathbb{N}}$ such that their first i data values all belong to the same orbit. Overall, we get a sequence $(y_n)_{n \in \mathbb{N}}$ such that from the i -th data word, all $y_n[0:i]$ belong to the same orbit. This means that there exists a sequence $(\mu_n)_{n \in \mathbb{N}}$ such that $(\mu_n(y_n))_{n \in \mathbb{N}}$ converges, since from step i , all $\mu_n(y_n)$ agree on their i first data values. Note that it actually converges to $y \in \mathbb{D}^\omega$ defined by $y[i] = \mu_i(y_i[i])$.

Formally, let $([x_n])_{n \in \mathbb{N}}$ be a sequence of elements of $\mathbb{D}^\omega / \text{Aut}(\mathbb{D})$. Let us build by induction on $n \in \mathbb{N}$ a subsequence $(y_m)_{m \in \mathbb{N}}$, along with a sequence $(\mu_m)_{m \in \mathbb{N}}$, such that for all $m \in \mathbb{N}$, the two following conditions hold:

- (i) $x_n[0:m]$ is in the same orbit as $y_m[0:m]$ for infinitely many $n \in \mathbb{N}$
- (ii) $|\mu_{m+1}(y_{m+1}) \wedge \mu_m(y_m)| \geq m$.

Since $\mathcal{D}$ is oligomorphic, $\mathbb{D}$ has finitely many orbits, so there is at least one orbit of $\mathbb{D}$ that contains infinitely many $x_n[0]$. Let y_0 be one of them. Then, $x_n[0]$ is in the same orbit as $y_0[0]$ for infinitely many $n \in \mathbb{N}$, and item (i) is satisfied. The second item (ii) holds for any choice of μ_0 and μ_1 , so one can pick $\mu_0 = \text{id}_{\mathbb{D}}$.

Now, let $m \in \mathbb{N}$, and assume by induction that we built y_m and μ_m satisfying items (i) and (ii). Consider the infinitely many x_n such that $x_n[0:m]$ is in the same orbit as $y_m[0:m]$. Since $\mathbb{D}$ is oligomorphic, $\mathbb{D}^{m+1}$ only has finitely many orbits. As a consequence, among those x_n , there are infinitely many such that all $x_n[0:m+1]$ are in the same orbit. Take y_{m+1} to be one of them. Then y_{m+1} satisfies item (i). Moreover, $y_{m+1}[0:m]$ is in the same orbit as $y_m[0:m]$, so it is also in the same orbit as $(\mu_m(y_m[0:m]))$. Thus, there exists $\mu_{m+1} \in \text{Aut}(\mathbb{D})$ such that $\mu_{m+1}(y_{m+1}[0:m]) = \mu_m(y_m[0:m])$. Such μ_{m+1} makes item (ii) true.

Finally, define $y \in \mathbb{D}^\omega$ as, for all $i \in \mathbb{N}$, $y[i] = \mu_i(y_i[i])$. An easy induction establishes, thanks to item (ii), that for all $m \in \mathbb{N}$, $|\mu_m(y_m) \wedge y| \geq m$, which means that $(\mu_m(y_m))_{m \in \mathbb{N}}$ converges to y . As a consequence, $([y_m])_{m \in \mathbb{N}}$ is a convergent subsequence of $([x_n])_{n \in \mathbb{N}}$. $\square$

This property suffices for the notions of uniform continuity and Cauchy continuity to coincide.

Theorem 12.29 ([142, Proposition 2.7]) *Let $\mathcal{D}$ be an oligomorphic data domain, and let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ be an equivariant function. Then, f is uniformly continuous if and only if it is Cauchy continuous.*

Proof. The left-to-right direction always hold, by definition. Conversely, let $\mathcal{D}$ be an oligomorphic data domain, and let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ be an equivariant function. Assume that f is not Cauchy continuous, and let $i \in \mathbb{N}$ and a sequence $(x_n, y_n)_{n \in \mathbb{N}}$ such that for all $n \in \mathbb{N}$, $|x_n \wedge y_n| \geq n$ but $|f(x_n) \wedge f(y_n)| \leq i$. We consider the sequence $([x_n], [y_n])_{n \in \mathbb{N}}$ of pairs

Here, we confuse the finite data word $u = d_0 \dots d_m$ with the tuple $(d_0, \dots, d_m)$, so the orbit of $x_n[0:m]$ is to be understood as the orbit of $(x_n[0], \dots, x_n[m])$.

For $(\mathbb{D}, =)$ and $(\mathbb{Q}, <)$ there is even a single orbit over singletons, namely the entire data domain. This is not the case anymore when one distinguishes some constants. See also Example 3.21 in [26]: Bojańczyk (2019), *Atom Book* for a more complex example.

One could show equally easily that $(\mu_m(y_m))_{m \in \mathbb{N}}$ is a Cauchy sequence, which implies that it converges since $(\mathbb{D}^\omega, \|\cdot, \cdot\|)$ is complete.

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

of elements of $\mathbb{D}^\infty / \text{Aut}(\mathbb{D}^\omega)$, i.e. of words up to automorphisms. Since $\mathbb{D}^\infty / \text{Aut}(\mathbb{D})$ is compact by Proposition 12.28, we can extract a subsequence which is convergent. We also call it $([x_n], [y_n])_{n \in \mathbb{N}}$ for convenience. This means that there are morphisms $(\mu_n)_{n \in \mathbb{N}}$ such that $(\mu_n(x_n), \mu_n(y_n))_{n \in \mathbb{N}}$ converges. By interleaving $(\mu_n(x_n))$ and $(\mu_n(y_n))$, we obtain a convergent sequence whose image is divergent. Formally, define $(z_n)_{n \in \mathbb{N}}$ as, for all $n \in \mathbb{N}$, $z_{2n} = \mu_n(x_n)$ and $z_{2n+1} = \mu_n(y_n)$. Then we have, for all $n \in \mathbb{N}$, $f(z_{2n}) = f(\mu_n(x_n)) = \mu_n(f(x_n))$, similarly $f(z_{2n+1}) = \mu_n(f(y_n))$. Since μ_n is bijective, we get that $\mu_n(f(x_n)) \wedge \mu_n(f(y_n)) = \mu_n(f(x_n) \wedge f(y_n))$, so $|f(z_{2n}) \wedge f(z_{2n+1})| \leq i$, which means that $(f(z_n))_{n \in \mathbb{N}}$ does not converge, while $(z_n)_{n \in \mathbb{N}}$ does, since both (x_n) and (y_n) converge to the same limit. This is a witness that f is not Cauchy continuous. $\square$

Recall that the Cartesian product of two compacts is compact.

Note that convergence is preserved when applying a morphism.

12.2.3. Characterising Functionality and Continuity

Our goal is to show that one can decide, under reasonable computability assumptions, the next-letter problem, as well as continuity and uniform continuity (hence also Cauchy continuity, see Theorem 12.29) over oligomorphic data domains. As a first step, we prove characterisations of these properties, and later on show how to decide them, by establishing that these characterisations can be checked by register automata.

Loop Removal

The main goal of this section is to give characterisations of functionality, continuity and uniform continuity for non-deterministic register transducers over oligomorphic data domains. Those characterisations consist in small witnesses, which are obtained by pumping arguments that rely on the finiteness of the number of configuration orbits.

We start by defining loop removal, which is instrumental in exhibiting small witnesses. Contrary to the finite alphabet case, no actual loop over the same configuration is guaranteed to exist over a given run. However, oligomorphicity guarantees that over long enough runs, a configuration *orbit* repeats.

Definition 12.8 (Loop removal) We consider a partial run to be *weakly looping* if it is of the shape $C \xrightarrow{u|u'} D \xrightarrow{v|v'} \mu(D) \xrightarrow{w|w'} E$ for some $v \neq \varepsilon$ and some morphism $\mu \in \text{Aut}(\mathcal{D})$. By applying μ to the first part of the run, we can *remove the weak loop* and get the shorter path $\mu(C) \xrightarrow{\mu(u)|\mu(u')} \mu(D) \xrightarrow{w|w'} E$, which is still a partial run by Proposition 12.13.

If, additionally, μ is such that $\mu(C) = C$, we say that the partial run is *looping*. We can then *remove the loop* and get the shorter path $C \xrightarrow{\mu(u)|\mu(u')} \mu(D) \xrightarrow{w|w'} E$, which is still a run, again because runs of register automata are closed under automorphisms.

Small run witnesses Loop removal allows to shorten witnesses of the existence of runs of register automata, which provides another way to decide whether a given register automaton is empty.

Proposition 12.30 (Small run witness) *Let T be a non-deterministic register transducer with k registers and n states over data domain $\mathcal{D}$. Assume that $C \xrightarrow{u|u'} D$ is a partial run for $C, D \in \text{Configs}(T)$ and $u, u' \in \mathbb{D}^*$. Then, there exists $w, w' \in \mathbb{D}^*$ with $|w| \leq n^2 \cdot \mathcal{R}_{\mathcal{D}}(2k)$ such that $C \xrightarrow{w|w'} D$.*

Proof. The main idea is a direct generalisation of the classical pumping argument yielding small run witnesses in the finite alphabet case: any large enough run must contain a loop and can thus be shortened.

Formally, Let T be a non-deterministic register transducer with k registers and n states over data domain $\mathcal{D}$. Let $u = d_0 \dots d_{l-1} \in \mathbb{D}^*$, and assume that there is a partial run $C_0 \xrightarrow{d_0|v_0} C_1 \dots C_{l-1} \xrightarrow{d_{l-1}|v_{l-1}} C_l$, and that $l \geq n^2 \cdot \mathcal{R}(2k)$ (otherwise we are done). We need to exhibit a shorter run; the process can then be iterated until $l \leq n^2 \cdot \mathcal{R}(2k)$. Consider the orbits of the pairs (C_0, C_i) for $0 \leq i \leq l$. Recall that each configuration C_i is itself a pair (q, v) , where $v : R \rightarrow \mathbb{D}$ is a valuation of the registers. Since there are $n^2 \cdot \mathcal{R}(2k)$ distinct orbits for pairs of configurations, there must be two pairs of configurations with the same orbit, i.e. there exists two indices $0 \leq i < j \leq l$ and some automorphism μ such that $\mu(C_0, C_i) = (C_0, C_j)$. By applying μ to the i first transitions of the partial run, we get the shorter run $C_0 \xrightarrow{\mu(d_0)|\mu(v_0)} \mu(C_1) \dots \mu(C_{i-1}) \xrightarrow{\mu d_{i-1}|\mu(v_{i-1})} \mu(C_i) = C_j \xrightarrow{d_j|v_j} C_{j+1} \dots C_{l-1} \xrightarrow{d_{l-1}|v_{l-1}} C_l$. $\square$

Technically, the iteration of the process can be done by taking the minimal u such that there is a run from C to D over input u , and assume by contradiction that $l \geq n^2 \cdot \mathcal{R}(2k)$. The following argument shows that if this is the case, there exists a strictly shorter such u , which yields a contradiction.

Remark 12.6 Note that by removing weak loops, we can drop the bound on the length of w to $|w| \leq n \cdot \mathcal{R}_{\mathcal{D}}(k)$. However, this comes at the price of modifying the prefix of the run, since then we only have $\mu(C) \xrightarrow{w|w'} D$ instead of $C \xrightarrow{w|w'} D$. This complexifies the proofs but does not change the complexity class, so we prefer to remove ‘strong’ loops. The reader can check that removing weak loops would suffice in our proofs.

A Characterisation of Non-Emptiness for Register Automata As a first application, we show that this small witness property yields another algorithm to decide emptiness of non-deterministic register automata.

Proposition 12.31 *Let A be a non-deterministic register automaton over some oligomorphic data domain $\mathcal{D}$. The following are equivalent:*

- (i) A recognises at least one data ω -word.
- (ii) There exists a partial run $C^t \xrightarrow{u} C \xrightarrow{v} \mu(C)$ with $v \neq \varepsilon$, C a final configuration and $\mu \in \text{Aut}(\mathcal{D})$.
- (iii) There exists a partial run $C^t \xrightarrow{u} C \xrightarrow{v} \mu(C)$ with $v \neq \varepsilon$, $|u|, |v| \leq n^2 \cdot \mathcal{R}(2k)$, C a final configuration and $\mu \in \text{Aut}(\mathcal{D})$.

Proof. The proof is the same as for the finite alphabet case, except that we manipulate orbits of configurations instead of states. Let A be a non-deterministic register automaton with n states and k registers.

First, assume that A recognises at least one data ω -word $x \in \mathbb{D}^\omega$, and let ρ be an accepting run of A over x (item (i)). Let $q_f \in F$ be a state that repeats infinitely often in $\text{states}(\rho)$. Since there are finitely many orbits of configurations, we obtain that there is a configuration orbit with state q_f that repeats at least twice, i.e. there exists C and an automorphism $\mu \in \text{Aut}(\mathcal{D})$ such that $C^t \xrightarrow{u} C \xrightarrow{v} \mu(C)$ for some $u, v \in \mathbb{D}^*$, where C is final, i.e. item (ii) holds.

Assume now that item (ii) holds. Loop removal of Proposition 12.30 immediately yields the existence of u', v' such that $|u'|, |v'| \leq n^2 \cdot \mathcal{R}(2k)$. Note that here, removing weak loops suffices with no alteration of the proof. Indeed, for all automorphisms $\mu \in \text{Aut}(\mathcal{D})$, we have $\mu(\#) = \#$, so $\mu(C^t) = \mu(C^t)$, so we could further show that we can assume $|u'|, |v'| \leq n \cdot \mathcal{R}(k)$.

Finally, suppose that $C^t \xrightarrow{u} C \xrightarrow{v} \mu(C)$ with $|u|, |v| \leq n^2 \cdot \mathcal{R}(2k)$, C a final configuration and $\mu \in \text{Aut}(\mathcal{D})$. We respectively call π and λ the sequences of transitions of the first and second part of the partial run. Then, $\pi\lambda^\omega$ is the sequence of transitions of a run over the data ω -word $u \cdot v \cdot \mu(v) \cdot \mu^2(v) \cdots \in \mathbb{D}^\omega$. $\square$

We specify that π and λ are sequences of transitions, and not partial runs, since in general $C \neq \mu(C)$, so the configurations do not repeat (but they repeat up to automorphism, which is enough for our purpose).

Decision of Emptiness of Register Automata: Take Two We are now ready to provide a standalone proof of Theorem 12.26, that we recall here.

Theorem 12.26 *Let $\mathcal{D}$ be a decidable (respectively, polynomially decidable) oligomorphic data domain. The emptiness problem for non-deterministic register automata over $\mathcal{D}$ is decidable (respectively in PSPACE).*

Proof. We show the result in the case of a polynomially decidable data domain, the more general case can be obtained by forgetting about complexity. Let $\mathbb{D}$ be a polynomially decidable oligomorphic data domain and let A be a non-deterministic register automaton with k registers and state space Q of size $|Q| = n$. Since $\mathbb{D}$ is polynomially decidable, any orbit of $\mathbb{D}^k$ can be represented by a first-order formula whose size is polynomial in k . Using the non-emptiness characterisation from Proposition 12.31, we only need to find a run of length polynomial in Q and $\mathcal{R}(k)$. We describe a non-deterministic algorithm that runs in polynomial space which checks the existence of such a run. It uses a counter bounded by $n^2 \mathcal{R}(2k)$, using space $2 \log(n \cdot \mathcal{R}(2k))$, guesses a run of the automaton and updates the type of configurations using space polynomial in k and $\log n$.

Recall that $\mathcal{R}(k)$ is exponential by Remark 12.3.

Simulating the run goes as follows: the algorithm maintains in memory the value of the counter, plus an *abstract configuration* (p, τ) , where $p \in Q$ is a state and τ is the type of the associated register valuation. Initially, it stores (q^t, τ^t) in memory for some initial state $q^t \in I$, where $\tau^t(x_1, \dots, x_k) = \bigwedge_{1 \leq i \leq k} x_i = \#$. Along its execution, when the algorithm

has some abstract configuration (p, τ) in memory, it guesses a new type $\tau'(y_1, \dots, y_k)$ as well as a transition $t = (p, \phi, \text{regOp}, q)$. Let

$$\begin{aligned} \varphi(x_1, \dots, x_k, \star, y_1, \dots, y_k) &:= \tau(x_1, \dots, x_k) \wedge \tau'(y_1, \dots, y_k) \wedge \\ &\quad \psi_t(x_1, \dots, x_k, \star, y_1, \dots, y_k) \end{aligned}$$

By Observation 12.20 and Proposition 12.22, for any configuration (p, v) such that $\text{type}(v) = \tau$, there exists $d \in \mathbb{D}$ and λ of type τ' such that $(p, v) \xrightarrow{d, \phi, \text{regOp}}_t (q, \lambda)$ if and only if $\varphi(x_1, \dots, x_k, d, y_1, \dots, y_k)$ is satisfiable. Since $\mathbb{D}$ is polynomially decidable, the algorithm can check that this is the case in polynomial space. It then moves to the new configuration type given by (q, τ') and, if its counter does not exceed $n^2 \mathcal{R}(2k)$, it increments it and continues the simulation; otherwise it rejects.

In parallel of the simulation of the run, the algorithm has to find a loop. To that end, at some point when the current abstract configuration is (p_f, τ_f) where p_f is final, it guesses that it is the start of the loop and stores it in memory. If it later on reaches again (p_f, τ_f) , it halts and declares that A is not empty.

Overall, the algorithm uses polynomial space $\log(n^2 \cdot \mathcal{R}(2k))$ to store its counter and space $2 \log n \cdot P(n, k)$ to store the current and the repeating abstract configuration, where $P(n, 2k + 1)$ is the maximum size of a formula describing the orbit of a $(2k + 1)$ -uple. Finally, it uses polynomial space to check satisfiability of φ .

We have described a non-deterministic algorithm that decides whether its input register automaton A is *non-empty*, which runs using a space polynomial in k and $\log n$, hence polynomial in the size of its input (recall that $k = |R| \leq |A|$). Its correctness results from Proposition 12.31. By Immerman–Szelepcsényi’s theorem [150, 151, Theorem 1, Theorem 1], it implies that there exists a non-deterministic algorithm which runs using polynomial space that decides whether its input register automaton is *empty*. By Savitch’s theorem [152, Theorem 1], this means that there exists a deterministic algorithm which again runs in polynomial space and decides whether its input register automaton is empty. In other words, the emptiness problem for register automata is in PSPACE. $\square$

[150]: Immerman (1988), ‘Nondeterministic Space is Closed Under Complementation’

[151]: Szelepcsényi (1988), ‘The Method of Forced Enumeration for Nondeterministic Automata’

[152]: Savitch (1970), ‘Relationships Between Nondeterministic and Deterministic Tape Complexities’

Characterising Functionality

Characterising functionality relies on the same idea, although is a bit more complicated, for two reasons. First, we now need to manipulate outputs; second, we need to exhibit patterns involving two runs, which makes pumping more involved.

Applying Proposition 12.31 to the product automaton recognising pairs of runs allows to synchronously pump runs:

Lemma 12.32 *Let A be a register automaton. The following are equivalent:*

- (i) *The register automaton A has two final runs $C_1 \xrightarrow{x}$ and $C_2 \xrightarrow{x}$ over the same input $x \in \mathbb{D}^\omega$.*

This result is quite auxiliary, but we prefer to present it separately, so as to lighten the exposition of the following more technical properties.

Recall that a run is final if some final state occurs infinitely often in it. It is not necessarily initial.

(ii) There exists two partial runs

$$C_1 \xrightarrow{u} F_1 \xrightarrow{v} D_1 \xrightarrow{w} \mu(F_1)$$

$$C_2 \xrightarrow{u} D_2 \xrightarrow{v} F_2 \xrightarrow{w} \mu(D_2)$$

with $u, v, w \in \mathbb{D}^*$, $v, w \neq \varepsilon$, F_1, F_2 final configurations and $\mu \in \text{Aut}(\mathcal{D})$.

(iii) There exists two partial runs

$$C_1 \xrightarrow{u} F_1 \xrightarrow{v} D_1 \xrightarrow{w} \mu(F_1)$$

$$C_2 \xrightarrow{u} D_2 \xrightarrow{v} F_2 \xrightarrow{w} \mu(D_2)$$

with $u, v, w \in \mathbb{D}^*$, $v, w \neq \varepsilon$, F_1, F_2 final configurations, $\mu \in \text{Aut}(\mathcal{D})$ and, moreover, $|u|, |v|, |w| \leq n^4 \cdot \mathcal{R}(4k)$.

Proof. We develop the argument, for the sake of completeness. The proof is the same as for Proposition 12.31, except that we cannot ensure that final configurations occur at the same time in both runs. Let A be a register automaton.

First, assume that A has two final runs $\rho_1 = C_1 \xrightarrow{x}$ and $\rho_2 = C_2 \xrightarrow{x}$ over the same input $x \in \mathbb{D}^\omega$. Necessarily, infinitely many configurations are final in ρ_1 and ρ_2 , so we can decompose both runs as

$$\rho_1 = C_1 \xrightarrow{x_0} F_1^0 \xrightarrow{y_0} D_1^0 \xrightarrow{x_1} F_1^1 \xrightarrow{y_1} D_1^1 \xrightarrow{x_2} \dots$$

$$\rho_2 = C_2 \xrightarrow{x_0} D_2^0 \xrightarrow{y_0} F_2^0 \xrightarrow{x_1} D_2^1 \xrightarrow{y_1} F_2^1 \xrightarrow{x_2} \dots$$

where for all $i \in \mathbb{N}$, F_1^i, F_2^i are final, and $x_0 \cdot y_0 \cdot x_1 \cdot y_1 \cdots = x$.

Consider the set of tuples of configurations (C_1, C_2, F_1^i, D_2^i) with $i \in \mathbb{N}$. There are finitely many orbits for these tuples, so, necessarily, there exists $i < j$ and a morphism $\mu \in \text{Aut}(\mathbb{D})$ such that $\mu(C_1) = C_1, \mu(C_2) = C_2, \mu(F_1^i) = F_1^j$ and $\mu(D_2^i) = D_2^j$. Thus, we can write

$$\rho_1 = C_1 \xrightarrow{u} F_1^i \xrightarrow{y_i} D_1^i \xrightarrow{w} F_1^j$$

$$\rho_2 = C_2 \xrightarrow{u} D_2^i \xrightarrow{y_i} F_2^i \xrightarrow{w} D_2^j$$

which yields the expected pattern (item (ii)), with $u = x_0 \cdot y_0 \cdot \dots \cdot x_i$, $v = y_i$, $w = x_{i+1} \cdot y_{i+1} \cdot \dots \cdot x_j$, $F_1 = F_1^i$ and $D_2 = D_2^i$.

Now, assume that this pattern is present, i.e. we have item (ii). The pair of partial runs $C_1 \xrightarrow{u} F_1$ and $C_2 \xrightarrow{u} D_2$ is a partial run in the product automaton $A \otimes A$, which has n^2 states and $2k$ registers, and similarly for v and w . We can thus apply Proposition 12.30 to each segment of the pair of runs, which yields $|u|, |v|, |w| \leq n^4 \mathcal{R}(4k)$, which gives item (iii).

Finally, the latter pattern yields two runs of A over $u \cdot v \cdot w \cdot \mu(v) \cdot \mu(w) \cdot \mu^2(v) \cdot \mu^2(w) \cdots \in \mathbb{D}^\omega$, which are final since they each contain infinitely many final configurations. This concludes the proof. $\square$

The above lemma is easily lifted to register transducers, yielding the same statement, with no conditions on the outputs.

We start by showing that we can remove loops while preserving mismatches. The proof is quite technical, but it provides a quite powerful tool to establish the existence of small witnesses for the patterns we exhibit.

Lemma 12.33 (Small mismatch witness) *There exists a polynomial $P(x, y, z)$ such that the following holds.*

Let T be an NRT, with k registers, n states and a maximal output length M . If there exists two partial runs $C_1 \xrightarrow{u|u_1} D_1$ and $C_2 \xrightarrow{u|u_2} D_2$ such that $u_1 \not\parallel u_2$, then there exists u' of length less than $P(\mathcal{R}(4k), n, M)$ and u'_1, u'_2 , so that $C_1 \xrightarrow{u'|u'_1} D_1, C_2 \xrightarrow{u'|u'_2} D_2$ with $u'_1 \not\parallel u'_2$.

Proof. Let $\rho_1 = C_1 \xrightarrow{u|u_1} D_1$ and $\rho_2 = C_2 \xrightarrow{u|u_2} D_2$ be two partial runs such that $u_1 \not\parallel u_2$. We assume that $|u| > P(\mathcal{R}(2k), n, M)$, and we want to show that we can obtain a strictly smaller data word with the desired property. Our goal is thus to remove synchronous loops in ρ_1, ρ_2 while preserving the mismatch. A *synchronous loop* is defined as a loop over the same input in the product transducer $T \otimes T$. Let $u_1 = \alpha a_1 \beta_1, u_2 = \alpha a_2 \beta_2$ with $a_1 \neq a_2$. We will only consider removing loops that do *not* contain the transitions producing the mismatching letters a_1 or a_2 .

We call the *effect* of a loop on ρ_1 , the length of the output factor removed from α , in particular the effect is 0 if the factor removed is in β_1 (and symmetrically for ρ_2). Removing loops that have the same effect on ρ_1 and ρ_2 preserves the mismatch.

The only case remaining is when any loop has different effects on ρ_1 and ρ_2 . The first step is to show that removing a loop with different effects and which cancels a mismatch ensures a very strong periodicity property on the outputs.

Let us consider the first synchronous loop occurring in ρ_1, ρ_2 which does not contain the transitions producing the mismatching outputs. From the previous cases we can assume that this loop has a non-zero effect on ρ_1 (without loss of generality). Let us assume that the loop occurs after the transition producing the mismatching output in ρ_2 . This means that there are no loops occurring before the transition producing a_2 , and thus $|\alpha| < Mn^2 \cdot \mathcal{R}(4k)$ and that all loops have a null effect on ρ_2 . From that we can deduce that there are fewer than $Mn^2 \cdot \mathcal{R}(4k)$ loops in total since any loop has to have a non null effect on ρ_1 . Thus we can deduce that $|u| \leq (Mn^2 \cdot \mathcal{R}(4k))(n^2 \cdot \mathcal{R}(4k))$, which yields a contradiction.

The only case which remains is that all loops occur before the transitions producing the mismatching outputs. We can assume, for u large enough, that there are at least two loops. Let $u_1 = \alpha_1 \beta_1 \beta'_1 \gamma_1 \delta_1 \delta'_1 \zeta_1 a_1 \eta_1$ and $u_2 = \alpha_2 \beta_2 \beta'_2 \gamma_2 \delta_2 \delta'_2 \zeta_2 a_2 \eta_2$ with $\alpha_1 \beta_1 \beta'_1 \gamma_1 \delta_1 \delta'_1 \zeta_1 = \alpha_2 \beta_2 \beta'_2 \gamma_2 \delta_2 \delta'_2 \zeta_2 = u_1 \wedge u_2$, where $\beta_1 \beta'_1, \beta_2 \beta'_2$ denote the outputs produced by the first occurring loop and $\delta_1 \delta'_1, \delta_2 \delta'_2$ correspond to the outputs of the second loop. What we mean by that is that removing the first loop gives the outputs $\alpha_1 \lambda(\beta_1) \gamma_1 \delta_1 \delta'_1 \zeta_1 a_1 \eta_1$ and $\alpha_2 \lambda(\beta_2) \gamma_2 \delta_2 \delta'_2 \zeta_2 a_2 \eta_2$; while removing the second loop gives the outputs $\alpha_1 \beta_1 \beta'_1 \gamma_1 \mu(\delta_1) \zeta_1 a_1 \eta_1$ and

$\alpha_2\beta_2\beta'_2\gamma_2\mu(\delta_2)\zeta_2a_2\eta_2$. Without loss of generality, we assume that $l = |\beta'_1| - |\beta'_2| > 0$. After removing the first loop we obtain $u'_1 = \alpha_1\lambda(\beta_1)\gamma_1\delta_1\delta'_1\zeta_1a_1\eta_1$ and $u'_2 = \alpha_2\lambda(\beta_2)\gamma_2\delta_2\delta'_2\zeta_2a_2\eta_2$, and we assume that $u'_1 \parallel u'_2$, otherwise we are done. We define $u_1 = \alpha\zeta a_1\eta_1$ and $u_2 = \alpha\zeta a_2\eta_2$ where α is the longest data word between $\alpha_1\beta_1\beta'_1\gamma_1\delta_1\delta'_1$ and $\alpha_2\beta_2\beta'_2\gamma_2\delta_2\delta'_2$, so that ζ is a suffix of both ζ_1 and ζ_2 .

Let us assume that $|\zeta| < l$. This means that $|u_1 \wedge u_2| \leq 2Mn^2 \cdot \mathcal{R}(4k)$. Since any loop has to produce at least one data value in u_1 or u_2 , u must have fewer than $4Mn^2 \cdot \mathcal{R}(4k)$ double loops. Thus we get that $|u| < (4Mn^2 \cdot \mathcal{R}(4k))(n^2 \cdot \mathcal{R}(4k))$ which gives a contradiction. Thus we can safely assume that $l \leq |\zeta|$. Let v denote the length l suffix of ζ , then we have that $\alpha_1\lambda(\beta_1)\gamma_1\delta_1\delta'_1\zeta_1v = \alpha_2\lambda(\beta_2)\gamma_2\delta_2\delta'_2\zeta_2$. Thus we have that ζ ends in v^2 , assuming that it is long enough. Repeating this we obtain that ζ is in the language v^*v^* for some v' suffix of v . Let t be the primitive root of v then we can even say that ζ is in t'^* for some t' suffix of t . So we obtain that $\zeta a_1 \parallel t'^\omega$ and $\zeta \parallel t'^\omega$ and $\zeta a_2 \not\parallel t'^\omega$. Let us consider now removing the second loop corresponding to outputs δ'_1, δ'_2 . Using the same arguments we get that $|\beta'_1| - |\beta'_2|$ and $|\delta'_1| - |\delta'_2|$ are both multiples of $|t|$. Note moreover that we cannot have $|\delta_1| - |\delta_2| < 0$, otherwise we would conclude that $\zeta a_1 \not\parallel t'^\omega$. Hence we obtain that all loops must have a larger effect on ρ_1 than on ρ_2 and that the effect difference is a multiple of $|t|$. Now that we have established that strong periodicity property of ζa_1 , the main idea is the following: remove all loops while preserving a ζ larger than the maximal effect of a loop but still smaller than twice that. Once this is done, repump just enough loops so that the first data word is larger than the second one. This is sure to cause a mismatch since $x\zeta \not\parallel y\zeta a_2$, for any x, y satisfying $0 < |x| - |y| < |\zeta|$. This concludes the proof. $\square$

We have now established the existence of small mismatch witnesses. This is sufficient to give a characterisation of functionality which, as we show later on (Theorem 12.40), is decidable.

Proposition 12.34 *There exists a polynomial $P(x, y, z)$ such that the following holds.*

Let $R \subseteq \mathbb{D}^\omega \times \mathbb{D}^\omega$ be a relation given by a non-deterministic register transducer with n states, k registers and a maximum output length M . The following are equivalent:

- (i) R is not functional
- (ii) There exists two partial runs:

$$\begin{aligned} I_1 &\xrightarrow{u|u_1} F_1 \xrightarrow{v|v_1} C_1 \xrightarrow{w|w_1} \mu(F_1) \\ I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} F_2 \xrightarrow{w|w_2} \mu(C_2) \end{aligned}$$

such that I_1 and I_2 are initial, F_1 and F_2 are final, $\mu \in \text{Aut}(\mathbb{D})$ and $u_1 \not\parallel u_2$

(iii) There exists two partial runs:

$$\begin{aligned} I_1 &\xrightarrow{u|u_1} F_1 \xrightarrow{v|v_1} C_1 \xrightarrow{w|w_1} \mu(F_1) \\ I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} F_2 \xrightarrow{w|w_2} \mu(C_2) \end{aligned}$$

such that I_1 and I_2 are initial, F_1 and F_2 are final, $\mu \in \text{Aut}(\mathbb{D})$, $u_1 \not\parallel u_2$ and, moreover, $|u|, |v|, |w| \leq P(\mathcal{R}(4k), n, M)$.

Proof. Let T be a non-deterministic register transducer with n states, k registers and a maximum output length M , which recognises some relation $R = \llbracket T \rrbracket$.

First, assume that R is not functional. This means that T has two accepting runs ρ_1 and ρ_2 over some input data word $x = \text{in}(\rho_1) = \text{in}(\rho_2) \in \mathbb{D}^\omega$, which yields two distinct outputs $y = \text{out}(\rho_1) \in \mathbb{D}^\omega$ and $z = \text{out}(\rho_2) \in \mathbb{D}^\omega$. Since they are distinct and both infinite, y and z necessarily mismatch at some position. Thus, we can decompose ρ_1 and ρ_2 as:

$$\begin{aligned} I_1 &\xrightarrow{u|u_1} C_1 \xrightarrow{t|t_1} \lambda_1 \\ I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{t|t_2} \lambda_2 \end{aligned}$$

where $u \in \mathbb{D}^*$ is some finite prefix of x such that the corresponding (finite) outputs mismatch, i.e. $u_1 \not\parallel u_2$, and $t = u^{-1} \cdot x$ is the infinite suffix yielding (infinite) outputs $t_1, t_2 \in \mathbb{D}^\omega$. By applying Lemma 12.32 to the two runs λ_1 and λ_2 over t , we get the pattern of (ii).

We now need to show that we have small witnesses. The bound on v and w is a consequence of Lemma 12.32 (iii), which yields a polynomial bound (that actually does not depend on M). Applying Lemma 12.33 to the pair of runs $I_1 \xrightarrow{u|u_1} F_1$ and $I_2 \xrightarrow{u|u_2}$ gives a polynomial bound on the size of u , which gives item (iii).

Finally, if we have the latter pattern, consider $x = u \cdot v \cdot w \cdot \mu(v) \cdot \mu(w) \cdot \mu^2(v) \cdot \mu^2(w) \dots$. Then, the first (respectively, second) partial run is a witness that there exists a run of T over x with output $y = u_1 \cdot v_1 \cdot w_1 \cdot \mu(v_1) \cdot \mu(w_1) \dots$ (respectively, $z = u_2 \cdot v_2 \cdot w_2 \cdot \mu(v_2) \cdot \mu(w_2) \dots$). Since $u_1 \not\parallel u_2$, we get that $y \not\parallel z$, so R is not functional. $\square$

Recall that in our study, we only consider transducers which produce infinite outputs, see Assumption 10.3.

Note that t_1 and t_2 may (or may not) be equal.

Characterising Continuity and Uniform Continuity

The above proof techniques can be adapted to get patterns for non-continuity and non-uniform continuity, that are reminiscent of those that we exhibited over $(\mathbb{D}, =)$ (cf Figure 12.3 on page 235 and Figure 12.5 on page 240).

To unify the presentation of the two patterns, we introduce the notion of critical pattern.

Definition 12.9 (Critical pattern) Let T be a non-deterministic register transducer. Two partial runs of T form a *critical pattern* if they are of the form

note that in general, the last input data word of each partial run are distinct, i.e. we do not necessarily have $w = z$.

$$\begin{aligned}
I_1 &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{w|w_1} D_1 \\
I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{z|w_2} D_2
\end{aligned}$$

for some morphism $\mu \in \text{Aut}(\mathbb{D})$, with D_1 and D_2 two co-reachable configurations, and such that one of the following holds:

- (a) $u_1 \not\parallel u_2$, or
- (b) $v_2 = \varepsilon$ and $u_1 \not\parallel u_2 \cdot w_2$, or
- (c) $v_1 = v_2 = \varepsilon$ and $u_1 \cdot w_1 \not\parallel u_2 \cdot w_2$.

Before characterising continuity and uniform continuity, we show that critical patterns also admit small witnesses.

Proposition 12.35 (Small critical patterns) *There exists a polynomial $P'(x, y, z)$ such that the following holds.*

Let T be a non-deterministic register transducer with n states and k registers, and with maximal output length M . Let $I_1, C_1, D_1, I_2, C_2, D_2$ be configurations of T , and $\mu \in \text{Aut}(\mathbb{D})$ be some morphism. If

$$\begin{aligned}
I_1 &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{w|w_1} D_1 \\
I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{z|w_2} D_2
\end{aligned}$$

forms a critical pattern, then we can assume $|u|, |v|, |w|, |z| \leq P'(\mathcal{R}(4k), n, M)$.

Proof. We want to remove loops in u, v, w, z without affecting the mismatches. The idea is to see such a critical pattern as a pair of runs which mismatch and leverage Lemma 12.33. Let T be a non-deterministic register transducer with states Q , with k registers and with maximal output length M . We let $n = |Q|$. Assume that T has a critical pattern for some $u, v, w, z, u_1, u_2, v_1, v_2, w_1, w_2 \in \mathbb{D}^*$, and with otherwise the same notations as in the statement of the proposition.

We first treat the case of $u_1 \not\parallel u_2$, as it is easier. In that case, Lemma 12.33 yields u' with mismatching outputs $u'_1 \not\parallel u'_2$ such that $|u'| \leq P(\mathcal{R}(4k), n, M)$. We can then apply Proposition 12.30 to the product transducer $T \otimes T$ to remove synchronous loops and get v', w', z' with $|v'|, |w'|, |z'| \leq n^4 \cdot \mathcal{R}(4k)$.

For the two other cases, we need to be careful not to affect the intermediate configurations, since the position of the mismatch of the second run lies in z . To that end, we consider runs that are coloured. Formally, we consider the register transducer T' which is the same as T , except that its state space is $Q \times \{R, G, B\}$, where the states of the partial run over u (respectively v and w, z) are coloured in red (respectively in green, blue). Since Lemma 12.33 is proved using loop removal, it does not remove loops which contain states with different colours. By applying it to the two runs of the critical pattern, we get that $|u \cdot v \cdot w|, |u \cdot v \cdot z| \leq P(\mathcal{R}(4k), 3n, M)$. This yields the expected result. $\square$

We are now ready to give a characterisation of continuity and uniform continuity for functions recognised by a non-deterministic register transducer.

Here, we ask that D_1 and D_2 are co-reachable, but not that they are jointly co-reachable.

Saying ‘We can assume $|u|, |v|, |w|, |z| \leq P'(\mathcal{R}(4k), n, M)$ ’ is a short for stating that if there is a critical pattern for some $u, v, w, z, u_1, u_2, v_1, v_2, w_1, w_2 \in \mathbb{D}^*$, then this is also the case for some $u', v', w', z', u'_1, u'_2, v'_1, v'_2, w'_1, w'_2$ with $|u'|, |v'|, |w'|, |z'| \leq P'(\mathcal{R}(4k), n, M)$.

It should be obvious what R, G and B stand for.

Proposition 12.36 Let T be a functional non-deterministic register transducer with n states and k registers, with maximum output length M , that recognises some partial function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$.

Then, f is not continuous if and only if T has a critical pattern

$$\begin{aligned} I_1 &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{w|w_1} D_1 \\ I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{z|z_2} D_2 \end{aligned}$$

where, moreover, C_1 is final.

Proof. The proof is very similar to that of $(\mathbb{D}, =)$ (Theorem 12.11), except that we manipulate configurations up to automorphisms instead of configurations taking a limited number of distinct data values. We develop the argument for completeness; the reader who learnt about the case of $(\mathbb{D}, =)$ may feel confident enough to jump to the characterisation of uniform continuity (Proposition 12.37). Let T be a functional non-deterministic register transducer with n states and k registers, with maximum output length M , that recognises some partial function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$.

First, assume that f is not continuous at some $x \in \mathbb{D}^\omega$. Let $i \in \mathbb{N}$ be such that for all $j \geq 0$, there exists $y \in \text{dom}(f)$ with $|x \wedge y| \geq j$ but $|f(x) \wedge f(y)| \leq i$. Let ρ_1 be a run of T over input x , which yields output $f(x)$, and pick some $q_f \in F$ that appears infinitely often in ρ_1 . We let $K > n\mathcal{R}(k)$ and $L > n^2\mathcal{R}(2k)$. Let $j \geq 0$ be such that there are $2Ki$ occurrences of q_f in the first j transitions of ρ_1 , and let $y \in \text{dom}(f)$ such that $|x \wedge y| \geq j$ but $|f(x) \wedge f(y)| \leq i$. Finally, let ρ_2 be a run of T which yields output $f(y)$. We can decompose ρ_1 and ρ_2 as follows:

$$\begin{aligned} I_1 &\xrightarrow{u^{\text{pr}}|v^{\text{pr}}} C_1^{\text{pr}} \xrightarrow{u^{\text{np}}|v^{\text{np}}} C_1^{\text{np}} \xrightarrow{u^{\text{mm}}|v^{\text{mm}}} C_1^{\text{mm}} \xrightarrow{t_1|t'_1} C_1^{\text{sf}} \\ I_2 &\xrightarrow{u^{\text{pr}}|w^{\text{pr}}} C_2^{\text{pr}} \xrightarrow{u^{\text{np}}|w^{\text{np}}} C_2^{\text{np}} \xrightarrow{u^{\text{mm}}|w^{\text{mm}}} C_2^{\text{mm}} \xrightarrow{t_2|t'_2} C_2^{\text{sf}} \end{aligned}$$

where C_1^{np} is final and $v^{\text{pr}} \cdot v^{\text{np}} \cdot v^{\text{mm}} \cdot t'_1 \not\parallel w^{\text{pr}} \cdot w^{\text{np}} \cdot w^{\text{mm}} \cdot t'_2$. We ask that there are Ki occurrences of q_f in the partial run of ρ_1 over u^{pr} , Li occurrences over u^{np} , and L occurrences of u^{mm} . First, let us show that each sequence of transitions in ρ_1 containing K occurrences of q_f contain at least one productive transition. Assume the contrary. Then, there exists two final configurations F_1 and F_2 in the same orbit with only unproductive transitions in between. Thus, we can write $I_1 \xrightarrow{u'|u'_1} \pi F_1 \xrightarrow{v'|\varepsilon} \mu(F_1)$ for some $u', u'_1, v' \in \mathbb{D}^*$ with $v' \neq \varepsilon$ and some $\mu \in \text{Aut}(\mathbb{D})$. In that case, $\pi \cdot \lambda^\omega$ is the sequence of transitions of an accepting run over $u' \cdot v' \cdot \mu(v') \cdot \mu^2(v') \dots$ which yields *finite* output $u'_1 \in \mathbb{D}^*$. This contradicts Assumption 10.3. As a consequence, we have that $|v^{\text{pr}}| > i$.

Then, since u^{mm} contains L occurrences of q_f we know that there are two pairs of configurations of ρ_1 and ρ_2 with state q_f for ρ_1 that belong to

Recall that over oligomorphic data domains, the renaming property does not hold anymore, so we cannot bound the number of distinct data values along a run.

Note that ρ_1 is not unique in general: there might be multiple runs over x yielding the same output $f(x)$.

Contrary to the case of $(\mathbb{D}, =)$, it does not suffice to take $L = K^2$, as we have no bound on the growth of $\mathcal{R}(k)$.

the same orbit, i.e. we can write

$$\begin{aligned} C_1^{\text{np}} &\xrightarrow{u^{\text{mm}}|v^{\text{mm}}} C_1^{\text{mm}} \\ C_2^{\text{np}} &\xrightarrow{u^{\text{mm}}|w^{\text{mm}}} C_2^{\text{mm}} \end{aligned}$$

as

$$\begin{aligned} C_1^{\text{np}} &\xrightarrow{p|p_1} F_1 \xrightarrow{l|l_1} \mu(F_1) \xrightarrow{s|s_1} C_1^{\text{mm}} \\ C_2^{\text{np}} &\xrightarrow{p|p_2} C_2 \xrightarrow{l|l_2} \mu(C_2) \xrightarrow{s|s_2} C_2^{\text{mm}} \end{aligned}$$

where $p \cdot l \cdot s = w$, idem for w_1 and w_2 , F_1 is final and $\mu \in \text{Aut}(\mathcal{D})$.

Now, there are two cases:

- If $v^{\text{pr}} \cdot v^{\text{np}} \not\parallel w^{\text{pr}} \cdot w^{\text{np}}$, then we get the expected critical pattern (with case (a)) by taking $u = u^{\text{pr}} \cdot u^{\text{np}} \cdot p$, $v = l$, $w = s$ and $z = s$.
- Otherwise, we know that in the partial run over u^{np} , there is a pair of configurations of ρ_1 and ρ_2 with state q_f for ρ_1 and with only improductive transitions in between for ρ_2 . Thus, we can write

$$\begin{aligned} C_1^{\text{pr}} &\xrightarrow{u^{\text{np}}|v^{\text{np}}} C_1^{\text{np}} \\ C_2^{\text{pr}} &\xrightarrow{u^{\text{np}}|w^{\text{np}}} C_2^{\text{np}} \end{aligned}$$

as

$$\begin{aligned} C_1^{\text{pr}} &\xrightarrow{p|p_1} F_1 \xrightarrow{l|l_1} \mu(F_1) \xrightarrow{s|s_1} C_1^{\text{np}} \\ C_2^{\text{pr}} &\xrightarrow{p|p_2} C_2 \xrightarrow{l|\varepsilon} \mu(C_2) \xrightarrow{s|s_2} C_2^{\text{np}} \end{aligned}$$

where F_1 is final. This yields a critical pattern (case (b)) for $u = u^{\text{pr}} \cdot p$, $v = l$, $w = s_1 \cdot u^{\text{mm}} \cdot t_1$ and $z = s_2 \cdot u^{\text{mm}} \cdot t_2$.

In both cases, the fact that ρ_1 and ρ_2 are accepting entail that D_1 and D_2 are each co-reachable.

Conversely, assume that T has a critical pattern, with the same notations as in the statement. Take $x = u \cdot v^\omega$. Since D_2 is co-reachable, take some $t, t_2 \in \mathbb{D}^\omega$ such that $D_2 \xrightarrow{t|t_2}$ is a final run. Define, for all $n \in \mathbb{N}$, $x_n = u \cdot v^n \cdot z \cdot t$. We have that $x_n \rightarrow_{n \rightarrow \infty} x$. The pattern is a witness that $f(x) = u_1 \cdot v_1^\omega$, while $f(x_n) = u_2 \cdot v_2^n \cdot w_2 \cdot t_2$. If $u_1 \not\parallel u_2$, this means that $|f(x) \wedge f(x_n)| \leq |u_1|$, which means that f is not continuous at x . If $v_2 = \varepsilon$, we have that $u_1 \not\parallel u_2 \cdot w_2$, which again entails that f is not continuous at x . Finally, we cannot have $v_1 = \varepsilon$ (case (c) of Definition 12.9), otherwise T produces some finite output over x (since C_1 is final), which contradicts Assumption 10.3. $\square$

We now move to the characterisation of uniform continuity, which follows the same lines, although it is slightly more complicated since we need

to manipulate three inputs at the same time. The characterisation is the same as above, except that we do not ask that C_1 is final.

Proposition 12.37 *Let T be a functional non-deterministic register transducer with n states and k registers, with maximum output length M , that recognises some partial function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$.*

Then, f is not uniformly continuous if and only if T has a critical pattern

$$\begin{aligned} I_1 &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{w|w_1} D_1 \\ I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{z|w_2} D_2 \end{aligned}$$

Proof. Let T be a functional non-deterministic register transducer with n states and k registers, with maximum output length M , that recognises some partial function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$.

Exhibit the pattern First, assume that f is not uniformly continuous. Let $i \geq 0$ be such that for all $j \geq 0$, there exists $x_1, x_2 \in \text{dom}(f)$ with $|x_1 \wedge x_2| \geq j$ but $|f(x_1) \wedge f(x_2)| < i$. Take $j \geq 3i \cdot n^2 \cdot \mathcal{R}(2k)$, and correspondingly $x_1, x_2 \in \text{dom}(f)$ such that $|x_1 \wedge x_2| \geq j$ but $|f(x_1) \wedge f(x_2)| \leq i$. Let ρ_1 and ρ_2 be accepting runs of T on x_1 and x_2 yielding respective outputs $f(x_1)$ and $f(x_2)$. Since $|x_1 \wedge x_2| \geq 3i \cdot n^2 \cdot \mathcal{R}(2k)$, we know there are at least $3i$ pairs of configurations of ρ_1 and ρ_2 before x_1 and x_2 mismatch that have the same orbit Ω . The argument is similar to that for continuity; we phrase it in a slightly different way to shed a different light on the problem. Take a sufficiently long prefix of ρ_1 and ρ_2 so that the mismatch is produced in both runs, and write those prefixes as

Note that we ask $|f(x_1) \wedge f(x_2)|$ is strictly less than i , to avoid carrying ‘+1’ along the proof.

$$\begin{aligned} I_1 &\xrightarrow{p|p_1} C \xrightarrow{u_0|v_0} \mu_1(C) \xrightarrow{u_1|v_1} \mu_2(C) \dots \xrightarrow{u_{3i}|v_{3i}} \mu_{3i+1}(C) \xrightarrow{s_1|t_1} E \\ I_2 &\xrightarrow{p|p_2} D \xrightarrow{u_0|w_0} \mu_1(D) \xrightarrow{u_1|w_1} \mu_2(D) \dots \xrightarrow{u_{3i}|w_{3i}} \mu_{3i+1}(D) \xrightarrow{s_2|t_2} G \end{aligned}$$

where $p, p_1, p_2, s_1, t_1, s_2, t_2 \in \mathbb{D}^*$; for all $0 \leq k \leq 3i$, $u_i, v_i, w_i \in \mathbb{D}^*$ with $u_i \neq \varepsilon$, and $\mu_i \in \text{Aut}(\mathcal{D})$. I_1, I_2, C, D, E, G are configurations of T , with E and G co-reachable. We ask that $p \cdot u_0 \dots u_{3i} \leq x_1, x_2$ and $p_1 \cdot v_0 \dots v_{3i} \cdot t_1 \not\leq w_0 \dots w_{3i} \cdot t_2$. Consider now the set of $3i$ pairs (v_i, w_i) . There are three cases:

- If there exists k such that $v_k = w_k = \varepsilon$, then we can rewrite the prefixes of runs as

$$\begin{aligned} I_1 &\xrightarrow{pu_0 \dots u_{k-1}|p_1v_0 \dots v_{k-1}} C' \xrightarrow{u_k|\varepsilon} \mu(C') \xrightarrow{u_{k+1} \dots u_{3i}s_1|v_{k+1} \dots v_{3i}t_1} E \\ I_2 &\xrightarrow{pu_0 \dots u_{k-1}|p_2w_0 \dots w_{k-1}} D' \xrightarrow{u_k|\varepsilon} \mu(D') \xrightarrow{u_{k+1} \dots u_{3i}s_2|v_{k+1} \dots v_{3i}t_2} G \end{aligned}$$

where $C' = \mu_k(C)$, $D' = \mu_k(D)$ and $\mu = \mu_{k+1} \circ \mu_k^{-1}$. This is a critical pattern of shape (c), with $C_1 = C'$, $C_2 = D'$, $D_1 = E$ and $D_2 = G$, as we have $p_1 \cdot v_0 \dots v_{3i} \cdot t_1 \not\leq w_0 \dots w_{3i} \cdot t_2$.

- We now focus on the first $2i$ pairs. If $v_j \neq \varepsilon$ and $w_k \neq \varepsilon$ for (strictly) more than i different $0 \leq j, k < 2i$, then we have that $v_0 \dots v_{2i-1} \not\leq w_0 \dots w_{2i-1}$, since we know that $|f(x_1) \wedge f(x_2)| \leq i$

and $|v_0 \dots v_{2i-1}|, |w_0 \dots w_{2i-1}| > i$, while the latter are prefixes of $f(x_1)$ and $f(x_2)$ respectively. This yields a critical pattern of shape (a), with $u = u_0 \dots u_{2i-1}$, $v = u_{2i}$ and $w = u_{2i+1} \dots u_{3i} \cdot s_1$, $z = u_{2i+1} \dots u_{3i} \cdot s_2$.

- Finally, if this is not the case, up to interverting x_1 and x_2 we can assume that $v_j \neq \varepsilon$ for at most i distinct $0 \leq j < 2i$. Thus, $v_j = \varepsilon$ for at least i distinct $0 \leq j < 2i$. If $w_j = \varepsilon$ for one of those j , we are back to the first case. Otherwise, it means that $|w_0 \dots w_{2i-1}| \geq i$. Now, we study the next i pairs of data words. If $v_j \neq \varepsilon$ for all $2i \leq j < 3i$, we get that $|v_0 \dots v_{3i-1}| > i$, which means that $v_0 \dots v_{3i-1} \not\leq w_0 \dots w_{3i-1}$, which again yields a critical pattern of shape (a). Otherwise, there exists some j such that $2i \leq j < 3i$ and $v_j = \varepsilon$. This is a critical pattern of shape (b), by taking $u = u_0 \dots u_{j-1}$, $v = u_j$ and $w = u_{j+1} \dots u_{3i} \cdot s_1$, $z = u_{j+1} \dots u_{3i} \cdot s_2$.

The pattern yields a witness of non-uniform continuity Conversely, assume that T has a critical pattern. Since D_1 and D_2 are co-accessible, there exists finite data words s, s_1, s_2 and t such that $D_1 \xrightarrow{s|s_1}$ and $D_2 \xrightarrow{t|s_2}$ are final runs. Let $x_n = u \cdot v \dots \mu^n(v) \cdot \mu^n(w) \cdot \mu^n(s)$ and $y_n = u \cdot v \dots \mu^n(v) \cdot \mu^n(z) \cdot \mu^n(t)$. We have, for all $n \in \mathbb{N}$, $f(x_n) = u_1 \cdot v_1 \dots \mu^n(v_1) \cdot \mu^n(w_1) \cdot \mu^n(s_1)$ and $f(y_n) = u_2 \cdot v_2 \dots \mu^n(v_2) \cdot \mu^n(w_2) \cdot \mu^n(s_2)$. Then, for all $n \in \mathbb{N}$, $|x_n \wedge y_n| \geq |u \cdot v^n| \geq |u| + n|v|$ (where $v \neq \varepsilon$).

Let $i \geq 0$ be a mismatch position in the pattern, i.e. such that

1. $u_1[i] \neq u_2[i]$, or
2. $v_2 = \varepsilon$ and $u_1[i] \neq u_2 \cdot w_2[i]$
3. $v_1 = v_2 = \varepsilon$ and $u_1 \cdot w_1[i] \neq u_2 \cdot w_2[i]$

We separately treat the different shapes of critical pattern, as μ^n might cancel the mismatch in one case:

- If $u_1 \not\leq u_2$, we have that $|f(x_n) \wedge f(y_n)| \leq i$. Then, for all $j \geq 0$, $|x_j \wedge y_j| \geq j$ but $|f(x_j) \wedge f(y_j)| \leq i$, which means that f is not uniformly continuous.
- If $v_2 = \varepsilon$ and $u_1 \not\leq u_2 \cdot w_2$, then let us show that $f(x_j)[i] \neq f(y_j)[i]$ for all but maybe one $j \in \mathbb{N}$. Let $i' = i - |u_2|$ be the position of the mismatch in w_2 . If $i' < 0$, we get that $u_1 \not\leq u_2$, and we are back to the previous case. Striving for a contradiction, assume that $u_1[i] = \mu^n(w_2[i']) = \mu^m(w_2[i'])$ for $m \neq n \in \mathbb{N}$. Then, we get that $\mu(w_2[i']) = w_2[i']$, since μ is increasing and bijective. Thus, $u_1[i] = w_2[i']$, which contradicts the fact that $u_1 \not\leq u_2 \cdot w_2$. Thus, $|f(x_j) \wedge f(y_j)| \leq i$ for all but maybe one $j \in \mathbb{N}$.
- If $v_1 = v_2 = \varepsilon$ and $u_1 \cdot w_1 \not\leq u_2 \cdot w_2$, then let $i' = i - |u_1|$, $i'' = i - |u_2|$. If either is negative, we are back to the previous case. Since $w_1[i'] \neq w_2[i'']$, this is the case for all iterations of μ , so we get that for all $j \in \mathbb{N}$, $|f(x_j) \wedge f(y_j)| \leq i$.

In all cases, we get that for all but maybe one $n \in \mathbb{N}$, $|f(x_n) \wedge f(y_n)| \leq i$. Up to taking only indices distinct from this n_0 that cancels the mismatch, we get that for all $j \geq 0$, $|x_j \wedge y_j| \geq j$ but $|f(x_j) \wedge f(y_j)| \leq i$, which means that f is not uniformly continuous. $\square$

Remark 12.7 Note that the above proof is still sound using the a priori weakest hypothesis that f is not Cauchy continuous, so Cauchy continuity is characterised by the same forbidden pattern. This yields a proof that Cauchy continuity and uniform continuity coincide for functions recognised by non-deterministic register transducers (which we already know by the more general Theorem 12.29).

12.2.4. Decision of Functionality and Continuity

Computing the Next Letter

In this section, we show how to compute the next-letter problem for non-deterministic register transducers over a decidable and representable oligomorphic data domain $\mathcal{D}$. By Theorems 11.2 and 11.3, this entails that continuity and ω -computability coincide for functions defined by NRT over $\mathcal{D}$, as stated in Theorem 12.39.

Proposition 12.38 *Let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ be a function defined by a non-deterministic register transducer over a decidable and representable oligomorphic data domain $\mathcal{D}$. Its next-letter function Next_f is computable.*

Proof. Let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ be a function defined by a non-deterministic register transducer T over a decidable and representable oligomorphic data domain $\mathcal{D}$. Given two input data words $u, v \in \mathbb{D}^*$, the next-letter problem asks to output $d \in \mathbb{D}$ such that $f(u \cdot \mathbb{D}^\omega) \subseteq v \cdot d \cdot \mathbb{D}^\omega$ if it exists, and answer No otherwise.

Solving the next-letter decision problem We first solve its decision version, or, more precisely, its complement. There is no next letter if and only if either v was already not a prefix of all $f(u \cdot x)$, i.e. $f(u \cdot \mathbb{D}^\omega) \not\subseteq v \cdot \mathbb{D}^\omega$, or if there exists $x_1, x_2 \in \mathbb{D}^\omega$ such that $f(u \cdot x_1)$ and $f(u \cdot x_2)$ mismatch right after v , i.e. $|f(u \cdot x_1) \wedge f(u \cdot x_2)| \leq |v|$.

We start by checking that $f(u \cdot \mathbb{D}^\omega) \not\subseteq v \cdot \mathbb{D}^\omega$, which is the case whenever $f(u \cdot \mathbb{D}^\omega) \cap (v \cdot \mathbb{D}^\omega)^c \neq \emptyset$. This reduces to checking emptiness of some register automaton. First, $f(u \cdot \mathbb{D}^\omega)$ is recognised by some register automaton over $\mathcal{D} \cup \text{elem}(u)$, where the elements of u are constants of the domain. Indeed, the automaton simulates T while checking that u is a prefix of the input. Similarly, $v \cdot \mathbb{D}^\omega$ is recognised by a *deterministic* register automaton over $\mathcal{D} \cup \text{elem}(v)$, which can be complemented. Their intersection is a register automaton over $\mathcal{D} \cup \text{elem}(u) \cup \text{elem}(v)$, which is oligomorphic since distinguishing constants does not affect oligomorphicity (Proposition 12.15). By Theorem 12.26, it is decidable whether such an automaton is empty.

We now check whether there exists $x_1, x_2 \in \mathbb{D}^\omega$ such that $f(u \cdot x_1)$ and $f(u \cdot x_2)$ mismatch right after v . This is equivalent to asking the existence of two runs $I_1 \xrightarrow{u|u_1} C_1 \xrightarrow{x_1|y_1} D_1$ and $I_2 \xrightarrow{u|u_2} C_2 \xrightarrow{x_2|y_2} D_1$ such that $|u_1 \cdot x_1 \wedge u_2 \cdot x_2| \leq |v|$. Finding these runs can be done with a register automaton over $\mathcal{D} \cup \text{elem}(u)$. Such an automaton has a $|v|$ -bounded counter. It guesses two runs and checks that their input has u as prefix. In parallel, it guesses a mismatch position j that it stores in its counter, and checks that the outputs indeed mismatch at position j .

Both cases are decidable, so, overall, the next-letter decision problem is decidable.

Finding the next letter If there does not exist a next letter, we are done and the algorithm simply outputs No. Otherwise, we know that such a letter exists, so it remains to exhibit it. To that end, it suffices to enumerate all data values and iteratively check whether $f(u \cdot \mathbb{D}^\omega) \subseteq v \cdot d \cdot \mathbb{D}^\omega$, which again reduces to the emptiness problem of some register automaton. A more efficient procedure is to guess some accepting run of T whose input has u as prefix, and simulate it until it produces the $|v| + 1$ -th output, which is the sought next letter. The only difficulty is that non-deterministic reassignment entails that at each step, the space of configurations to explore is infinite. However, the existence of a run only depends on the type of a configuration, so it suffices to guess the type, and corresponding data values that yield the right type. $\square$

ω -Computability and Continuity Coincide

As a direct corollary of Proposition 12.38, Theorems 11.2 and 11.3, we obtain:

Theorem 12.39 ([142, Theorems 2.14 and 2.15]) *Let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ be a function defined by a non-deterministic register transducer T over a decidable and representable oligomorphic data domain $\mathcal{D}$. Then,*

- (i) f is ω -computable iff f is continuous
- (ii) f is uniformly computable iff f is uniformly continuous
- (iii) f is m -computable iff f is m -continuous, for all $m : \mathbb{N} \rightarrow \mathbb{N}$.

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

Recall that, by Theorem 12.29, Cauchy continuity and uniform continuity coincide over oligomorphic data domains.

Deciding Functionality, Continuity and ω -Computability

We now have all the elements to decide the problems that we target. In this section, $\mathcal{D}$ denotes a decidable oligomorphic data domain. We start with functionality; the proof for the two other problems is similar.

Theorem 12.40 ([142, Theorem 3.12]) *The functionality problem for functions defined by non-deterministic register transducers over $\mathcal{D}$ is decidable. If, moreover, $\mathcal{D}$ is polynomially decidable, then the functionality problem is PSPACE-complete.*

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

Proof. Let $P(x, y, z)$ be some polynomial satisfying the properties of Proposition 12.34.

Let T be a non-deterministic register transducer with n states, k registers and maximum output length on transitions M . By Proposition 12.34, we have that T is not functional if and only if there exists two partial runs

$$\begin{aligned} I_1 &\xrightarrow{u|u_1} F_1 \xrightarrow{v|v_1} C_1 \xrightarrow{w|w_1} G_1 \\ I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} F_2 \xrightarrow{w|w_2} D_2 \end{aligned}$$

such that I_1 and I_2 are initial, F_1 and F_2 are final, $G_1 = \mu(F_1)$ and $D_2 = \mu(C_2)$ for some $\mu \in \text{Aut}(\mathbb{D})$, $u_1 \not\parallel u_2$ and $|u|, |v|, |w| \leq P(\mathcal{R}(4k), n, M)$. First, note that asking $G_1 = \mu(F_1)$ and $D_2 = \mu(C_2)$ for some $\mu \in \text{Aut}(\mathbb{D})$ is

the same as asking that $\text{type}((G_1, D_2)) = \text{type}((F_1, C_2))$, as a consequence of Proposition 12.17. Besides, since M is the maximum output length, we know that $|u_1|, |u_2| \leq M \cdot |u| \leq M \cdot P(\mathcal{R}(4k), n, M)$.

Decision We first establish decidability. Let us show that the above pattern can be recognised by a register automaton. It consists in a product automaton which recognises interleavings of marked traces of runs of T (see Proposition 10.2). It has three $P(\mathcal{R}(4k), n, M)$ -bounded counters, which checks that $|u|, |v|, |w| \leq P(\mathcal{R}(4k), n, M)$. It starts by simulating $I_1 \xrightarrow{u|u_1} F_1$ and $I_2 \xrightarrow{u|u_2} C_2$ while checking that $|u| \leq P(\mathcal{R}(4k), n, M)$, and that F_1 is final. Checking that $u_1 \not\parallel u_2$ can be done using an additional register and two $M \cdot P(\mathcal{R}(4k), n, M)$ -bounded counters. Along the run, the automaton also maintains the type of the current pair of configurations of the two runs of T that it simulates. At some point where the configuration of the first run is final, it guesses that it corresponds to (F_1, C_2) and memorises its type $\text{type}((F_1, C_2))$. The next fragment of $F_1 \xrightarrow{v|v_1} C_1$ and $C_2 \xrightarrow{v|v_2} F_2$ is easier to simulate, as it suffices to check that $|v| \leq P(\mathcal{R}(4k), n, M)$ and that F_2 is final. Finally, the last part of the pattern $C_1 \xrightarrow{w|w_1} \mu(F_1)$ and $F_2 \xrightarrow{w|w_2} \mu(C_2)$ is checked as follows: using its last counter, the automaton checks that $|w| \leq P(\mathcal{R}(4k), n, M)$. If it reaches a pair (G_1, D_2) of configurations such that $\text{type}((G_1, D_2)) = \text{type}((F_1, C_2))$, it accepts. Note that the type of a configuration includes its state, so $\text{type}(G_1) = \text{type}(F_1)$ implies that G_1 is final. Finally, if its counter exceeds its maximal bound, i.e. $|w| > P(\mathcal{R}(4k), n, M)$, the automaton rejects. Overall, the above automaton is non-empty if and only if T has the pattern characterising non-functionality, so functionality is decidable as it reduces to checking emptiness of this automaton.

Complexity upper bound for polynomially decidable data domains

We now need to show that the problem is in PSPACE if $\mathcal{D}$ is polynomially decidable. The above register automaton is certainly of a size exponential in the size of T , since it stores in its states counters that are bounded by $P(\mathcal{R}(4k), n, M)$, where $\mathcal{R}(4k)$ is exponential in k . However, it can be represented in polynomial space: its states can be stored in polynomial space, since the values of the counters can be stored in binary. The labels of its transitions can also be represented using polynomial space, since they belong to the same set as those of T , plus tests of the form $\star = r$ to test that the output is correct. Then, given two states p and q and the label (ϕ, regOp) of a transition, one can decide in polynomial space whether there exists a transition between p and q labelled by (ϕ, regOp) . First, if it consists in simulating a transition of T , this is immediate as it suffices to check the presence of such transition in the transition table of T . Updating the type of the current pair of configurations is also doable in polynomial space, since we assumed that FO satisfiability is in PSPACE over $\mathcal{D}$. Finally, updating the counters is also doable in polynomial space, as it only consists in increments and equality tests. An analysis of the algorithm of Theorem 12.26 establishes that the algorithm for checking emptiness runs in polynomial space given such a representation of its input (see Section B.2.2 for details). This allows to conclude, since $\text{NPSPACE} = \text{PSPACE}$ [152, Theorem 1].

Recall that $k = |R| \leq |T|$, so $\mathcal{R}(4k)$ is singly exponential, and not doubly exponential.

The reader can refer to Section B.2.1 for a more detailed discussion on the way to represent automata.

[152]: Savitch (1970), ‘Relationships Between Nondeterministic and Deterministic Tape Complexities’

Complexity lower bound Finally, hardness is obtained by reducing from the functionality problem of register automata over $(\mathbb{D}, =)$, which is PSPACE-complete Theorem 12.8. Indeed, by definition any data domain contains at least the equality predicate, so non-deterministic register transducers over $(\mathbb{D}, =)$ can be simulated by NRT over any oligomorphic data domain. $\square$

Using similar arguments, one can establish an analogous statement for the continuity and uniform continuity problem.

Theorem 12.41 ([142, Theorem 3.12]) *The continuity and uniform continuity problem for functions defined by non-deterministic register transducers over $\mathcal{D}$ is decidable. If, moreover, $\mathcal{D}$ is polynomially decidable, then the functionality problem is PSPACE-complete.*

If $\mathcal{D}$ is representable, ω -computability and uniform computability are decidable, and PSPACE-complete for polynomially decidable data domains.

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

Proof. The main idea is the same, as it consists in showing that the pattern that characterises non-continuity is recognisable by a register automaton, that can moreover be described in polynomial space when the data domain is polynomially decidable.

Let T be a functional non-deterministic register transducer with n states and k registers, with maximum output length M , that recognises some partial function $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$.

By Proposition 12.37, f is not uniformly continuous if and only if T has a critical pattern

$$\begin{aligned} I_1 &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{w|w_1} D_1 \\ I_2 &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{z|z_2} D_2 \end{aligned}$$

Moreover, by Proposition 12.36 it is not continuous if additionally C_1 is final.

Let us describe the construction of a register automaton that recognises the above patterns. We treat the case of uniform continuity; checking that a configuration is final can be done by adding one bit of information to the states. First, checking that we have $I_1 \xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1)$ and $I_2 \xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2)$, where $|u|, |v| \leq P(\mathcal{R}(4k), n, M)$ and $u_1 \not\parallel u_2$ is done as for functionality. It remains to check that $\mu(C_1) \xrightarrow{w|w_1} D_1$ and $\mu(C_2) \xrightarrow{w|w_2} D_2$, where $|w| \leq P(\mathcal{R}(4k), n, M)$ and D_1 and D_2 are each co-reachable. The only remaining difficulty is to test co-reachability. By a simple pumping argument, one can show that D_1 is co-reachable if and only if there exists some configuration F_1 and some morphism $v \in \text{Aut}(\mathbb{D})$ such that $D_1 \xrightarrow{x|y} F_1 \xrightarrow{z|t} \mu(F_1)$, with $|x|, |z| \leq n\mathcal{R}(k)$. This property can again be checked by our register automaton, using an additional $n\mathcal{R}(k)$ -bounded counter, and memorising the type of F_1 , and similarly for D_2 . Membership of PSPACE results from the same complexity analysis as for functionality, as the additional counters and memory does not

change the fact that the automaton can be described in space that is polynomial in the size of the input transducer.

PSPACE -hardness is again obtained by reducing to the case of $(\mathbb{D}, =, \mathfrak{C})$, which is PSPACE -complete by Theorem 12.12.

Finally, the statement about ω -computability and uniform computability is a direct consequence of Theorem 12.39. $\square$

As a consequence, we get PSPACE -completeness for all the problems we consider over data domain $(\mathbb{Q}, <, 0)$. Indeed, such a domain is representable, as explained in Example 11.1, and polynomially decidable (Example 12.4).

Theorem 12.42 ([142, Theorem 3.13]) *For relations given by non-deterministic register transducers over $(\mathbb{Q}, <)$, the problems of deciding functionality, continuity/ ω -computability and uniform continuity/uniform computability are PSPACE -complete.*

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

12.3. The Case of a Discrete Order

We have seen in Example 12.3 that $(\mathbb{N}, <, 0)$ is not oligomorphic, as its group of automorphisms is the trivial group $\{\text{id}_{\mathbb{N}}\}$, so each element is its own orbit. Neither is $(\mathbb{Z}, <)$, whose automorphisms are translations: an orbit over $\mathbb{Z}^2$ is characterised by the distance between each element of the pair, which can take any value in $\mathbb{Z}$. As a consequence, over these data domains, we cannot apply the reasonings of Section 12.2, which heavily rely on the notion of orbit loop, which is defined as a pair of configurations that occur along a run and which belong to the same orbit. In particular, the argument of Proposition 12.36 to exhibit a critical pattern breaks, since there are infinitely many orbits of configurations and we might not be able to find orbit loops. Neither can we get small mismatch witnesses using Lemma 12.33, which relies on finding and removing orbit loops.

The notion of orbit loop is not defined in Section 12.2, since we instead present the patterns using morphisms. For the present argument, it is however more convenient to make this notion explicit and use a loop terminology.

In this section, we study the case of $(\mathbb{N}, <, 0)$ and conclude the part by explaining how to transfer it to $(\mathbb{Z}, <)$ and to $(\mathbb{Z}, <, \mathfrak{C})$ where $\mathfrak{C}$ is a finite set of distinguished constants that are interpreted as elements of $\mathbb{Z}$.

Remark 12.8 We need to have access to the constant 0, as it is the minimal element of $\mathbb{N}$, and thus plays a special role.

Assumption 12.10 We assume that registers are initialised with $\# = 0$.

Observation 12.43 For all morphisms $\mu \in \text{Aut}(\mathbb{Q}_+)$, we have $\mu(C^i) = C^i$, since they preserve the constant 0.

12.3.1. Non-Deterministic Register Automata over $(\mathbb{N}, <, 0)$

$(\mathbb{N}, <, 0)$ as a substructure of $(\mathbb{Q}_+, <, 0)$

Notation 12.11 In the following, $\mathbb{Q}_+$ denotes the set of non-negative rational numbers.

$$\mathbb{Q}_+ = \{q \in \mathbb{Q} \mid q \geq 0\}$$

To lighten the notations, we sometimes simply write $\mathbb{N}$ to denote the data domain $(\mathbb{N}, <, 0)$, and $\mathbb{Q}_+$ for $(\mathbb{Q}_+, <, 0)$. No ambiguity arises, as in the argument, those respective sets are only considered with the structure of their linear ordering plus the constant 0.

Fact 12.1 We abundantly use the fact that $(\mathbb{N}, <, 0)$ is a substructure of $(\mathbb{Q}_+, <, 0)$, which is oligomorphic. In particular, any register automaton A over $(\mathbb{N}, <, 0)$ is also a register automaton over $(\mathbb{Q}_+, <, 0)$. When there is an ambiguity, we denote $A_{\mathbb{N}}$ the automaton as seen over $\mathbb{N}$ and $A_{\mathbb{Q}_+}$ when interpreted over $\mathbb{Q}_+$. When the focus is set over the language they recognise, we write, for $\mathbb{D} = \mathbb{N}, \mathbb{Q}_+$, $L_{\mathbb{D}}(A) = L(A_{\mathbb{D}}) = \{w \in \mathbb{D}^\omega \mid \text{there is a accepting run of } A \text{ over } w\}$. The notation is extended to transducers in the expected way, e.g. $T_{\mathbb{Q}_+}$ denotes T operating over $(\mathbb{Q}_+, <, 0)$.

Example 12.5 The register automaton of Figure 12.6a is non-empty in $(\mathbb{Q}_+, <)$, as $1 \cdot \frac{1}{2} \cdot \dots \cdot \frac{1}{n} \dots$ is accepted. However, it is empty in $(\mathbb{N}, <)$, since its only run consists in an infinite descending chain, which does not exist in $\mathbb{N}$.

Similarly, in Figure 12.6b, the register automaton initially guesses some upper bound B which it stores in r_M , then it asks to see an infinite increasing chain which is bounded from above by B . This is possible in $(\mathbb{Q}_+, <)$, but not in $(\mathbb{N}, <)$.

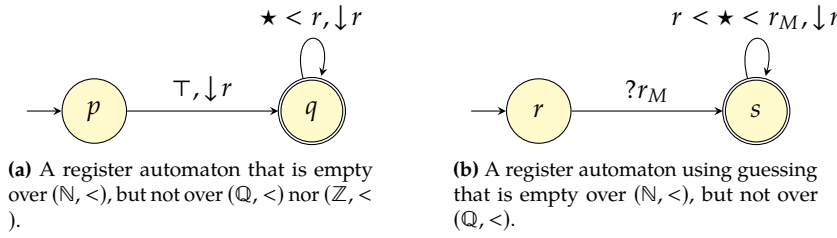


Figure 12.6.: Two register automata that are empty over $(\mathbb{N}, <)$, but not over $(\mathbb{Q}, <)$.

One can check that the self-transitions respectively on states q and s induce an orbit loop in $(\mathbb{Q}, <)$ for each automaton. It does not in $(\mathbb{N}, <)$, since then the only loops (in the sense developed in Section 12.2.3) are *actual* loops, i.e. configurations that repeat (not up to automorphism).

Note that the automaton of Figure 12.6a is not empty in $(\mathbb{Z}, <)$, since it accepts e.g. $0 \cdot (-1) \cdot (-2) \dots$. As a matter of fact, the self-transition of state q induces a loop in $(\mathbb{Z}, <)$, as one can from any data value of $\mathbb{Z}$ to another with a translation.

Be careful however, as, unlike for oligomorphic domains (Proposition 12.31), the absence or presence of an orbit loop is not a witness of (non-)emptiness here. For instance, equip the first automaton (Figure 12.6a) with an additional register which stores a value that is constant along the run, and is not tested (cf Figure 12.7). It does not contain any oligomorphic loop, and still it is not empty over $(\mathbb{Z}, <)$.

We say *emphself*-transition instead of *self-loop* to highlight the fact that this is not a loop in $\mathbb{N}$.

This observation is even more obvious in $(\mathbb{N}, <, 0)$.

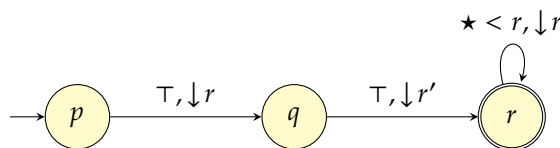


Figure 12.7.: A register automaton that is not empty over $(\mathbb{Z}, <)$ but does not have any orbit loop.

These observations call for a more precise examination of what it means to be a loop in $(\mathbb{N}, <, 0)$, i.e. at which conditions a path can be taken an infinite number of times. As we demonstrate, this is characterised by a criterion on the order between the registers before and after taking the path (see Definition 12.16 and Proposition 12.55). To characterise continuity in the presence of guessing, we also need the weaker notion of a *quasi-loop*, that encompasses paths that can be taken arbitrarily many times, over finite inputs which are increasing for the prefix order (see Section 12.3.5). For instance, the self-transition in Figure 12.6b is a quasi-loop: it suffices to guess bigger and bigger values of the initial upper bound. However, no run can contain infinitely many occurrences of such a transition, since the value that is initially guessed on the first transition of the run sets a bound on the number of times the transition can be taken. This means that it is not a loop.

In contrast, the self-transition of state q in Figure 12.6a is not even a quasi-loop: the first letter in the input sets a bound on the number of times it can be taken.

As we demonstrate, in the absence of non-deterministic reassignment, both notions coincide.

Partial Runs over $(\mathbb{N}, <, 0)$ and $(\mathbb{Q}_+, <, 0)$

Before moving to the study of loops, observe that partial runs in $(\mathbb{Q}, <, 0)$ and $(\mathbb{N}, <, 0)$ coincide, in the sense that one can always ‘stretch’ the intervals between the values in $\mathbb{Q}$ to obtain values in $\mathbb{N}$ by multiplying by a large enough constant.

Lemma 12.44 (Stretching lemma) *Let A be a register automaton over $(\mathbb{Q}_+, <, 0)$ with registers R . If $(p, \nu) \xrightarrow{u} (q, \lambda)$ for some configurations $(p, \nu), (q, \lambda) \in \text{Configs}(A)$ and some finite data word $u \in (\mathbb{Q}_+)^*$ with $u \neq \varepsilon$, then we have $(p, \nu') \xrightarrow{v} (q, \lambda')$ for some valuations $\nu', \lambda' : R \rightarrow \mathbb{N}$ and some $v \in \mathbb{N}^+$. Moreover, if C is initial, then we can assume that C' is also initial.*

Proof. The proof simply consists in multiplying all data value by the product (or even some common multiple) of their denominators. Formally, write $\nu(R) \cup \lambda(R) \cup \text{elem}(u) = \{\frac{a_l}{b_l}, \dots, \frac{a_l}{b_l}\}$ (for $l = |\nu(R) \cup \lambda(R) \cup \text{elem}(u)|$). By letting $K = \text{lcm}\{b_i \mid 1 \leq i \leq l\}$, we have, by letting $\mu : x \in \mathbb{Q}_+ \mapsto Kx$, that $\mu \in \text{Aut}(\mathbb{Q}_+)$, so $\mu(C) \xrightarrow{\mu(u)} \mu(D)$, which yields the expected result as, by construction, all elements now take their values in $\mathbb{N}$. Moreover, ν_R^t is preserved by all automorphisms of $\text{Aut}(\mathbb{Q}_+)$, since they preserve 0, so initial configurations are preserved. $\square$

Notation 12.12 In the following, for $K \in \mathbb{N} \setminus \{0\}$ we denote, by a slight abuse of notation, $K : x \mapsto K \cdot x$ the morphism of $(\mathbb{Q}_+, <, 0)$ that consists in multiplying by K . No ambiguity should arise due to typing considerations.

As a consequence, we have that:

Corollary 12.45 *Let A be a register automaton over data domain $(\mathbb{Q}_+, <, 0)$, that operates over finite data words. Then, $L_{\mathbb{N}}(A) \neq \emptyset$ if and only if $L_{\mathbb{Q}_+}(A) \neq \emptyset$.*

A similar study could be conducted for $(\mathbb{Z}, <)$, in which $(\star < r, \downarrow r)$ is a loop. Instead of redoing all the work, we show how to extend the study directly in Section 12.3.6.

Quasi-looping is characterised by a similar yet weaker criterion as for loops.

Recall that runs are closed under automorphisms, see Proposition 12.13.

Proof. By the above proposition, any accepting run in $\mathbb{Q}_+$ induces an accepting run in $\mathbb{N}$, by multiplying all data values by a large enough factor. $\square$

By specialising Theorem 12.24 to finite data words, we also get the following, that was observed in [39, Theorem 2]:

Corollary 12.46 ([39]) *Let A be a register automaton over $(\mathbb{N}, <, 0)$, that operates over finite data words. Then, $\text{lab}(L_{\mathbb{N}}(A))$ is a regular language.*

Since register automata over data ω -words can simulate register automata over finite data words, we have (this is also a consequence of [39, Theorem 14]):

Theorem 12.47 ([39]) *The emptiness problem for non-deterministic register automata over finite data words over $(\mathbb{N}, <, 0)$ is PSPACE-complete.*

Proof. Membership of PSPACE is obtained by adding self-transitions to the final states. PSPACE-hardness results from the fact that the emptiness problem is PSPACE-hard, already for register automata over finite data words with equality only [28, Theorem 5.1].

We provide a few additional details for membership of PSPACE: given a register automaton $A = (Q, I, F, \Delta)$ over finite data words with data domain $(\mathbb{N}, <, 0)$, construct a register automaton over data ω -words in $\mathbb{Q}_+$: $A' = (Q, I, F, \Delta')$, where $\Delta' = \Delta \cup \{q_f \xrightarrow{\top, \text{keep}^R} q_f \mid q_f \in F\}$. If $L_{\mathbb{N}}(A) \neq \emptyset$, one can easily construct an accepting run in A' with prefix $w \in L_{\mathbb{N}}(A)$. Conversely, if $L_{\mathbb{Q}_+}(A') \neq \emptyset$, it means in particular that some final state $q_f \in F$ is reachable by reading some finite data word $w \in (\mathbb{Q}_+)^*$. From w , one obtains some $w' \in \mathbb{N}^*$ by multiplying by a large enough factor, as in Lemma 12.44; such a w' allows to reach q_f in A . Finally, the size of A' is linear in $|A|$, so checking that $L_{\mathbb{Q}_+}(A') \neq \emptyset$ can be done using a space polynomial in $|A|$ by Theorem 12.27. $\square$

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

We could also get the result equally easily from Corollary 12.46.

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

Widening Types and Looping Configurations in $(\mathbb{N}, <, 0)$

In this section, we study the conditions under which a sequence of transitions constitutes a loop, i.e. can be taken infinitely many times to yield a run over $(\mathbb{N}, <, 0)$. As witnessed by Example 12.5, it requires more conditions than over $(\mathbb{Q}_+, <, 0)$, as $\mathbb{N}$ forbids to take infinitely many values in a finite interval. More precisely, it forbids infinite decreasing chains, as well as infinite increasing chains that have an upper bound. Correspondingly, if a sequence of transitions is to be loop, it has to affect the relative order between registers in a way that does not induce such chains when iterated. This necessary condition is captured by the notion of widening type, that is defined for types of the superstructure $(\mathbb{Q}_+, <, 0)$. As we demonstrate, this condition is sufficient: a widening type necessarily contains a pair of valuations such that one can be mapped to the other through a morphism that preserves $\mathbb{N}$. Starting from a configuration that take its values in $\mathbb{N}$, one can then iteratively apply this morphism in the same way as loops are iterated in oligomorphic domains, hence obtaining an infinite run.

Types of Configurations The relative order between registers plays a key role. As explained in Example 12.3, types in $(\mathbb{Q}_+, <, 0)$ are characterised by this information. Given a valuation $v : R \rightarrow \mathbb{N}$, we write $\text{type}(v)$ instead of $\text{type}_{(\mathbb{Q}_+, <, 0)}(v)$ (we rely on the fact that $(\mathbb{N}, <, 0)$ is a substructure of $(\mathbb{Q}_+, <, 0)$, see Fact 12.1). No ambiguity arises, since $\text{type}_{(\mathbb{N}, <, 0)}(v)$ is irrelevant here: there are infinitely many such types. Similarly, when we mention some orbit, we refer to the orbit in $(\mathbb{Q}_+, <, 0)$. Given a valuation $v : R \rightarrow \mathbb{N}$, its type (in $\mathbb{Q}_+$) is equivalent to

$$\text{type}(v) \equiv \bigwedge_{\bowtie \in \{<, >, =\}} \bigwedge_{\substack{r, s \in R \\ v(r) \bowtie v(s)}} r \bowtie s \wedge \bigwedge_{\substack{r \in R \\ v(r) = 0}} r = 0 \quad (12.2)$$

This is a more generally true for valuations over $\mathbb{Q}_+$, and even $\mathbb{Q}$, but we want to insist on $\mathbb{N}$ here.

This notation is extended to configurations by letting $\text{type}((q, v)) = (q, \text{type}(v))$.

Observation 12.48 Let A be a register automaton. For any configuration $C \in \text{Configs}(A)$, if $\text{type}(C) = \text{type}(C')$, then $C = C'$, since it imposes that all registers are valued 0.

Widening Types and Valuations Let us now introduce the central notion of this section. It concerns the type of *pairs* of valuations, and phrases conditions about the relative order of registers between them. Informally, a type is widening if it prevents intervals between registers from shrinking. Otherwise, one would need to put infinitely many values in a finite interval when iterating the loop by repeatedly applying some morphism. Note that widening is to be understood in a ‘greater than or equal to’ sense, as intervals do not have to *strictly* increase.

Definition 12.13 (Widening type) Let $R = \{r_1, \dots, r_k\}$ be a set of registers, and $R' = \{r'_1, \dots, r'_k\}$ be its primed copy. Let $\sigma(r_1, \dots, r_k, r'_1, \dots, r'_k)$ denote some type of $2k$ -tuples. We say that it *preserves types* when $\sigma(v, \lambda) \Rightarrow \text{type}(v) = \text{type}(\lambda)$ for all $v, \lambda : R \rightarrow \mathbb{Q}_+$. We further say that σ is *widening* if

The designation of ‘widening type’ is justified by Lemma 12.49, which makes the link with widening valuations.

To be clear: a widening type is required to preserve types, by definition.

- (a) The value of each register increases between v and λ : for all $r \in R$, $\sigma \Rightarrow r \leq r'$ is valid in $(\mathbb{Q}_+, <, 0)$, i.e. all valuations $v, \lambda : R \rightarrow \mathbb{Q}_+$ that satisfy σ are such that $v(r) \leq \lambda(r)$, for all $r \in R$.
- (b) If a given register is constant between v and λ , then all registers below are also constant: for all $r, s \in R$, if $\sigma \Rightarrow s = s'$ and $\sigma \Rightarrow r \leq s$ are valid in $\mathbb{Q}_+$, then $\sigma \Rightarrow r = r'$ is valid in $\mathbb{Q}_+$.

The first item is meant to forbid infinite descending chains, while the second targets infinite increasing chains that are bounded from above.

By extension, for two valuations v and λ over R , we say that (v, λ) is *pseudo-widening* if $\text{type}(v, \lambda)$ is widening.

Note that since widening types preserve types, this a pseudo-widening pair of valuations necessarily satisfies $\text{type}(v) = \text{type}(\lambda)$.

We say that (v, λ) is *widening* when $\text{type}(v) = \text{type}(\lambda)$ and:

- 1. for all $r \in R$, $\lambda(r) \geq v(r)$
- 2. for all $r, s \in R$, $|\lambda(s) - \lambda(r)| \geq |v(s) - v(r)|$

Observe that a widening pair of valuations is in particular pseudo-widening.

Remark 12.9 We differentiate between pseudo-widening and widening pairs of valuations because two valuations might be pseudo-widening while it is not possible to obtain a loop from them. For instance, let

$v : r \mapsto 1, s \mapsto 4$ and $\lambda : r \mapsto 3, s \mapsto 5$, and assume that $v \xrightarrow[A]{u} \lambda$ for some $u \in \mathbb{N}$. $\text{type}(v, \lambda)$ is widening, and yet, the interval between r and s strictly decreases. As a consequence, there cannot exist a morphism $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ and that maps v to λ , which would be a sufficient condition for a loop to exist by taking $u \cdot \zeta(u) \cdot \zeta^2(u) \dots$

Widening Types Contain Widening Valuations We now demonstrate that from a pseudo-widening pair of valuations, it is in fact always possible to ‘widen’ the target valuation to get a widening pair of valuations. In other words, if a widening type is inhabited by a pair of valuations, then it is inhabited by some widening pair of valuations, which justifies the name of *widening type*.

Lemma 12.49 *Let $R = \{r_1, \dots, r_k\}$, and let $\sigma(r_1, \dots, r_k, r'_1, \dots, r'_k)$ be a $2k$ -type that is widening. Then, there exists two valuations $v, \lambda : R \rightarrow \mathbb{Q}_+$ such that $\text{type}(v, \lambda) = \sigma$ and (v, λ) is widening.*

Proof. Let $R = \{r_1, \dots, r_k\}$, and let $\sigma(r_1, \dots, r_k, r'_1, \dots, r'_k)$ be a $2k$ -type that preserves types.

Given a pair of valuations (v, λ) with $v, \lambda : R \rightarrow \mathbb{Q}_+$, we say that a pair of registers $r \neq s \in R$ is *shrinking* in (v, λ) if $|\lambda(s) - \lambda(r)| < |v(s) - v(r)|$.

We show that, from a pair of valuations of type σ that has a pair of shrinking registers, one can obtain a pair of valuations with one less pair of shrinking registers, again of type σ . For conciseness, we formulate it as a reasoning by contradiction.

Since σ is a type, we know that it is satisfiable. Let (v, λ) be a pair of valuations such that $\text{type}(v, \lambda) = \sigma$ and that has a minimal number of shrinking pairs of registers. First, σ is widening, so we have that for all $r \in R$, $\sigma \Rightarrow r' \geq r$, which means that $\lambda(r) \geq v(r)$. Aiming for a contradiction, suppose that (v, λ) has a pair (r, s) of shrinking registers. Up to swapping r and s , assume that $v(s) > v(r)$. σ preserves types, which means that $\text{type}(\lambda) = \text{type}(v)$, so $\lambda(s) > \lambda(r)$. Since σ is widening, we have $\lambda(s) \geq v(s)$. Moreover, we cannot have $\lambda(s) = v(s)$, otherwise, since $v(r) < v(s)$, we would get that $\sigma \Rightarrow r = r'$, i.e; $v(r) = \lambda(r)$ by item(b) of the definition of a widening type. This would contradict the fact that (r, s) is shrinking. Thus, $\lambda(s) > v(s)$. The schema is depicted in Figure 12.8 (where we just do not know whether $\lambda(r) \geq v(s)$). Our goal is to lift $\lambda(s)$ so that (r, s) is not shrinking anymore, while not inducing new shrinking intervals. To that end, we apply a morphism that is non-contracting, that does not modify the values of $v(s)$ nor of $\lambda(r)$, and that lifts $\lambda(s)$ high enough. Thus, choose B such that $v(s) \leq B < \lambda(s)$ and $\lambda(r) \leq B < \lambda(s)$, and pick M such that $M - \lambda(r) \geq v(s) - v(r)$.

Consider the morphism μ of $(\mathbb{Q}_+, <, 0)$ that sends the interval $[0; B]$ to $[0; B]$, $[B; \lambda(s)]$ to $[B; M]$ and $[\lambda(s); +\infty[$ to $[M; +\infty[$ in the following

$$\text{way: } \begin{cases} x \in [0; B] & \mapsto x \\ x \in [B; \lambda(s)] & \mapsto v(s) + \frac{M-B}{\lambda(s)-B}(x-B) \\ x \in [\lambda(s); +\infty[& \mapsto x + M - \lambda(s) \end{cases} \text{ . Apply } \mu \text{ to } (v, \lambda);$$

this yields some pair $(v', \lambda') = (\mu(v), \mu(\lambda))$. One can check that if a pair of registers was not shrinking in (v, λ) , it is not shrinking in (v', λ') . Moreover, we have that $\lambda'(s) - \lambda'(r) = M - \lambda(r)$, and $v'(s) - v'(r) = v(s) - v(r)$, so $\lambda'(s) - \lambda'(r) \geq v'(s) - v'(r)$: (r, s) is not shrinking in (v', λ') .

Note that the construction could be done by induction, by reasoning on the number of remaining shrinking intervals.

In the following, for any formula φ we simply write $\sigma \Rightarrow \varphi$ to say that $\sigma \Rightarrow \varphi$ is valid in $(\mathbb{Q}_+, <, 0)$.

We cannot have $v(s) = v(r)$, otherwise (r, s) is not shrinking.

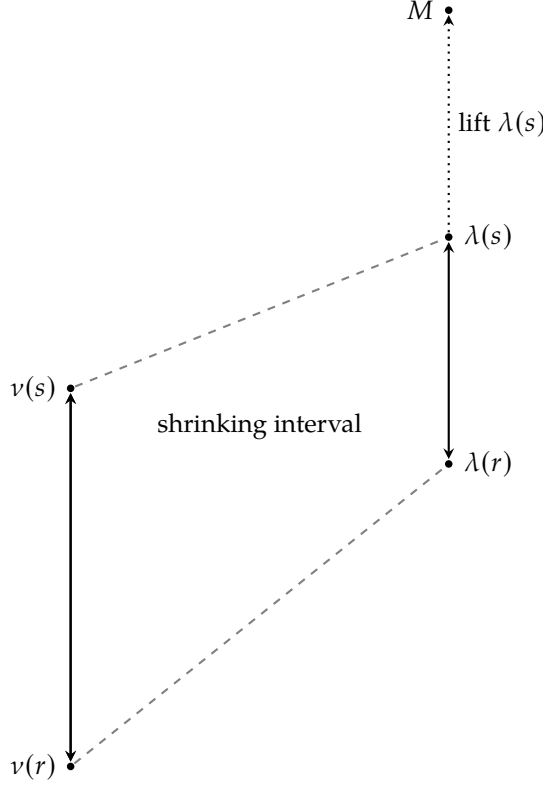


Figure 12.8. A schema of the relative order between values. It can be that $\lambda(r) > \lambda(s)$.

As a consequence, (v', λ') has strictly less shrinking pairs of registers, which yields the sought contradiction.

Overall, we get that (v, λ) is such that $\text{type}(v, \lambda) = \sigma$ and it has no shrinking pairs. In other words, for all $r, s \in R$, $|\lambda(s) - \lambda(r)| \geq |v(s) - v(r)|$. Moreover, since σ is widening, we get by item (a) that $\lambda(r) \geq v(r)$ for all $r \in R$. As a consequence, (v, λ) is widening, which concludes the proof. $\square$

Loops in $\mathbb{N}$ Let us formalise what we mean by a loop in $\mathbb{N}$. It relies on the notion of a morphism that preserves $\mathbb{N}$. We say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ preserves $\mathbb{N}$ if $f(\mathbb{N}) \subseteq \mathbb{N}$, i.e. for all $n \in \mathbb{N}$, $f(n) \in \mathbb{N}$. We later on make a connection with the notion of widening type.

Notation 12.14 In the following, we need to make a distinction between valuations or configurations that take their values in $\mathbb{N}$ versus those that take their values in $\mathbb{Q}_+$. To that end, we sometimes say for short that a valuation v over registers R is an $\mathbb{N}$ -configuration if $v : R \rightarrow \mathbb{N}$, and that it is a $\mathbb{Q}_+$ -configuration if $v : R \rightarrow \mathbb{Q}_+$.

Definition 12.15 (Loop in $\mathbb{N}$) Let R be some finite set of registers and let $v, \lambda : R \rightarrow \mathbb{N}$ be two valuations. We say that the pair (v, λ) is *looping* if there exists a morphism $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ and such that $\zeta(v) = \lambda$.

Let A be a register automaton over $(\mathbb{N}, <, 0)$, and let $\rho : (p, v) \xrightarrow{u} (q, \lambda)$ be a partial run of A , where $u \neq \varepsilon$. We say that ρ is a *loop in $\mathbb{N}$* if $p = q$ and, moreover, (v, λ) is looping. We also say that $((p, v), (p, \lambda))$ is a *looping pair of configurations in $\mathbb{N}$* .

Conceptually, ζ plays a different role than morphisms that send an element of an orbit to another. To distinguish between the two, we use the letter ζ instead of μ , although both objects are elements of $\text{Aut}(\mathbb{Q}_+)$.

Since A is over data domain $(\mathbb{N}, <, 0)$, the valuations of C and D take their values in $\mathbb{N}$, and $u \in \mathbb{N}^+$.

Let us first observe that a loop in $\mathbb{N}$ can indeed be iterated:

Observation 12.50 Let A be a register automaton, and let $C \xrightarrow[A]{\rho} D$ be a loop in $\mathbb{N}$. Then, there exists $x \in \mathbb{N}^\omega$ such that $C \xrightarrow[A]{\rho^\omega} x$. Moreover, if ρ contains a final configuration, then the run ρ^ω over x is final.

Proof. Since ρ is a loop in $\mathbb{N}$, there exists $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ such that $D = \zeta(C)$. As a consequence, $C \xrightarrow[A]{\rho} \zeta(C) \xrightarrow[A]{\rho} \zeta^2(C) \xrightarrow[A]{\rho} \dots$ is a run of A over $x = u \cdot \zeta(u) \cdot \zeta^2(u) \dots$. Moreover, as applying a morphism does not change the state of a configuration, if a final configuration appears in ρ then it also appears in all the iterations of ρ under ζ . Thus, in that case, ρ^ω contains infinitely many occurrences of a final configuration, so it is final. $\square$

Strictly speaking, here, ρ denotes the sequence of transitions between C and D , and not the run. Indeed, configurations are not taken into account, since they do not necessarily repeat (although they do up to automorphism).

Looping amounts to Widening (and Conversely) We now show that the notion of looping pair actually coincides with that of widening pair. The construction is merely an exercise: since the interval between each register value is widening, it is possible to pick images for elements in $\mathbb{N}$ that themselves belong to $\mathbb{N}$; conversely, an increasing and bijective function that preserves $\mathbb{N}$ necessarily increases the size of integer intervals.

Lemma 12.51 Let $\nu, \lambda : R \rightarrow \mathbb{N}$ be a pair of valuations in $\mathbb{N}$ for some finite set of registers R . (ν, λ) is looping if and only if it is widening.

Proof. Let $\nu, \lambda : R \rightarrow \mathbb{N}$ for some finite set of registers R . First, assume that (ν, λ) is looping, i.e. there exists $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ and such that $\zeta(\nu) = \lambda$. A morphism that preserves $\mathbb{N}$ necessarily widens integer intervals, i.e. for all $m, n \in \mathbb{N}$, $|\zeta(n) - \zeta(m)| \geq |n - m|$ (see Lemma C.1 and its proof in the appendix). Let $r, s \in R$. By taking $m = \nu(r)$ and $n = 0$, we get that $\lambda(r) \geq \nu(r)$, which is item 1 of the definition. Item 2 is obtained by taking $m = \nu(r)$ and $n = \nu(s)$.

Conversely, assume that (ν, λ) is widening. The construction relies on the observation that given two intervals, there exists a monotone bijection in $\mathbb{Q}_+$ that sends the smallest to the biggest while preserving $\mathbb{N}$, as illustrated in Figure 12.9 on the next page. Given $a < b, c < d$ such that $d - c > b - a$, one sends $[a; b - 1]$ to $[c; c + (b - 1 - a)]$ using a translation $x \mapsto x + c - a$. Then, $[b - 1; b]$ is bijectively sent to $[c + (b - 1 - a); d]$; the choice of the mapping does not matter as there are no integer values to preserve, except for the bounds of the intervals. If the intervals have the same size, $\text{id}_{\mathbb{Q}_+}$ does the deed.

The result is then obtained by applying this construction to each interval between successive registers in ν . Since $\text{type}(\nu) = \text{type}(\lambda)$, we know that the order between registers is the same in λ and in ν , so registers that are successive in ν are also successive in λ .

Two registers r and s are successive in ν if $\nu(s) \geq \nu(r)$ and there does not exist $t \in R$ such that $\nu(s) > \nu(t) > \nu(r)$.

Formally, this is done by induction on the number of registers. We write the details for completeness: let us show by induction on $|R|$ that if a pair of valuations $\nu, \lambda : R \rightarrow \mathbb{N}$ is widening, then there exists a morphism $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ such that $\zeta(\nu) = \lambda$. First, if $R = \{r\}$ is a singleton, (ν, λ) is widening if and only if $\lambda(r) \geq \nu(r)$. Then, take

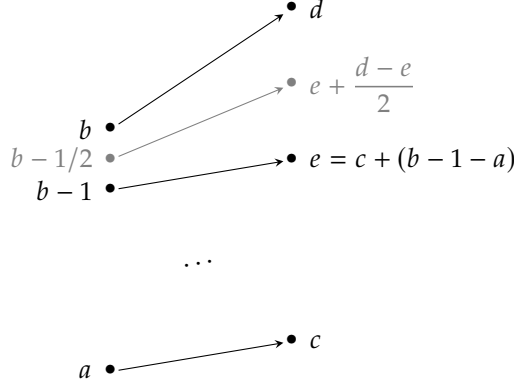


Figure 12.9.: How to send an interval $[a; b]$ to a strictly bigger one $[c; d]$ while preserving $\mathbb{N}$.

some morphism $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that maps $[0; \nu(r)]$ to $[0; \lambda(r)]$ in the way that is explained above, and which sends $[\nu(r); +\infty[$ to $[\lambda(r); +\infty[$ with $x \mapsto x + \lambda(r) - \nu(r)$. Such a morphism preserves $\mathbb{N}$.

Now let $i \geq 1$, and assume the results holds for all R of size $|R| = i$. Let R be some set of registers of size $i + 1$, and let (ν, λ) be a pair of valuations over R that is widening. Let $s \in R$ be such that $\nu(s)$ is the maximum value of ν . Since $\text{type}(\nu) = \text{type}(\lambda)$, we know that $\lambda(s)$ is the maximum value of λ . Let $R' = R \setminus \{s\}$. Observe that $(\nu|_{R'}, \lambda|_{R'})$ is again widening. By the induction hypothesis, there exists $\zeta \in \text{Aut}(\mathbb{Q}_+)$ such that $\zeta(\nu|_{R'}) = \lambda|_{R'}$ and $\zeta(\mathbb{N}) \subseteq \mathbb{N}$. There are two cases:

- If $\nu(s) = \nu(r)$ for some $r \neq s$, then we also have $\lambda(s) = \lambda(r)$. Thus, $\zeta(\nu) = \zeta(\lambda)$, and taking the same ζ is suitable for (ν, λ) .
- $\nu(s) > \nu(r)$ for all $r \neq s$. Take the maximum such r , i.e. $\nu(s) > \nu(r) \geq \nu(t)$ for all $t \neq r, s$. Again, since $\text{type}(\nu) = \text{type}(\lambda)$, we also have that $\lambda(s) > \lambda(r) \geq \lambda(t)$. Besides, as (ν, λ) is widening, we know that $\lambda(s) - \lambda(r) \geq \nu(s) - \nu(r)$. By the reasoning around Figure 12.9, we know there exists $f : [\nu(r); \nu(s)] \mapsto [\lambda(r); \lambda(s)]$ that is bijective and which preserves integer values. Consider the following

$$\zeta': \begin{cases} x \in [0; \nu(r)] & \mapsto \zeta(x) \\ x \in [\nu(r); \nu(s)] & \mapsto f(x) \\ x \in [\nu(s); +\infty[& \mapsto x + \lambda(s) - \nu(s) \end{cases} . \text{ The function } \zeta' \in \text{Aut}(\mathbb{Q}_+), \text{ as it is bijective and increasing. Moreover, it preserves } \mathbb{N}, \text{ and is such that } \zeta'(\nu) = \lambda.$$

In both cases, we have established the existence of some morphism that sends ν to λ and that preserves $\mathbb{N}$, so the inductive invariant holds at step $i + 1$.

Overall, we have constructed $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ such that $\zeta(\nu) = \lambda$, so (ν, λ) is looping. This concludes the proof. $\square$

An immediate corollary is that the type of a looping pair of valuations is necessarily widening:

Corollary 12.52 *Let $\nu, \lambda : R \rightarrow \mathbb{N}$ be a pair of valuations over some finite set of registers R . If (ν, λ) is looping, then $\text{type}(\nu, \lambda)$ is widening.*

Observe that multiplying by a constant preserves the character of being widening for $\mathbb{Q}_+$ -configurations, thus of being looping for $\mathbb{N}$ -configurations:

Formally, s is such that $\nu(s) = \max \nu(R)$.

$\nu|_{R'}$ denotes the restriction of ν to R' , i.e. $\nu|_{R'} : r \in R' \mapsto \nu(r)$.

This is a corollary of the easy implication, namely that looping implies widening, and can also be deduced from Lemma C.1.

We spelled out the proof in Section C.0.1 for completeness; this is merely an exercise.

Observation 12.53 Let $\nu, \lambda : R \rightarrow \mathbb{Q}_+$. If (ν, λ) is widening, then so is $(K\nu, K\lambda)$, for all $K \in \mathbb{N} \setminus \{0\}$. Moreover, $\text{type}(K\nu, K\lambda) = \text{type}(\nu, \lambda)$.

By combining lemmas 12.51 and 12.49, we get that a widening type always contains a looping pair:

Lemma 12.54 Let $R = \{r_1, \dots, r_k\}$ be some finite set of registers, and let $\sigma(r_1, \dots, r_k, r'_1, \dots, r'_k)$ be a widening type. Then, there exists a pair of valuations $\nu, \lambda : R \rightarrow \mathbb{N}$ that has type σ and (ν, λ) is looping.

Proof. We spell out the proof, to wrap everything together.

Let $R = \{r_1, \dots, r_k\}$, and let $\sigma(r_1, \dots, r_k, r'_1, \dots, r'_k)$ be a widening type over R . By Lemma 12.49, we know that there exists $\nu', \lambda' : R \rightarrow \mathbb{Q}_+$ such that $\sigma(\nu', \lambda')$ and (ν', λ') is widening. Take some factor $K \in \mathbb{N} \setminus \{0\}$ such that $\nu = K(\nu')$ and $\lambda = K(\lambda')$ take their values in $\mathbb{N}$. By Observation 12.53, (ν, λ) is again widening and $\text{type}(\nu, \lambda) = \text{type}(\nu', \lambda') = \sigma$. By Lemma 12.51, we get that (ν, λ) is looping. $\square$

A pair of valuations (ν, λ) has type σ if they jointly satisfy σ , i.e. $\sigma(\nu, \lambda)$ holds.

K can be taken as the product of all denominators of rational numbers that appear in ν, λ ; see Lemma 12.44.

Candidate Loops The above result yields a characterisation of the existence of a loop that can be checked with the tools that we developed for $(\mathbb{Q}_+, <, 0)$, which is oligomorphic. This characterisation is expressed in terms of a candidate loop, which denotes partial runs in $\mathbb{Q}_+$ that have a widening type.

Definition 12.16 (Candidate loop in $\mathbb{N}$) Let A be a register automaton over $(\mathbb{Q}_+, <, 0)$. A *candidate loop in $\mathbb{N}$* is a partial run $C \xrightarrow{u} D$ such that (C, D) is pseudo-widening, i.e. $\text{type}(C, D)$ is widening.

Note that a loop in $\mathbb{N}$ is in particular a candidate loop in $\mathbb{N}$.

The above analysis establishes that a candidate loop is actually a witness of the existence of a loop in $\mathbb{N}$, in the sense that it contains a loop in $\mathbb{N}$ in its orbit.

Proposition 12.55 Let A be a register automaton over $(\mathbb{N}, <, 0)$. If there exists a partial run $\rho : B \xrightarrow{u}_{A_{\mathbb{Q}_+}} C \xrightarrow{v}_{A_{\mathbb{Q}_+}} D$ in $A_{\mathbb{Q}_+}$ such that $C \xrightarrow{v} D$ is a candidate loop in $\mathbb{N}$, then there exists some morphism $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that $\mu(\rho) : \mu(B) \xrightarrow{\mu(u)}_{A_{\mathbb{N}}} \mu(C) \xrightarrow{\mu(v)}_{A_{\mathbb{N}}} \mu(D)$ is a partial run in $A_{\mathbb{N}}$ where $\mu(C) \xrightarrow{\mu(v)}_{A_{\mathbb{N}}} \mu(D)$ is a loop in $\mathbb{N}$.

The notion of candidate loop in $\mathbb{N}$ might seem redundant with that of a widening type, and it indeed is accessory. It is introduced to prepare for the analogous notion of candidate synchronised loop, that concerns pairs of partial runs.

The prefix $B \xrightarrow{u} C$ of the run is mentioned in the statement for the following reason: from a candidate loop in $\mathbb{N}$, one can deduce a loop in $\mathbb{N}$ of the same type, however the morphism that maps one to the other does not necessarily preserves $\mathbb{N}$. Here, we show that up to multiplying by some factor, we get that the prefix also belongs to $\mathbb{N}$.

Implicitly, B, C, D take their values in $\mathbb{Q}_+$.

Proof. Let A be a register automaton over $(\mathbb{N}, <, 0)$, and assume that there exists a partial run $\rho : B \xrightarrow{u}_{A_{\mathbb{Q}_+}} C \xrightarrow{v}_{A_{\mathbb{Q}_+}} D$ in $A_{\mathbb{Q}_+}$ such that $C \xrightarrow{v} D$ is a candidate loop in $\mathbb{N}$, i.e. $\text{type}(C, D)$ is widening. This implies that C and D have the same state, as a widening type preserves types by definition. Write $C = (p, \nu)$ and $D = (p, \lambda)$. By Lemma 12.54, we know that there exists a looping pair of valuations in $\mathbb{N}$ such that $\text{type}(\nu, \lambda) = \text{type}(\nu', \lambda')$. Let $v \in \text{Aut}(\mathbb{Q}_+)$ be such that $v(\nu, \lambda)$ is a pair of $\mathbb{N}$ -valuations that is looping. We are almost done; the only remaining obstacle is that v does not necessarily preserves $\mathbb{N}$, so it might be that $v(B)$ or $v(u)$ or $v(v)$ lie in $\mathbb{Q}_+$. However, by multiplying by a large enough factor K , we get that they all belong to $\mathbb{N}$, i.e. $Kv(B) \xrightarrow{Kv(u)}_{A_{\mathbb{N}}} Kv(C) \xrightarrow{Kv(v)}_{A_{\mathbb{N}}} Kv(D)$ is a

Recall that having the same type is equivalent to belonging to the same orbit, see Proposition 12.17.

partial run of $A_{\mathbb{N}}$ (see Lemma 12.44). Multiplying by a constant preserves the fact of being looping, since it preserves the fact of being widening (see Observation 12.53). We exhibited the sought loop, by taking the morphism $\mu = K \circ v$, which is a morphism of $(\mathbb{Q}_+, <, 0)$. $\square$

Recall that morphisms have a group structure (see Section 4.4.6).

Runs and Loops

We now show that one can employ pumping arguments in register automata over $(\mathbb{N}, <, 0)$: if there is an infinite run in some automaton, then there necessarily is a loop, in the sense of Definition 12.15. This is obtained by a Ramsey argument.

Proposition 12.56 *Let A be a register automaton over data domain $(\mathbb{N}, <, 0)$. If there is a run $\rho : B \xrightarrow{x}$ for some configuration $B \in \text{Configs}(A)$ and some $x \in \mathbb{N}^\omega$, then there exists configurations B', C, D and finite data words $u, v \in \mathbb{N}^*$ with $v \neq \varepsilon$ such that $B' \xrightarrow{u} C \xrightarrow{v} D$, $\text{type}(B') = \text{type}(B)$ and $C \xrightarrow{v} D$ is a loop in $\mathbb{N}$.*

Moreover, if ρ is final, we can take C and D final.

Proof. Let $A = (Q, I, F, \Delta)$ be a register automaton over data domain $(\mathbb{N}, <, 0)$, and assume that there is a run $\rho : B \xrightarrow{x}$ for some configuration $B \in \text{Configs}(A)$ and some $x \in \mathbb{N}^\omega$. To avoid repetition, we treat both parts of the statement together, highlighting when we use the hypothesis that ρ is final. Write $\rho = C_0 \xrightarrow{d_0} C_1 \xrightarrow{d_1} C_2 \dots$, where $C_0 = B$. For each $i \geq 0$, let $\tau_i = \text{type}(C_i)$.

Restrict to configurations that have the same type Since there are only finitely many types (in $(\mathbb{Q}_+, <, 0)$), we get that there exists some type τ that occurs infinitely often along the run. If ρ is final, we can further assume that such a type τ is final. Let $(C_j)_{j \in \mathbb{N}}$ be an infinite subsequence of $(C_i)_{i \in \mathbb{N}}$ such that for all $j \in \mathbb{N}$, $\text{type}(C_j) = \tau$. We colour the set of unordered pairs as follows: given $j < k$, we let $c(\{C_j, C_k\}) = \text{type}(C_j, C_k)$.

Some type is said *final* if its state is final. Recall that the type of a configuration (q, v) is the pair $(q, \text{type}(v))$.

What we colour is unordered pairs, although they are de facto ordered by the mention of j and k .

[118]: Ramsey (1930), ‘On a Problem of Formal Logic’

Among them, infinitely many have the same pairwise type (Ramsey) By the infinite version of Ramsey’s theorem [118, Theorem A], there is an infinite subset such that all pairs have the same colour σ , where σ is a type of pairs of configurations. Let $(C_k)_{k \in \mathbb{N}}$ be an infinite subsequence such that for all $j < k$, $\text{type}(C_j, C_k) = \sigma$. In the following, we assume that σ is given as a quantifier-free formula $\sigma(r_1, \dots, r_k, r'_1, \dots, r'_k)$.

Recall that $(\mathbb{Q}_+, <, 0)$ is oligomorphic, so its types are characterised by first-order formulas (see Propositions 12.17 and 12.21). From Example 12.4, we can see that in $(\mathbb{Q}, <, 0)$ (and $\mathbb{Q}_+$), quantifier-free formulas suffice.

Such a pairwise type is necessarily widening Striving for a contradiction, suppose that σ is not widening. There are two cases:

- There exists some $r \in R$ such that $\sigma \Rightarrow r > r'$. It means that for all $j < k$, $C_j(r) > C_k(r)$. In particular, this means that $C_0(r) > C_1(r) > \dots > C_n(r) \dots$, which yields an infinite descending chain in $\mathbb{N}$, and leads to a contradiction.

- There exists some s such that $\sigma \Rightarrow s = s'$ and some r which satisfies $\sigma \Rightarrow r < s$ and $\sigma \not\Rightarrow r = r'$. If $\sigma \Rightarrow r > r'$, we are back to the first case. Otherwise, it means $\sigma \Rightarrow r < r'$. Then, on the one hand, we have $C_0(r) < C_1(r) < \dots < C_n(r) < \dots$. On the other hand, $C_0(s) = C_1(s) = \dots = C_n(s) = \dots$. But we also have that for all $k \in \mathbb{N}$, $C_k(r) < C_k(s) = C_0(s)$. Overall, we get an infinite increasing chain which is bounded from above by $C_0(s)$, which again leads to a contradiction.

This yields a candidate loop Thus, σ is widening. Pick some pair of configurations $C = C_k$ and $D = C_l$ for some $k < l$ from the last extracted subsequence that satisfy $\text{type}(C, D) = \sigma$. Then, $B \xrightarrow{u} C \xrightarrow{v} D$ for some $u \in \mathbb{N}^*$ and $v \in \mathbb{N}^+$, and $C \xrightarrow{v} D$ is a candidate loop in $\mathbb{N}$.

It would actually suffice that u and v take their values in $\mathbb{Q}_+$.

Which yields a loop By Proposition 12.55, we get that there exists a morphism $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that $\mu(B) \xrightarrow[A_{\mathbb{N}}]{\mu(u)} \mu(C) \xrightarrow[A_{\mathbb{N}}]{\mu(v)} \mu(D)$ is a partial run in $A_{\mathbb{N}}$ and $\mu(C) \xrightarrow{\mu(v)} \mu(D)$ is a loop. Let $B' = \mu(B)$; by Proposition 12.17, we have that $\text{type}(B) = \text{type}(B')$. Moreover, if ρ is final, we constructed C and D so that they are final, so this is also the case of $\mu(C)$ and $\mu(D)$. This yields the sought loop. $\square$

Emptiness Problem

Our characterisation of the non-emptiness of register automata over $(\mathbb{N}, <, 0)$ is now ripe, so let us harvest it.

Proposition 12.57 *Let $A = (Q, I, F, \Delta)$ be a register automaton over $(\mathbb{N}, <, 0)$, and let A' be the same register automaton over $(\mathbb{Q}, <, 0)$.*

$L_{\mathbb{N}}(A) = \emptyset$ if and only if $A_{\mathbb{Q}_+}$ contains a candidate loop over final configurations that is reachable, i.e., $C' \xrightarrow[A']{u} C \xrightarrow[A']{v} D$ for some final configurations $C, D \in \text{Configs}(A')$ such that $\text{type}(C, D)$ is widening, and some finite data words $u, v \in (\mathbb{Q}_+)^*$ with $v \neq \varepsilon$.

Proof. We simply have to apply our study of loops in $\mathbb{N}$. Let $A = (Q, I, F, \Delta)$ be a register automaton over $(\mathbb{N}, <, 0)$.

First, assume that $L_{\mathbb{N}}(A) = \emptyset$. It means that there exists an accepting run $\rho : C' \xrightarrow{x}$ for some $x \in \mathbb{N}^\omega$. By Proposition 12.56, we know that there exists configurations B', C, D and finite data words $u, v \in \mathbb{N}^*$ with $v \neq \varepsilon$ such that $\text{type}(B') = \text{type}(C')$, $B' \xrightarrow{u} C$ and $C \xrightarrow{v} D$ is a loop in $\mathbb{N}$. Moreover, we have that C and D are final. By Lemma 12.51, we get that (C, D) is widening, so in particular $\text{type}(C, D)$ is widening (Corollary 12.52). We obtain the result by pointing out that $\text{type}(B') = \text{type}(C')$ implies that $B' = C'$, since $\text{type}(C')$ requires that all registers are valued 0 (Observation 12.48).

We now move to the other implication: assume that $C' \xrightarrow[A_{\mathbb{Q}_+}]{u} C \xrightarrow[A_{\mathbb{Q}_+}]{v} D$ for some final configurations $C, D \in \text{Configs}(A_{\mathbb{Q}_+})$ such that $\text{type}(C, D)$

is widening, and some finite data words $u, v \in (\mathbb{Q}_+)^*$ with $v \neq \varepsilon$. By Proposition 12.55, we know that there exists some morphism $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that $\mu(C') \xrightarrow[A_{\mathbb{N}}]{\mu(u)} \mu(C) \xrightarrow[A_{\mathbb{N}}]{\mu(v)} \mu(D)$, where $\mu(C) \xrightarrow[A_{\mathbb{N}}]{\mu(v)} \mu(D)$ is a loop in $\mathbb{N}$. By Observation 12.43, we have that $\mu(C') = C'$. By Observation 12.50, we get that there exists $x \in \mathbb{N}^\omega$ such that $\mu(C) \xrightarrow[A_{\mathbb{N}}]{x}$ is a final run in $A_{\mathbb{N}}$. Thus, $\mu(u) \cdot x \in L_{\mathbb{N}}(A)$. $\square$

The above characterisation yields a PSPACE decision procedure for the emptiness problem of register automata over data ω -words. Note that this result is already known from [39]. Their proof relies on similar ideas, but the above analysis works out the details and yields additional properties that are needed for the characterisation and decision of functionality and continuity.

Theorem 12.58 ([39, Theorem 16],[142, Corollary 4.18]) *The emptiness problem for register automata over $(\mathbb{N}, <, 0)$ is PSPACE-complete.*

Proof. Let A be a register automaton over $(\mathbb{N}, <, 0)$. By the above Proposition 12.57, we know that $L(A) \neq \emptyset$ if and only if there exists two final configurations D, E over $\mathbb{Q}_+$ and two words $u, v \in (\mathbb{Q}_+)^*$ with $v \neq \varepsilon$ such that $C' \xrightarrow{u} C \xrightarrow{v} D$ and $\text{type}(C, D)$ is widening.

Let us show that this property can be checked by a register automaton over $(\mathbb{Q}_+, <, 0)$. The automaton simulates A over the superstructure $(\mathbb{Q}_+, <, 0)$, while keeping in memory the type of the current configuration. At some point, along the run, it guesses the current configuration is the start C of the candidate loop, and memorises its type. From this point on, it maintains in memory the pair type $\text{type}(C, E)$, where E denotes the current configuration. If it encounters a configuration D such that $\text{type}(C, D)$ is widening (in particular, this implies that $\text{type}(C) = \text{type}(D)$), it enters an accepting sink state. Checking that a pair type is widening can be done in polynomial space from the quantifier-free formula that represents the pair type, which is itself of size polynomial in $|A|$. Overall, such an automaton can be described in polynomial space, and $L_{\mathbb{Q}_+}(A') \neq \emptyset$ if and only if $L_{\mathbb{N}}(A) \neq \emptyset$. Since the emptiness problem for register automata over $(\mathbb{Q}_+, <, 0)$ is in PSPACE (Theorem 12.27), this yields a PSPACE decision procedure to check the emptiness of A .

As usual, PSPACE-hardness results from PSPACE-hardness of the emptiness problem over $(\mathbb{N}, =)$ [28, Theorem 5.1]. $\square$

12.3.2. Characterisation and Decision of Functionality

The notion of loop in $\mathbb{N}$ proves sufficient for providing characterisations of functionality and continuity that are analogous to the oligomorphic case. The patterns are essentially the same; one simply strengthens the notion of synchronised loop by asking that both paths can jointly be iterated in $(\mathbb{N}, <, 0)$. This amounts to requiring that the type of the pair of valuations at the beginning and at the end of the synchronised loop is widening.

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

In the case of $(\mathbb{N}, <, 0)$, the characterisation and decision of functionality and continuity are less similar than in the oligomorphic case. We thus prefer to group the study differently.

Synchronised Loops

The characterisations that we exhibited for the case of $(\mathbb{D}, =, \mathfrak{C})$ (Section 12.1.3) and of oligomorphic data domains (Section 12.2.3) heavily rely on a notion of *synchronised loop*. Such a notion can also be defined in $(\mathbb{N}, <, 0)$.

Definition 12.17 Let T be a non-deterministic register transducer over domain $(\mathbb{N}, <, 0)$. A *synchronised loop in $\mathbb{N}$ of T* is a pair of partial runs

$$\begin{aligned} C_1 &\xrightarrow{u|u_1} D_1 \\ C_2 &\xrightarrow{u|u_2} D_2 \end{aligned}$$

such that there exists an $\mathbb{N}$ -preserving morphism $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that maps (C_1, C_2) to (D_1, D_2) .

First, observe that a synchronised loop can indeed be iterated in the same way as loops in $\mathbb{N}$ (the proof is the same as for Observation 12.50):

Observation 12.59 Let T be a register transducer, and let $C_1 \xrightarrow[T]{\rho_1} D_1$ and $C_2 \xrightarrow[T]{\rho_2} D_2$ be a pair of partial runs that form a synchronised loop in $\mathbb{N}$, that we call L . Then, there exists $x \in \mathbb{N}^\omega$ and $y, z \in \mathbb{N}^\infty$ such that $C_1 \xrightarrow[T]{\rho_1^\omega} y$ and $C_2 \xrightarrow[T]{\rho_2^\omega} z$. If L is 1-final (respectively, 2-final, final), then the run over x yielding output y (respectively z , both) is final.

Candidate synchronised loops We have seen that the existence of loops in $\mathbb{N}$ is characterised by a witness in $(\mathbb{Q}_+, <, 0)$, that we called a candidate loop. Such a notion can be lifted to synchronised loops as follows.

Definition 12.18 (Candidate synchronised loop) Let T be a non-deterministic register transducer. A pair of partial runs

$$\begin{aligned} C_1 &\xrightarrow{u|u_1} D_1 \\ C_2 &\xrightarrow{u|u_2} D_2 \end{aligned}$$

is a *candidate synchronised loop* if the pair of pairs of configurations $((C_1, C_2), (D_1, D_2))$ is pseudo-widening. Recall that this is the case whenever $\text{type}(C_1, C_2, D_1, D_2)$ is widening.

Note that here, we consider the *pairwise* type of *pairs* of configurations. Thus, $\text{type}(C_1, C_2, D_1, D_2)$ can be written as some quantifier-free formula $\sigma(r_1, \dots, r_k, s_1, \dots, s_k, r'_1, \dots, r'_k, s'_1, \dots, s'_k)$, where $r_1, \dots, r_k$ denotes the registers of $C_1, s_1, \dots, s_k$, those of C_2 , and similarly their primed copy respectively concern D_1 and D_2 . Let us spell out what it means that $\text{type}(C_1, C_2, D_1, D_2)$ is widening:

- For all $r \in R$, $D_1(r) \geq C_1(r)$ and $D_2(r) \geq C_2(r)$
- For all $s \in R$, if $D_1(s) = C_1(s)$ then for all $r \in R$ such that $C_1(r) \leq C_1(s)$, we have $D_1(r) = C_1(r)$. This is also the case over C_2 and D_2 : for all $r \in R$ such that $C_2(r) \leq C_1(s)$, we have $D_2(r) = C_2(r)$.

- The same holds over C_2 and D_2 : for all $s \in R$, if $D_2(s) = C_2(s)$ then for all $r \in R$ such that $C_2(r) \leq C_2(s)$, we have $D_2(r) = C_2(r)$. And again this also holds for C_1 and D_1 : for all $r \in R$ such that $C_1(r) \leq C_2(s)$, we have $D_1(r) = C_1(r)$.

This cross dependency is crucial to guarantee that we indeed have a synchronised loop, so that it can be jointly iterated over the *same* input in $\mathbb{N}$.

A synchronised loop is *1-final* if C_1 (hence D_1 , since they have the same type) is final, and similarly for *2-final*. It is *final* if it is 1-final, and moreover a final configuration occurs in the second run. Formally, we ask that it is of the form

$$\begin{array}{ccc} C_1 & \xrightarrow{v|v_1} E_1 & \xrightarrow{w|w_1} D_1 \\ C_2 & \xrightarrow{x|x_2} F_2 & \xrightarrow{y|y_2} D_2 \end{array}$$

where C_1, F_2 and D_1 are final. Note that we do not require final configurations to appear at the same time.

An Automaton for Pairs of Runs Showing that a candidate synchronised loop is indeed a witness of the existence of an actual synchronised loop can be done by generalising the study that we conducted for register automata. To avoid redoing all the work, we instead apply the previous results to a register automaton that recognises pairs of runs. This idea was already used in Section 12.1, although in a different perspective: the goal was then to leverage the renaming property of register automata over $(\mathbb{D}, =)$.

We recall the formal definition of the data language $L_{\otimes}(T)$ of interleaved traces of runs of a transducer T (Equation 10.1):

$$L_{\otimes}(T) = \{ \rho_1 \otimes \rho_2 \mid \rho_1 \text{ and } \rho_2 \text{ are accepting runs of } T \}$$

A product of T with itself, along with a flattening of runs of T into their traces, establishes that $L_{\otimes}(T)$ is recognised by a register automaton with twice as many registers as T :

Proposition 10.2 *Let T be register transducer with k registers. The data language $L_{\otimes}(T)$ is recognised by a non-deterministic register automaton $A_{\otimes}^T$ with $2k$ registers, whose size is polynomial in $|T|$.*

Besides, for all configurations $(p, v), (p', v'), (q, \lambda), (q', \lambda') \in \text{Configs}(T)$, all data values $d_0, \dots, d_n, e_0, \dots, e_n$ and all finite data words $y_0, \dots, y_n$ and $z_0, \dots, z_n$, there is a partial run

$$((p, p'), (v, v')) \xrightarrow{d_0 \cdot y_0 \$ e_0 \cdot z_0 \$ \dots \$ d_n \cdot y_n \$ e_n \cdot z_n} ((q, q'), (\lambda, \lambda'))$$

in $A_{\otimes}^T$ if and only if there are two partial runs

$$(p, v) \xrightarrow[T]{d_0 \dots d_n | y_0 \dots y_n} (q, \lambda) \quad (p', v') \xrightarrow[T]{e_0 \dots e_n | z_0 \dots z_n} (q', \lambda')$$

Requiring that final configurations appear synchronously would be too strong: already in the finite alphabet case, one cannot in general exhibit a synchronised loop where both states are final at the same time.

This choice of presentation has the unfortunate side effect to make the arguments look more technical, as they lie on a syntactical level. However, it greatly increases the conciseness of the argument and avoids repetitions. It is an interesting exercise to adapt the proof techniques of the previous section to synchronised loops.

in T such that (p, v) and (p', v') are jointly reachable in T , i.e. there exists a finite input data word $u \in \mathbb{D}^*$ such that $C^i \xrightarrow[T]{u|u_1} (p, v)$ and $C^i \xrightarrow[T]{u|u_2} (p', v')$.

With the same notations, there is a run

$$((p, p'), (v, v')) \xrightarrow{d_0 \cdot y_0 \$ e_0 \cdot z_0 \$ d_1 \cdot y_1 \$ e_1 \cdot z_1 \dots} ((q, q'), (\lambda, \lambda'))$$

in $A_{\otimes}^T$ if and only if there are two runs

$$(p, v) \xrightarrow[T]{d_0 d_1 \dots | y_0 \cdot y_1 \dots} (p', v') \xrightarrow[T]{e_0 e_1 \dots | z_0 \cdot z_1 \dots}$$

By making all the states of $A_{\otimes}^T$ final, one can also recognise interleaved traces of runs of T that are not necessarily accepting.

We now establish a series of lemmas that make the connection between candidate synchronised loops and synchronised loops in $\mathbb{N}$. In the same way as for register automata (see Proposition 12.55), from a candidate synchronised loop in $\mathbb{N}$, one can build a synchronised loop in $\mathbb{N}$.

Lemma 12.60 *Let T be a non-deterministic register transducer over $(\mathbb{N}, <, 0)$. If some candidate loop in $\mathbb{N}$ is reachable from some pair of configurations (B_1, B_2) , i.e.*

$$\begin{array}{l} B_1 \xrightarrow[T_{Q_+}]{u|u_1} C_1 \xrightarrow[T_{Q_+}]{v|v_1} D_1 \\ B_2 \xrightarrow[T_{Q_+}]{u|u_2} C_2 \xrightarrow[T_{Q_+}]{v|v_2} D_2 \end{array}$$

where $\text{type}(C_1, C_2, D_1, D_2)$ is widening, then there exists a morphism $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that

$$\begin{array}{c} \mu(B_1) \xrightarrow[T_{\mathbb{N}}]{\mu(u)|\mu(u_1)} \underbrace{\mu(C_1) \xrightarrow[T_{\mathbb{N}}]{\mu(v)|\mu(v_1)} \mu(D_1)}_{\mathbb{L}} \\ \mu(B_2) \xrightarrow[T_{\mathbb{N}}]{\mu(u)|\mu(u_2)} \underbrace{\mu(C_2) \xrightarrow[T_{\mathbb{N}}]{\mu(v)|\mu(v_2)} \mu(D_2)}_{\mathbb{L}} \end{array}$$

are partial runs in $\mathbb{N}$ and $\mu(C_1) \xrightarrow[T_{\mathbb{N}}]{\mu(v)|\mu(v_1)} \mu(D_1)$ and $\mu(C_2) \xrightarrow[T_{\mathbb{N}}]{\mu(v)|\mu(v_2)} \mu(D_2)$ form a synchronised loop $\mathbb{L}$ in $\mathbb{N}$.

Proof. Let T be a non-deterministic register transducer over $(\mathbb{N}, <, 0)$. The idea is to leverage Proposition 12.55 by applying it to $A_{\otimes}^T$. Assume that there exists a candidate synchronised loop reachable from configurations (B_1, B_2) :

$$\begin{array}{l} B_1 \xrightarrow[T_{Q_+}]{u|u_1} C_1 \xrightarrow[T_{Q_+}]{v|v_1} D_1 \\ B_2 \xrightarrow[T_{Q_+}]{u|u_2} C_2 \xrightarrow[T_{Q_+}]{v|v_2} D_2 \end{array}$$

Once again, we mention the prefixes of the synchronised loop to avoid having to put them back in $\mathbb{N}$ manually every time we use the lemma.

where $\text{type}(C_1, C_2, D_1, D_2)$ is widening. By Proposition 10.2, this means that there exists a partial run $(B_1, B_2) \xrightarrow[A_{\otimes}^{T_{\mathbb{Q}_+}}]{w} (C_1, C_2) \xrightarrow[A_{\otimes}^{T_{\mathbb{Q}_+}}]{x} (D_1, D_2)$

where w is an encoding of $(u, u_1), (u, u_2)$ and x encodes $(v, v_1), (v, v_2)$. This forms a reachable candidate loop in $A_{\otimes}^{T_{\mathbb{Q}_+}}$, since $\text{type}(C_1, C_2, D_1, D_2)$ is widening. Using Proposition 12.55, we get that there exists a partial run $(\mu(B_1), \mu(B_2)) \xrightarrow{\mu(w)} (\mu(C_1), \mu(C_2)) \xrightarrow{\mu(x)} (\mu(D_1), \mu(D_2))$

in $A_{\otimes}^{T_{\mathbb{Q}_+}}$ that takes its values in $\mathbb{N}$, and such that $(\mu(C_1), \mu(C_2)) \xrightarrow{\mu(x)} (\mu(D_1), \mu(D_2))$ forms a loop in $\mathbb{N}$, that we call L . The way interleaved traces are encoded implies that $\mu(w)$ encodes the pairs $(\mu(u), \mu(u_1))$ and $(\mu(u), \mu(u_2))$; similarly $\mu(x)$ encodes $(v, v_1), (v, v_2)$, and they all belong to $\mathbb{N}$. Using the converse direction of Proposition 10.2, we get that $B'_1 \xrightarrow[T_{\mathbb{N}}]{u'|u'_1} C'_1 \xrightarrow[T_{\mathbb{N}}]{v'|v'_1} D'_1$ and $B'_2 \xrightarrow[T_{\mathbb{N}}]{u'|u'_2} C'_2 \xrightarrow[T_{\mathbb{N}}]{v'|v'_2} D'_2$ are both partial runs of $T_{\mathbb{N}}$, where $B'_1 = \mu(B_1)$, $u' = \mu(u)$, and so on. Since L is a loop in $\mathbb{N}$ in $A_{\otimes}^{T_{\mathbb{Q}_+}}$, we know that there exists an $\mathbb{N}$ -preserving morphism $\zeta \in \text{Aut}(\mathbb{Q}_+)$ such that $\zeta(C'_1, C'_2) = (D'_1, D'_2)$. In other words,

$$\begin{aligned} \mu(C_1) &\xrightarrow[T]{\mu(u)|\mu(u_1)} \mu(D_1) \\ \mu(C_2) &\xrightarrow[T]{\mu(u)|\mu(u_2)} \mu(D_2) \end{aligned}$$

forms a synchronised loop in $\mathbb{N}$ for T , which is reachable from $(\mu(B_1), \mu(B_2))$ through a partial run in $\mathbb{N}$. This concludes the proof. $\square$

Conversely, from a pair of runs over the same input, one can extract a candidate synchronised loop (and hence a synchronised loop, by the above result).

Lemma 12.61 *Let T be a non-deterministic register transducer over $(\mathbb{N}, <, 0)$. If there exists two runs ρ_1 and ρ_2 in T over the same input $x \in \mathbb{N}^\omega$:*

$$\begin{aligned} B_1 &\xrightarrow[T]{x|y} \rho_1 \\ B_2 &\xrightarrow[T]{x|z} \rho_2 \end{aligned}$$

then T admits a candidate synchronised loop L over $\mathbb{Q}_+$ that is reachable from B_1 and B_2 , i.e. there exists configurations C_1, C_2, D_1, D_2 in $\mathbb{Q}_+$ such that

$$\begin{aligned} B_1 &\xrightarrow[T_{\mathbb{Q}_+}]{u|u_1} C_1 \xrightarrow[T_{\mathbb{Q}_+}]{v|v_1} D_1 \\ B_2 &\xrightarrow[T_{\mathbb{Q}_+}]{u|u_2} C_2 \xrightarrow[T_{\mathbb{Q}_+}]{v|v_2} D_2 \end{aligned}$$

where $\text{type}(C_1, C_2, D_1, D_2)$ is widening.

Moreover, if ρ_1 (respectively ρ_2 , both) is final, then L is 1-final (resp. 2-final, final).

Proof. The main idea is to apply Proposition 12.57 to a register automaton that recognises interleaved traces of runs of T that have the same input, which we construct from $A_{\otimes}^T$.

We use the prime symbol to denote the application of μ for clarity, to avoid a morphism soup.

Exhibit a synchronised loop Let T be a non-deterministic register transducer over $(\mathbb{N}, <, 0)$. Let ρ_1 and ρ_2 be two runs of T over the same input $x \in \mathbb{N}^\omega$. We write, with the same notations as above:

$$\begin{array}{c} B_1 \xrightarrow[T]{x|y} \rho_1 \\ B_2 \xrightarrow[T]{x|z} \rho_2 \end{array}$$

Let L be the data language of interleaved traces of runs (not necessarily initial) of T over the same input, and that additionally start in respective configurations B'_1 and B'_2 such that $\text{type}(B'_1, B'_2) = \text{type}(B_1, B_2)$. Formally,

$$L_{\otimes}^{\tau}(T) = \left\{ \rho_1 \otimes \rho_2 \left| \begin{array}{l} \rho_1 : B'_1 \xrightarrow[T]{x'|y'} \text{ and} \\ \rho_2 : B'_2 \xrightarrow[T]{x'|z'} \text{ are both runs of } T \text{ such that} \\ \text{in}(\rho_1) = \text{in}(\rho_2) \text{ and } \text{type}(B'_1, B'_2) = \text{type}(B_1, B_2) \end{array} \right. \right\}$$

One can construct a register automaton A over data domain $(\mathbb{N}, <, 0)$ that recognises $L_{\otimes}^{\tau}(T)$: we already know by Proposition 10.2 that interleaved traces of runs of a transducer can be recognised by $A_{\otimes}^T$. The fact that they are not initial does not obstruct the construction: the register automaton simply guesses B'_1 and B'_2 at the beginning using non-deterministic reassignment. Checking that $\text{type}(B'_1, B'_2) = \text{type}(B_1, B_2)$ is immediate, as the register automaton can check the type of a configuration on its transitions (recall that in $(\mathbb{Q}_+, <, 0)$, types are quantifier-free formulas, so they can be encoded in tests). Checking that $\text{in}(\rho_1) = \text{in}(\rho_2)$ is done using an additional register and an M -bounded counter. Note that here, all states are accepting, since A recognises all runs of T , whether they are final or not. We know that $L_{\otimes}^{\tau}(T) \neq \emptyset$, since $\rho_1 \otimes \rho_2 \in L$. By Proposition 12.57, we get that there exists a candidate loop in A , i.e. a configuration in $\mathbb{Q}_+$ that repeats up to automorphism and whose type is widening.

This can also be implemented without using non-deterministic reassignment: before processing its input, the automaton asks to read two valuations, and checks that they have the correct type. It guesses the corresponding states using the usual non-determinism. We use non-deterministic reassignment to avoid the case where the loop occurs in this initial phase.

As a consequence, we can write $(C^l, C^l) \xrightarrow[A]{s} (C_1, C_2) \xrightarrow[A]{t} (D_1, D_2)$ for some configurations C_1, C_2, D_1, D_2 of T such that $\text{type}(C_1, C_2) = \text{type}(D_1, D_2)$ and $\text{type}(C_1, C_2, D_1, D_2)$ is widening. Recall that configurations of $A_{\otimes}^T$ consist in pairs of configurations of T . We omit in our reasoning the additional register used to control that the inputs of the runs ρ_1 and ρ_2 coincide. This is not harmful since if a type is widening, its subtypes are widening as well. By definition of $L_{\otimes}^{\tau}(T)$, we know that s encodes two pairs (u, u_1) and (u, u_2) for some $u, u_1, u_2 \in \mathbb{N}^*$; similarly, t encodes (v, v_1) and (v, v_2) for $v, v_1, v_2 \in \mathbb{N}^*$. Overall, we get:

$$\begin{array}{c} B'_1 \xrightarrow[T]{u|u_1} C_1 \xrightarrow[T]{v|v_1} D_1 \\ B'_2 \xrightarrow[T]{u|u_2} C_2 \xrightarrow[T]{v|v_2} D_2 \end{array}$$

where $\text{type}(C_1, C_2, D_1, D_2)$ is widening and $\text{type}(B'_1, B'_2) = \text{type}(B_1, B_2)$. Since (B'_1, B'_2) is in the same orbit as (B_1, B_2) , there exists a morphism $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that $\mu(B'_1, B'_2) = B_1, B_2$. By applying such a morphism to the above partial runs, we exhibit a candidate synchronised loop L in T that is reachable from (B_1, B_2) .

Taking care of final states If ρ_1 is final, one restricts the accepting states of A to the pairs of states that are accepting in ρ_1 . We again have that the modified $L_{\otimes}^{\tau}(T) \neq \emptyset$, since ρ_1 is final. This, way, we know that in the above synchronised loop that we exhibited, in $B_1 \xrightarrow[T]{u|u_1} C_1 \xrightarrow[T]{v|v_1} D_1$ we have that C_1 is accepting (hence D_1 as well, since they have the same type). Thus, L is 1-final. By interverting ρ_1 and ρ_2 , we get that L is 2-final if ρ_2 is final. If both are final, by adding one bit of information to the states, A can require to alternately see an accepting state in ρ_1 and ρ_2 . More precisely, when it is in F_2 for some final configuration in ρ_2 , it stores in memory the information that it read a final configuration of ρ_2 ; then, when it reaches some final configuration F_1 in ρ_1 , it visits an accepting state, and asks again to see a final configuration in ρ_2 . In particular, this means that the only accepting states of A are those that contain an accepting state of ρ_1 . This way, we get that in $B_1 \xrightarrow[T]{u|u_1} C_1 \xrightarrow[T]{v|v_1} D_1$, C_1 and D_1 are final by the same reasoning as above. Moreover, we also have that a final state occurs in ρ_2 in between. In other words, L can be written as

$$\begin{array}{c} C_1 \xrightarrow[T]{a|a_1} E_1 \xrightarrow[T]{b|b_1} D_1 \\ C_2 \xrightarrow[T]{a|a_2} F_2 \xrightarrow[T]{b|b_2} D_2 \end{array}$$

where C_1, F_2, D_1 are final, which means that L is final. This concludes the proof. $\square$

By combining them together, we get an analogous of the pumping argument used in the finite alphabet case: there exists two (infinite) runs from a pair of configurations if and only if there exists a synchronised loop that is reachable from a (possibly distinct) pair of configurations that belongs to the same orbit. Note that we cannot in general assume that the synchronised loop is reachable from the *same* pair of configurations, hence the necessity to mention the prefix of the run in Lemma 12.60, to ensure that we can assume it also takes its values in $\mathbb{N}$.

Characterising (Non-)Functionality This theoretical arsenal provides a characterisation of functionality that is reminiscent of the case of oligomorphic data domains.

Proposition 12.62 *Let T be some non-deterministic register transducer over $(\mathbb{N}, <, 0)$ that recognises some relation $R \subseteq \mathbb{N}^{\omega} \times \mathbb{N}^{\omega}$. T is not functional if and only if there exists two partial runs of $T_{\mathbb{Q}_+}$ in $\mathbb{Q}_+$*

$$\begin{array}{c} C^i \xrightarrow[T_{\mathbb{Q}_+}]{u|u_1} C_1 \xrightarrow[T_{\mathbb{Q}_+}]{v|v_1} D_1 \\ \underbrace{\hspace{10em}}_{L} \\ C^i \xrightarrow[T_{\mathbb{Q}_+}]{u|u_2} C_2 \xrightarrow[T_{\mathbb{Q}_+}]{v|v_2} D_2 \end{array}$$

such that $u_1 \not\parallel u_2$ and L is a candidate loop in $\mathbb{N}$ that is final. To be clear: in the statement, $u, u_1, u_2, v, v_1, v_2, w, w_1, w_2 \in (\mathbb{Q}_+)^*$.

Proof. The result essentially follows from Lemmas 12.60 and 12.61. Let T be some non-deterministic register transducer over $(\mathbb{N}, <, 0)$ that recognises some relation $R \subseteq \mathbb{N}^\omega \times \mathbb{N}^\omega$.

First, assume that T has the above pattern, with the same notations as in the statement. By Lemma 12.60, we know that there exists a morphism $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that

$$\begin{array}{c} \mu(C') \xrightarrow[T_{\mathbb{N}}]{\mu(u)|\mu(u_1)} \mu(C_1) \xrightarrow[T_{\mathbb{N}}]{\mu(v)|\mu(v_1)} \mu(D_1) \\ \underbrace{\hspace{10em}}_{\mu(L)} \\ \mu(C') \xrightarrow[T_{\mathbb{N}}]{\mu(u)|\mu(u_2)} \mu(C_2) \xrightarrow[T_{\mathbb{N}}]{\mu(v)|\mu(v_2)} \mu(D_2) \end{array}$$

We call the resulting synchronised loop $\mu(L)$ since it is obtained from L by applying μ to each partial run.

is a partial run of T in $\mathbb{N}$, where $\mu(L)$ forms a synchronised loop in $\mathbb{N}$. By Observation 12.59, we get that there exists $x \in \mathbb{N}^\omega$, $y, z \in \mathbb{N}^\infty$ such that $\mu(C_1) \xrightarrow[T_{\mathbb{N}}]{x|y}$ and $\mu(C_2) \xrightarrow[T_{\mathbb{N}}]{x|z}$ are runs of T in $\mathbb{N}$. They are moreover final, since L is final. Necessarily, $\mu(C') = C'$ (Observation 12.48), so we get that $C' \xrightarrow[T_{\mathbb{N}}]{\mu(u)|\mu(u_1)} \mu(C_1) \xrightarrow[T_{\mathbb{N}}]{x|y}$ and $C' \xrightarrow[T_{\mathbb{N}}]{\mu(u)|\mu(u_2)} \mu(C_2) \xrightarrow[T_{\mathbb{N}}]{x|z}$ are two accepting runs of T in $\mathbb{N}$. This implies that y, z are infinite words, by Assumption 10.3. Since $u_1 \not\parallel u_2$, we get that $\mu(u_1) \not\parallel \mu(u_2)$, as morphisms preserve mismatches, so $u \cdot x$ has two distinct images, $u_1 \cdot y$ and $u_2 \cdot z$, under $\llbracket T \rrbracket$. In other words, T is not functional.

Conversely, assume that T is not functional, and let $x, y, z \in \mathbb{N}^\omega$ such that $(x, y), (x, z) \in \llbracket T \rrbracket$, with $y \neq z$. Correspondingly, let ρ_1 and ρ_2 be accepting runs of T over input x that respectively yield output y and z . Since $y \neq z$, we have $y \not\parallel z$ as they are both infinite data words (Assumption 10.3), so we have $C' \xrightarrow{s|s_1} B_1 \xrightarrow{t|t_1}$ and $C' \xrightarrow{s|s_2} B_2 \xrightarrow{t|t_2}$ for some $\mathbb{N}$ -configurations B_1, B_2 and $s, s_1, s_2 \in \mathbb{N}^*$, $t, t_1, t_2 \in \mathbb{N}^\omega$, where $s_1 \not\parallel s_2$. Since $B_1 \xrightarrow{t|t_1}$ and $B_2 \xrightarrow{t|t_2}$ are final, by Lemma 12.61, there exists a candidate synchronised loop from B_1 and B_2 that is final, i.e. there exists configurations C_1, C_2, D_1, D_2 and data words $u', u'_1, u'_2, v', v'_1, v'_2 \in \mathbb{N}^*$ with $v' \neq \varepsilon$ such that

$$\begin{array}{c} B_1 \xrightarrow[T_{\mathbb{N}}]{u'|u'_1} C_1 \xrightarrow[T_{\mathbb{N}}]{v'|v'_1} D_1 \\ B_2 \xrightarrow[T_{\mathbb{N}}]{u'|u'_2} C_2 \xrightarrow[T_{\mathbb{N}}]{v'|v'_2} D_2 \end{array} \quad \begin{array}{c} \rho_1 \\ \rho_2 \end{array}$$

where a final configuration occurs in ρ_1 and ρ_2 , and $\text{type}(C_1, C_2, D_1, D_2)$ is widening. Here, data words take their values in $\mathbb{Q}_+$, as we do not need that they belong to $\mathbb{N}$. This yields the expected pattern by taking $u = s \cdot u'$, $u_1 = s_1 \cdot u'_1$, $u_2 = s_2 \cdot u'_2$, and $v = v'$, $v_1 = v'_1$, $v_2 = v'_2$. We have that $u_1 \not\parallel u_2$, since $s_1 \not\parallel s_2$. $\square$

Decision of Functionality

This characterisation yields the decidability of the functionality problem for transducers over $(\mathbb{N}, <, 0)$. We just need to show that we can slightly

We preferred to first provide a stronger pattern to accentuate the proximity with the oligomorphic case, even though it implies a bit of additional work.

weaken the pattern, to highlight the fact that its two parts can be decided independently, and to get a small witness property on the mismatch from our study of the oligomorphic case (Lemma 12.33).

Proposition 12.63 *There exists a polynomial $P(x, y, z)$ such that the following holds.*

Let T be some non-deterministic register transducer over $(\mathbb{N}, <, 0)$ with n states, k registers and maximum output length M . T is not functional if and only if there exists two pairs of partial runs of $T_{\mathbb{Q}_+}$ in $\mathbb{Q}_+$ that satisfy

- (a) *There is a pair of configurations that is reachable by two ‘short’ partial runs that yield a mismatch:*

$$\begin{aligned} C^t &\xrightarrow[T_{\mathbb{Q}_+}]{u|u_1} B_1 && \text{where } |u| \leq P(\mathcal{R}_{\mathbb{Q}_+}(4k), n, M) \\ C^t &\xrightarrow[T_{\mathbb{Q}_+}]{u|u_2} B_2 && \text{and } u_1 \not\parallel u_2 \end{aligned}$$

- (b) *There is a candidate synchronised loop L that is final and that starts from a pair of configurations that has the same type as (B_1, B_2) :*

$$\begin{aligned} &\underbrace{C_1 \xrightarrow[T_{\mathbb{Q}_+}]{v|v_1} D_1}_{L} \\ &\underbrace{C_2 \xrightarrow[T_{\mathbb{Q}_+}]{v|v_2} D_2} \end{aligned} \quad \text{where } \text{type}(C_1, C_2) = \text{type}(B_1, B_2)$$

Let us stress out once again that in the statement, $u, u_1, u_2, v, v_1, v_2, w, w_1, w_2 \in (\mathbb{Q}_+)^*$.

Proof. We start with the left-to-right direction, which relies on the ‘small mismatch witness’ Lemma 12.33. By Proposition 12.63, we get the above pattern with additionally $B_1 = C_1, B_2 = C_2$. Since both partial runs $C^t \xrightarrow[T_{\mathbb{Q}_+}]{u|u_1} B_1$ and $C^t \xrightarrow[T_{\mathbb{Q}_+}]{u|u_2} B_2$ are in $(\mathbb{Q}_+, <, 0)$, which is oligomorphic, we can apply Lemma 12.33 to bound the length of u by $P(\mathcal{R}_{\mathbb{Q}_+}(4k), n, M)$, where P is the polynomial given in the statement of the lemma. This yields the result.

Conversely, assume that we have the above pattern. Since $\text{type}(C_1, C_2) = \text{type}(B_1, B_2)$, we know there exists $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that $\mu(C_1, C_2) = B_1, B_2$. Thus, by applying μ to L , we get that

$$\begin{aligned} &\underbrace{C^t \xrightarrow[T_{\mathbb{Q}_+}]{u|u_1} B_1 \xrightarrow[T_{\mathbb{Q}_+}]{\mu(v)|\mu(v_1)} \mu(D_1)}_{\mu(L)} \\ &\underbrace{C^t \xrightarrow[T_{\mathbb{Q}_+}]{u|u_2} B_2 \xrightarrow[T_{\mathbb{Q}_+}]{\mu(v)|\mu(v_2)} \mu(D_2)} \end{aligned}$$

where $\mu(L)$ is a candidate synchronised loop that is final since L is, and $u_1 \not\parallel u_2$. $\square$

It remains to show that this pattern can be checked in polynomial space, to get:

Theorem 12.64 ([142, Corollary 4.20]) *The functionality problem for functions defined by non-deterministic register transducers over $(\mathbb{N}, <, 0)$ is PSPACE-complete.*

Proof. We describe an algorithm that checks the pattern of Proposition 12.63 in polynomial space. It initially guesses $\text{type}(B_1, B_2) := \sigma$ (which is equal to $\text{type}(C_1, C_2)$), which can be stored in polynomial space, since it amounts to a quantifier-free formula of size quadratic in $2|R|$ (see Equation 12.2). Checking item (a) is done as in the oligomorphic case (see the proof of Theorem 12.40). Construct a register automaton over finite data words over $(\mathbb{Q}_+, <, 0)$ that recognises interleaved traces of partial runs of T whose inputs match and whose outputs mismatch before $P(\mathcal{R}_{\mathbb{Q}_+}(4k), n, M)$, then encounter (B_1, B_2) that has type σ . It can be described in polynomial space: equip $A_{\otimes}^T$ with an additional register that checks that inputs match, and another additional register along with a $P(\mathcal{R}_{\mathbb{Q}_+}(4k), n, M)$ -bounded counter that checks that output mismatch. Recall that adding a B-bounded counter can be done in space logarithmic in B (Section B.2.3). Checking that (B_1, B_2) has type σ can be done using tests on the transitions, since σ is given as a quantifier-free formula.

Checking item (b) is done in a similar way: one can again construct a register automaton of polynomial size to check it (see the proof of Lemma 12.61). It initially guesses a pair of configurations (C_1, C_2) , and checks that they have type σ . Then, it maintains in memory the type of (C_1, C_2, E_1, E_2) , where (E_1, E_2) is the current pair of configurations. If, at some point, such a type is widening, the automaton accepts.

Both automata can be described in polynomial space, so checking whether they are empty can be done in polynomial space (Theorem 12.27), as $(\mathbb{Q}_+, <, 0)$ is polynomially decidable (Example 12.4). $\square$

12.3.3. ω -Computability and Continuity Coincide

We now show that ω -computability and continuity coincide over $(\mathbb{N}, <, 0)$. As for the oligomorphic case, it amounts to showing that the next-letter problem is computable.

Proposition 12.65 *Let $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$ be a function defined by a non-deterministic register transducer over $(\mathbb{N}, <, 0)$. Its next-letter function Next_f is computable.*

Proof. The proof is, *mutatis mutandis*, the same as for Proposition 12.38. The only missing block is to decide emptiness for register automata over $\mathbb{N}$ with distinguished constants, i.e. over $(\mathbb{N}, <, 0 \cup \mathbb{C})$, where $\mathbb{C} \subset_f \mathbb{N}$. This is a consequence of [39, Theorem 14]. The result can also be obtained from Theorem 12.73 (see, in particular, Corollary 12.75), since $(\mathbb{N}, <, 0 \cup \mathbb{C})$ can be obtained as a quantifier-free interpretation of $(\mathbb{N}, <, 0)$ (it suffices to *saturate* $\mathbb{C}$, i.e. to include all constants between 0 and $\max \mathbb{C}$). $\square$

As a consequence, we have, with Theorems 11.2 and 11.3:

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

It could even be made linear by simply storing the ordering of the registers, plus whether they are equal to 0 or not.

We have written the proof in Section C.0.1, just in case.

[39]: Segoufin and Torunczyk (2011), ‘Automata based verification over linearly ordered data domains’

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

Theorem 12.66 ([142, Theorem 4.22]) Let $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$ be a function defined by a non-deterministic register transducer T over $(\mathbb{N}, <, 0)$.

- (i) f is ω -computable iff f is continuous
- (ii) f is Cauchy computable iff f is Cauchy-continuous
- (iii) f is uniformly computable iff f is uniformly continuous
- (iv) f is m -computable iff f is m -continuous, for all $m : \mathbb{N} \rightarrow \mathbb{N}$.

12.3.4. Characterisation and Decision of Uniform Continuity

We now turn to uniform continuity over $(\mathbb{N}, <, 0)$, and show that it is enough to consider uniform continuity over $(\mathbb{Q}_+, <, 0)$ and restrict our attention to configurations that are candidate co-reachable in $\mathbb{N}$, i.e. from which a candidate loop in $\mathbb{N}$ is reachable.

Definition 12.19 (Candidate co-reachable in $\mathbb{N}$) Let T be a transducer over $(\mathbb{Q}_+, <, 0)$. A $\mathbb{Q}_+$ -configuration B is *candidate co-reachable in $\mathbb{N}$* if there exists

$C, D \in \text{Configs}(T)$ such that $B \xrightarrow{u|u'} C \xrightarrow{v|v'} D$ for some $u, u' \in (\mathbb{Q}_+)^*$ where $C \xrightarrow{v|v'} D$ forms a candidate loop in $\mathbb{N}$ (i.e., $\text{type}(C, D)$ is widening).

Proposition 12.67 Let T be a non-deterministic register transducer over $(\mathbb{N}, <, 0)$ that recognises a function $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$. f is not uniformly continuous if and only if $T_{\mathbb{Q}_+}$ has a critical pattern

$$\begin{array}{l} C^l \xrightarrow{u|u_1}_{T_{\mathbb{Q}_+}} C_1 \xrightarrow{v|v_1}_{T_{\mathbb{Q}_+}} \mu(C_1) \xrightarrow{w|w_1}_{T_{\mathbb{Q}_+}} D_1 \\ C^l \xrightarrow{u|u_2}_{T_{\mathbb{Q}_+}} C_2 \xrightarrow{v|v_2}_{T_{\mathbb{Q}_+}} \mu(C_2) \xrightarrow{z|w_2}_{T_{\mathbb{Q}_+}} D_2 \end{array}$$

where D_1 and D_2 are each candidate co-reachable in $\mathbb{N}$.

Proof. Let T' be $T_{\mathbb{Q}_+}$, but restricted to configurations that are candidate co-reachable in $\mathbb{N}$. Note that candidate co-reachability only depends on the type of the configuration, and of the position of the state in the transition graph of the transducer, so this restriction can be done by adding a finite amount of information to the states. Let us show that T' recognises a uniformly continuous function if and only if T does.

$\Rightarrow$: If T is not uniformly continuous, then neither is T' First, note that runs of T are in particular runs of T' , so $\llbracket T \rrbracket \subseteq \llbracket T' \rrbracket$. Thus, if T recognises a function that is not uniformly continuous, then either T' is not even functional, or it recognises a function that is not uniformly continuous either.

$\Leftarrow$: the pattern yields a witness of non-uniform continuity in $\mathbb{N}$ Conversely, assume that T' is not uniformly continuous. The proof follows similar lines as in the oligomorphic case (Proposition 12.37), but we need to be careful to obtain a witness in $\mathbb{N}$. The idea is to first iterate the loop in $\mathbb{Q}_+$. Then, the obtained witness has a regular structure, that allows to

Recall that $T_{\mathbb{Q}_+}$ denotes T operating over data domain $(\mathbb{Q}_+, <, 0)$ instead of $(\mathbb{N}, <, 0)$.

Note that we do not ask that D_1 and D_2 are jointly candidate co-reachable in $\mathbb{N}$, simply that each one is.

obtain data words that are in $\mathbb{N}$ by multiplying by a large enough factor. By Proposition 12.37, it means that T' has a critical pattern

$$\begin{aligned} C' &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{w|w_1} D_1 \\ C' &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{z|w_2} D_2 \end{aligned}$$

such that D_1 and D_2 are co-reachable. By definition of T' , this means that D_1 and D_2 are candidate co-reachable in $\mathbb{N}$, so we can write

$$\begin{aligned} C' &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{w|w_1} D_1 \xrightarrow{s|s_1} E_1 \xrightarrow{t|t_1} G_1 \\ C' &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{z|w_2} D_2 \xrightarrow{s'|s_2} E_2 \xrightarrow{t'|t_2} G_2 \end{aligned}$$

where $\text{type}(E_1, G_1)$ and $\text{type}(E_2, G_2)$ are widening. Note that we do not need to have $|s| = |s'|$, nor $|t| = |t'|$.

Actually, we could obtain $|s| = |s'|$ by iterating each loop (E_1, G_1) and (E_2, G_2) to reach the least common multiple of $|s|$ and $|s'|$, idem for t and t' , but this is not necessary here.

Iterate the loop of the pattern, in $\mathbb{Q}_+$ Let $i \geq 0$ be a mismatch position in the pattern, i.e. such that

- (a) $u_1[i] \neq u_2[i]$, or
- (b) $v_2 = \varepsilon$ and $u_1[i] \neq u_2 \cdot w_2[i]$
- (c) $v_1 = v_2 = \varepsilon$ and $u_1 \cdot w_1[i] \neq u_2 \cdot w_2[i]$

We need to show that for all $j \in \mathbb{N}$, there exists $x_j, y_j \in \mathbb{N}^\omega$ such that $|x_j \wedge y_j| \geq j$ but $|f(x_j) \wedge f(y_j)| \leq i$. Thus, let $j \geq 0$. We iterate j times the loop between (C_1, C_2) and $\mu(C_1, C_2)$. We get that

Note that we show not only that T is uniformly continuous iff T' is, but also that they have the same modulus of continuity.

$$\begin{aligned} C' &\xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} \mu(C_1) \xrightarrow{\mu(v)|\mu(v_1)} \mu^2(C_1) \dots \xrightarrow{\mu^j(v)|\mu^j(v_1)} \mu^j(C_1) \xrightarrow{\mu^j(w)|\mu^j(w_1)} \mu^j(D_1) \\ C' &\xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} \mu(C_2) \xrightarrow{\mu(v)|\mu(v_2)} \mu^2(C_2) \dots \xrightarrow{\mu^j(v)|\mu^j(v_2)} \mu^j(C_2) \xrightarrow{\mu^j(w)|\mu^j(w_2)} \mu^j(D_2) \end{aligned}$$

are partial runs of T' . In the following, write $p = u \cdot v \dots \mu^j(v) \cdot \mu^j(w)$, $p' = u \cdot v \dots \mu^j(v) \cdot \mu^j(z)$, $p_1 = u_1 \cdot v_1 \dots \mu^j(v_1) \cdot \mu^j(w_1)$ and $p_2 = u_2 \cdot v_2 \dots \mu^j(v_2) \cdot \mu^j(w_2)$ (bear in mind that they depend on j). We have $|p \wedge p'| \geq |u| + j|v|$. By a similar reasoning as for the oligomorphic case, we also know that for all but maybe one $j \in \mathbb{N}$, we have $|p_1 \wedge p_2| \leq i$. We recall the main ideas of the argument: in items (a) and (c), μ^j is applied on both sides, so it preserves the mismatch, and we have the result for *all* $j \in \mathbb{N}$. However, the ‘but maybe one’ is due to the fact that in item (b), μ^j may cancel the mismatch between u_1 and $u_2 w_2$ if, by chance, $u_1[i] = u_2 \cdot \mu^j(w_2)[i]$. However, it cannot happen for two distinct $j \in \mathbb{N}$, otherwise it means that $u_1[i]$ is a fixpoint of μ , which implies that $u_1[i] = u_1 \cdot w_2[i]$ (see the proof of Proposition 12.37 for details).

Turning candidate loops to actual loops in $\mathbb{N}$ At this point, all data words are in $\mathbb{Q}_+$, and we only have candidate loops in $\mathbb{N}$. By Proposition

12.55, we know that there exists $\alpha \in \text{Aut}(\mathbb{Q}_+)$ such that $C' \xrightarrow{\alpha(p)|\alpha(p_1)} \alpha(\mu^j(D_1)) \xrightarrow{\alpha(s)|\alpha(s_1)} \alpha(E_1) \xrightarrow{\alpha(t)|\alpha(t_1)} \alpha(G_1)$ is a partial run in $\mathbb{N}$, and additionally $\alpha(E_1) \xrightarrow{\alpha(t)|\alpha(t_1)} \alpha(G_1)$ is a loop in $\mathbb{N}$. Symmetrically, since α pre-

serves the fact that $\text{type}(\alpha(E_2, G_2))$ is widening, by applying again Proposition 12.55 we obtain there exists $\beta \in \text{Aut}(\mathbb{Q}_+)$ such that $C^t \xrightarrow{\beta(\alpha(p'))|\beta(\alpha(p'_2))} \beta(\alpha(\mu^j(D_2))) \xrightarrow{\beta(\alpha(s'))|\beta(\alpha(s'_2))} \beta(\alpha(E_2)) \xrightarrow{\beta(\alpha(t'))|\beta(\alpha(t'_2))} \beta(\alpha(G_2))$ is a partial run in $\mathbb{N}$, and $\beta(\alpha(E_2)) \xrightarrow{\beta(\alpha(t'))|\beta(\alpha(t'_2))} \beta(\alpha(G_2))$ is a loop in $\mathbb{N}$. Observe that by construction, β preserves the fact that a pair of valuations is widening, since it widens all intervals. Note that it might map $\alpha(E_1)$ and $\alpha(G_1)$ to $\mathbb{Q}$ -valuations however. By letting $\gamma = \beta \circ \alpha$, we have that

$$\begin{aligned} C^t &\xrightarrow{\gamma(p)|\gamma(p_1)} \gamma(\mu^j(D_1)) \xrightarrow{\gamma(s)|\gamma(s_1)} \gamma(E_1) \xrightarrow{\gamma(t)|\gamma(t_1)} \gamma(G_1) \\ C^t &\xrightarrow{\gamma(p')|\gamma(p'_2)} \gamma(\mu^j(D_2)) \xrightarrow{\gamma(s')|\gamma(s'_2)} \gamma(E_2) \xrightarrow{\gamma(t')|\gamma(t'_2)} \gamma(G_2) \end{aligned}$$

are partial runs of $T_{\mathbb{Q}_+}$ such that $(\gamma(E_1), \gamma(G_1))$ is a widening pair of valuations, as well as $(\gamma(E_2), \gamma(G_2))$. Multiply all elements by some large enough factor K , to get partial runs in $\mathbb{N}$. This preserves the fact of being widening (Observation 12.53), so we get that $(K\gamma(E_1), K\gamma(G_1))$ and $(K\gamma(E_2), K\gamma(G_2))$ are looping, since they are widening and take their values in $\mathbb{N}$ (Lemma 12.51). As a consequence, there exists $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ such that $\zeta(K\gamma(E_1)) = K\gamma(G_1)$. Similarly, there exists an $\mathbb{N}$ -preserving $\eta \in \text{Aut}(\mathbb{Q}_+)$ such that $\eta(K\gamma(E_2)) = K\gamma(G_2)$ (note that they are distinct in general).

Wrap up We now have two loops in $\mathbb{N}$ (not necessarily synchronised), so we can iterate them to get accepting runs in $\mathbb{N}$. Thus, let $x_j = K\gamma(p) \cdot K\gamma(s) \cdot K\gamma(t) \cdot \zeta(K\gamma(t)) \cdot \zeta^2(K\gamma(t)) \dots$ and $y_j = K\gamma(p') \cdot K\gamma(s') \cdot K\gamma(t') \cdot \zeta(K\gamma(t')) \cdot \zeta^2(K\gamma(t')) \dots$. We have that $x_j, y_j \in \mathbb{N}^\omega$ are such that $|x_j \wedge y_j| \geq j$, since $|p \wedge p'| \geq j$ and we only applied morphisms. Then, $f(x_j) = K\gamma(p_1) \cdot K\gamma(s_1) \cdot K\gamma(t_1) \cdot \zeta(K\gamma(t_1)) \cdot \zeta^2(K\gamma(t_1)) \dots$ and $f(y_j) = K\gamma(p_2) \cdot K\gamma(s_2) \cdot K\gamma(t_2) \cdot \zeta(K\gamma(t_2)) \cdot \zeta^2(K\gamma(t_2)) \dots$, so $|f(x_j) \wedge f(y_j)| \leq i$ for all but maybe one $j \in \mathbb{N}$, since again, we only applied morphisms. We can now conclude that T is not uniformly continuous. $\square$

As a consequence, we get that uniform continuity is decidable over $(\mathbb{N}, <, 0)$. More precisely,

Theorem 12.68 ([142, Theorem 4.24]) *The uniform continuity problem for functions defined by non-deterministic register transducers over $(\mathbb{N}, <, 0)$ is PSPACE-complete.*

Proof. We are left to show that uniform continuity of the transducer that we built in the above proof T' is in PSPACE. Checking uniform continuity of a transducer over $(\mathbb{Q}_+, <, 0)$ is doable in polynomial space by Theorem 12.42. We have to be careful however, since computing T' explicitly may be too costly. Yet, checking if a configuration is candidate co-reachable in $\mathbb{N}$ can be done in polynomial space, using Proposition 12.57 and Theorem 12.58, since it amounts to finding a reachable candidate loop in $\mathbb{N}$.

The complexity lower bound can once more be obtained by a reduction from the emptiness problem of register automata over $(\mathbb{N}, =)$, which is PSPACE-complete [28, Theorem 5.1]. $\square$

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

12.3.5. Characterisation and Decision of Continuity

The Non-Guessing Case

We start with the case where non-deterministic reassignment is disallowed, for which the argument is simpler. This is essentially due to the fact that, from a sequence of accepting runs over inputs which have longer and longer common prefixes, we can extract a run (not necessarily accepting) over the limit of the inputs, since there are finitely many possible configurations after reading some input finite data word $w \in \mathbb{N}^*$ (more precisely, they take their values in $Q \times \text{elem}(w)^R$).

Proposition 12.69 *Let T be a non-guessing transducer over $(\mathbb{N}, <, 0)$ that recognises a function $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$. f is not continuous if and only if T has a critical pattern*

$$\begin{array}{c} C^l \xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} D_1 \xrightarrow{w|w_1} E_1 \\ \underbrace{\hspace{10em}}_{\mathbb{L}} \\ C^l \xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} D_2 \xrightarrow{z|w_2} E_2 \end{array}$$

such that $\mathbb{L}$ forms a candidate synchronised loop that is 1-final (i.e. C_1 is final), and E_1 and E_2 are each candidate co-reachable in $\mathbb{N}$.

Proof. Let T be a non-guessing transducer over $(\mathbb{N}, <, 0)$ that recognises a function $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$. Let Q be its set of states, and R its set of registers.

$\Rightarrow$: If f is not continuous, then it has the pattern First, assume that f is not continuous at some $x \in \mathbb{N}^\omega$. Let $i \geq 0$, along with a sequence $(x_n)_{n \in \mathbb{N}}$ such that for all $n \in \mathbb{N}$, $x_n \in \text{dom}(f)$ and $x_n \rightarrow_{n \infty} x$ but $|f(x_n) \wedge f(x)| \leq i$. Up to taking subsequences, we can assume that for all $n \in \mathbb{N}$, $|x_n \wedge x| \geq n$. For all $n \in \mathbb{N}$, denote ρ_n some accepting run of T that yields output $f(x_n)$, and write $\text{configs}(\rho_n) = C_0^n C_1^n \dots$. Since T is non-guessing, we know that for all $j \in \mathbb{N}$ and all $n \in \mathbb{N}$, $C_j^n \in Q \times (\text{elem}(x_n[:j]))^R$.

Extract a run of T from the ρ_n Let us extract from $(\rho_n)_{n \in \mathbb{N}}$ a run ρ on x such that for all $j \in \mathbb{N}$, $\rho[:j] = \rho_n[:j]$ for infinitely many $j \in \mathbb{N}$. We proceed by induction on the length of the partial run $\rho[:j]$. First, all ρ_n start with C^l , so take $\text{configs}(\rho)[0] = C^l$. Now, assume that ρ has been inductively constructed up to its j -th configuration $C_j = (q_j, v_j)$. We know that there are infinitely many ρ_m where $m \geq j$ such that $\rho_m[:j] = \rho[:j]$. Now, there are finitely many outgoing transitions from q_j . Since for all $m \geq j$, we have $|x_m \wedge x| \geq j$, we know that the C_{j+1}^n range in $Q \times (\text{elem}(x[:j]))^R$, which is finite. Thus, among the infinitely many ρ_m that coincide with ρ up to length j , there are infinitely many which have the same $j+1$ -th transition t_j and configuration C_{j+1} . Thus, we have that $C_j \xrightarrow[t_j]{x[j]|s_j} C_{j+1}$ for some output word $s_j \in \mathbb{N}^*$. By letting $\rho[:j+1] = \rho[:j] \cdot t_j C_{j+1}$, we get the inductive invariant at step $j+1$.

Note that the argument is quite similar to the proof of Proposition 12.28, which establishes that $\mathbb{D}^\infty / \text{Aut}(\mathbb{D})$ is compact.

Let $y' \in \mathbb{N}^\infty$ be the (possibly finite) output of ρ . By construction, $y' \leq f(x_m)$ for infinitely many $m \in \mathbb{N}$. Let π be an accepting run of T over input x that yields output $y = f(x)$. There are two cases, depending on whether y' is infinite.

► If $y' \in \mathbb{N}^\omega$, write

$$\begin{aligned} C^t &\xrightarrow{p|p_1} B_1 \xrightarrow{s|s_1} \\ C^t &\xrightarrow{p|p_2} B_2 \xrightarrow{s|s_2} \end{aligned}$$

where $p \cdot s = x$, $p_1 \cdot s_1 = y$, $p_2 \cdot s_2 = y'$ and $p_1 \not\parallel p_2$. By Lemma 12.61, we know that T admits a candidate synchronised loop L over $\mathbb{Q}_+$ that is reachable from B_1 and B_2 , i.e. there exists configurations C_1, C_2, D_1, D_2 in $\mathbb{Q}_+$ such that

$$\begin{aligned} C^t &\xrightarrow[T]{p|p_1} B_1 \xrightarrow[T]{u'|u'_1} C_1 \xrightarrow[T]{v|v_1} D_1 \xrightarrow{t|t_1} \\ C^t &\xrightarrow[T]{p|p_2} B_2 \xrightarrow[T]{u'|u'_2} C_2 \xrightarrow[T]{v|v_2} D_2 \xrightarrow{t|t_2} \end{aligned}$$

where $u' \cdot v \cdot t = s$, $u'_1 \cdot v_1 \cdot t_1 = s_1$, $u'_2 \cdot v_2 \cdot t_2 = s_2$, $\text{type}(C_1, C_2, D_1, D_2)$ is widening and C_1, D_1 are final. We then get a critical pattern by taking $u = p \cdot u'$ and $u_1 = p_1 \cdot u'_1$, $u_2 = p_2 \cdot u'_2$, since then $u_1 \not\parallel u_2$, as $p_1 \not\parallel p_2$. Take also $w = w_1 = w_2 = \varepsilon$ and $E_1 = D_1$, $E_2 = D_2$; they are both candidate co-reachable in $\mathbb{N}$ (actually, co-reachable in $\mathbb{N}$), as witnessed by t .

► If $y' \in \mathbb{N}^*$, we can write

$$\begin{aligned} C^t &\xrightarrow{p|p_1} B_1 \xrightarrow{s|s_1} \\ C^t &\xrightarrow{p|p_2} B_2 \xrightarrow{s|\varepsilon} \end{aligned}$$

where $p \cdot s = x$, $p_1 \cdot s_1 = y$, $p_2 = y'$ and $|p_1| > i$. By Lemma 12.61 we can extract a candidate synchronised loop that produces ε on its second component, i.e. we have

$$\begin{aligned} C^t &\xrightarrow[T]{p|p_1} B_1 \xrightarrow[T]{u'|u'_1} C_1 \xrightarrow[T]{v|v_1} D_1 \xrightarrow{t|t_1} \\ C^t &\xrightarrow[T]{p|p_2} B_2 \xrightarrow[T]{u'|\varepsilon} C_2 \xrightarrow[T]{v|\varepsilon} D_2 \xrightarrow{t|\varepsilon} \end{aligned}$$

Let m be such that ρ and ρ_m coincide up to D_2 , i.e. $|\rho \wedge \rho_m| > 2|p \cdot u' \cdot v|$. We know that $|f(x) \wedge f(x_m)| \leq i$. Write ρ_m as $C^t \xrightarrow{p|p_2} B_2 \xrightarrow{u'|\varepsilon} C_2 \xrightarrow{v|\varepsilon} D_2 \xrightarrow{z|w_2} E_2 \xrightarrow{t'|t'_2}$, where $p_2 \cdot u' \cdot w_2 \not\parallel p_1 \cdot u'_1$ (recall that $|p_1| > i$, so the mismatch necessarily occurs in p_1). We get a critical pattern by taking $u = p \cdot u'$, $u_1 = p_1 \cdot u'_1$, $u_2 = p_2 \cdot u'_2$, and E_1 such that $D_1 \xrightarrow{w|w_1} E_1$ for $w = t[:|z|]$ the prefix of length $|z|$ of t , and w_1 the corresponding output. We then have that E_1 and E_2 are each co-reachable in $\mathbb{N}$, as witnessed respectively by $t[:|z|]$ and t' , so they are in particular candidate co-reachable in $\mathbb{N}$.

It is slightly impolite to use again C_1 and C_2 , since they were used in another context, namely the definition of ρ . May the reader forgive this departure from good mathematical morals.

The '2' factor is due to the fact that runs consist of both configurations and transitions, so they are twice as long as their corresponding inputs.

The pattern yields a point of non-continuity This direction is established in the same way as for uniform continuity, except that now we can fix x instead of building two sequences x_n and y_n , since we know that C_1 is final. We redo the reasoning, for completeness. Assume that T has the pattern of the statement:

$$\begin{array}{c} C' \xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} D_1 \xrightarrow{w|w_1} E_1 \\ \underbrace{\hspace{10em}} \\ L \\ \underbrace{\hspace{10em}} \\ C' \xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} D_2 \xrightarrow{z|w_2} E_2 \end{array}$$

where L is a candidate synchronised loop, C_1 is final, and E_1 and E_2 are each candidate co-reachable in $\mathbb{N}$. By Lemma 12.60, we know that there exists a morphism $\mu \in \text{Aut}(\mathbb{Q}_+)$ such that

$$\begin{array}{c} \mu(C') \xrightarrow{\mu(u)|\mu(u_1)} \mu(C_1) \xrightarrow{\mu(v)|\mu(v_1)} \mu(D_1) \xrightarrow{\mu(w)|\mu(w_1)} \mu(E_1) \\ \underbrace{\hspace{10em}} \\ \mu(L) \\ \underbrace{\hspace{10em}} \\ \mu(C_2) \xrightarrow{\mu(u)|\mu(u_2)} \mu(C_2) \xrightarrow{\mu(v)|\mu(v_2)} \mu(D_2) \xrightarrow{\mu(z)|\mu(w_2)} \mu(E_2) \end{array}$$

are partial runs in $\mathbb{N}$ and $C'_1 \xrightarrow{v'|v'_1} D'_1$ and $C'_2 \xrightarrow{v'|v'_2} D'_2$ form a synchronised loop L in $\mathbb{N}$, where we denote $C'_1 = \mu(C_1)$, $u' = \mu(u)$, etc. Thus, there exists $\zeta \in \text{Aut}(\mathbb{Q}_+)$ that preserves $\mathbb{N}$ and such that $\zeta(C'_1, C'_2) = (D'_1, D'_2)$. By the same construction as for Proposition 12.67, we can apply another morphism v so that $v(\mu(L))$ is a synchronised loop in $\mathbb{N}$ and both $v(E'_1)$ and $v(E'_2)$ are co-reachable in $\mathbb{N}$. Thus, let $x = v(u') \cdot v(v') \cdot \zeta(v(v')) \cdot \zeta^2(v(v')) \dots$, and let, for all $n \in \mathbb{N}$, $x_n = v(u') \cdot v(v') \cdot \zeta(v(v')) \cdot \dots \cdot \zeta^n(v(v')) \cdot \zeta^n(v(z)) \cdot \zeta^n(t)$, where t is such that $v(E_2) \xrightarrow{t|t_2} \text{ for some } t_2 \in \mathbb{N}^\omega$. We have that $f(x) = v(u'_1) \cdot v(v'_1) \cdot \zeta(v(v'_1)) \cdot \zeta^2(v(v'_1)) \dots$, and, for all $n \in \mathbb{N}$, $f(x_n) = v(u'_2) \cdot v(v'_2) \cdot \zeta(v(v'_2)) \cdot \dots \cdot \zeta^n(v(v'_2)) \cdot \zeta^n(v(w_2)) \cdot \zeta^n(t_2)$. Since ζ preserves $\mathbb{N}$, and v was chosen so that every element lie in $\mathbb{N}$, we get that $x, x_n, f(x), f(x_n) \in \mathbb{N}^\omega$. For all $n \in \mathbb{N}$, we have that $|x \wedge x_n| \geq |u| + n|v| \geq n$. Now, since $u_1 \cdot w_1 \not\parallel u_2 \cdot w_2$, we get, by the same reasoning as for uniform continuity, that $|f(x) \wedge f(x_n)| \leq i$ for all but maybe one $n \in \mathbb{N}$. This witnesses that f is not continuous at $x \in \mathbb{N}^\omega$. $\square$

Remark 12.10 Note that the strategy of extracting a run from the sequence of ρ_n could also be applied to the case of $(\mathbb{D}, =)$ and of oligomorphic data domains. We did not do so at the time since we could then bound the length of partial runs that do not contain loops, which yielded conceptually simpler arguments (in particular, avoiding the use of Ramsey's theorem*). Moreover, we found it relevant to present different proof techniques, to show the different ways in which the problem can be approached.

* At this point, the reader might start entertaining the idea that the author has a grudge against the highly esteemed Ramsey's theorem. This is actually quite the contrary. The rationale behind this is that powerful tools should be used sparingly: 'with great power comes great responsibilities'. For instance, using it when a pumping argument suffices might obfuscate the matters.

This characterisation again yields a polynomial space algorithm to decide whether a function recognised by a non-guessing register transducer is continuous.

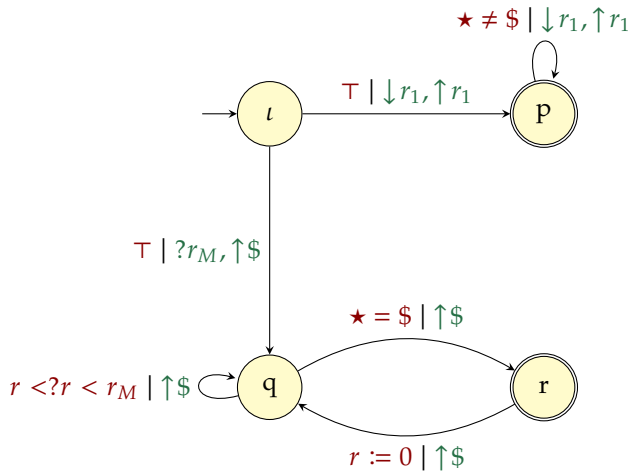
Theorem 12.70 *The continuity problem for functions defined by non-guessing non-deterministic register transducers over $(\mathbb{N}, <, 0)$ is PSPACE-complete.*

Proof. The proof is the same as for uniform continuity. It suffices to note that checking whether C_1 is final can be done by adding one bit of information to the states of the register automaton that recognises the pattern. $\square$

Allowing Guessing

When non-deterministic reassignment is allowed, it might be that in the pattern of Proposition 12.69, the bottom path $C_2 \xrightarrow{v|v_2} D_2$ is quasi-looping, i.e. can be iterated arbitrarily many times, but not infinitely many (see Example 12.5). Thus, this notion of quasi-looping has to show through the pattern that characterises non-continuity. In this section, we give the main ideas on how to get a characterisation of continuity in $(\mathbb{N}, <, 0)$ in the presence of guessing.

Example 12.6 For instance, consider the transducer of Figure 12.10. It recognises the function $f_{\$}^B : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$ such that $f_{\$}^B(x) = x$ if x does not contain $\$$, and $f_{\$}^B(x) = (\$)^\omega$ for all x that contain infinitely many occurrences of $\$$ that are separated by blocks of length at most M , for some $M \in \mathbb{N}$. It is otherwise undefined, i.e. its domain is $\text{dom}(f_{\$}^B) = 1^\omega \sqcup \{b_1 \cdot \$ \cdot b_2 \cdot \$ \cdot \dots \mid \exists M \in \mathbb{N}, \forall k \in \mathbb{N}, |b_k| \leq M\}$. $\$$ can be chosen as 0, but it can also be some constant added to the domain, or any distinguished integer. This transducer does not have the pattern of Proposition 12.69,



We did not explicitly chose 0 to highlight the fact that it is the minimal element of $\mathbb{N}$ plays no role here.

Figure 12.10: A non-deterministic register transducer that recognises $f_{\$}$ of Example 12.6. It initially guesses whether $\$$ appears in the input, and checks that its guess is correct along the run. It also initially guesses an upper bound on the length before reading $\$$, that constrains the number of iterations of the self-transition over state q . $r < ?r < r_M$ is not allowed in the syntax; it is meant to consist in guessing a value that lies between r and r_M , and storing it in r . This can be implemented by adding another register r' that stores this value, and checks on the next step that it is indeed between r and r_M , then interverting the roles of r and r' and so on. Recall that here, registers are initialised with 0. $r := 0$ is not allowed in the syntax either, and consists in reassigning r to 0. This can be done with reassignment, by guessing the content of r and checking that it is 0 on the next step, or by storing in the states the fact that it is supposed to be equal to 0, and modify the tests accordingly. See the proof of Proposition 4.6 for more details on the latter.

since the value of r_M that is initially guessed sets a bound on the number of times the self-transition over state q can be taken; and indeed, it

induces a type that is not widening. Then, once the transition to state r is taken, the input mismatch so it cannot yield a critical pattern anymore. However, the self-transition over q can be taken arbitrarily many times, for a sequence of inputs that have longer and longer common prefixes, where the occurrence of $\$$ is farer and farer in the input. For instance, take $\$ \neq 1$ and let $x_n = (1^n \$)^\omega$. By guessing $r_M = n$, the bottom run is accepting and yields output $\$^\omega$, which is a witness of non-continuity since $f(1^n) = 1^\omega \neq \$^\omega$. Intuitively, r and r_M are allowed to have this ill-mannered attitude because their values does not appear in the input, and can thus vary independently.

Remark 12.11 Note that $f_\$^B$ cannot be recognised by a non-guessing transducer. This can be shown by noting that the restriction to the finite alphabet $\{0, 1\}$, where we take $\$ = 0$, is not ω -regular. Indeed, its domain is then $\text{dom}(f_\$^B) = 1^\omega \sqcup \{1^{n_1} \cdot 0 \cdot 1^{n_2} \cdot 0 \cdot \dots \mid \exists M \in \mathbb{N}, \forall k \in \mathbb{N}, n_k \leq B\}$. This ω -language is however ωB -regular, since we can write $\text{dom}(f_\$^B) = 1^\omega \sqcup (1^B 0)^\omega$, using the syntax of [59, Section 2]. Non-guessing register transducers can have a non- ω -regular behaviour, in the sense that the accepting runs do not have an ω -regular structure (see Property 4.38 for register automata); however when they are restricted to a finite alphabet of data values, they always are ω -regular (Proposition 10.3).

[59]: Bojańczyk and Colcombet (2006), ‘Bounds in ω -Regularity’

The main idea is to restrict the pattern of Proposition 12.69 to registers that take values that belong to the input, so that we can again show that their type is widening, using similar arguments as in the non-guessing case. Actually, we only need to exclude registers whose value remains above the maximal value that appears in the input.

Assumption 12.20 In the following, we assume that automata and transducers over $(\mathbb{Q}_+, <, 0)$ and $(\mathbb{N}, <, 0)$ are equipped with a special register $r_{\max}$ that stores the maximal value appearing in the input.

Given a transducer T , this is done by replacing each transition $p \xrightarrow{\phi | \text{regOp}, v}$ q with two transitions $p \xrightarrow{\star \leq r_{\max} \wedge \phi | \text{regOp}\{r_{\max} \leftarrow \text{keep}\}, v}$ q (the data value is below the maximal value) and $p \xrightarrow{\star > r_{\max} \wedge \phi | \text{regOp}\{r_{\max} \leftarrow \text{set}\}, v}$ q , where $\text{regOp}\{r_{\max} \leftarrow \text{instr}\}$ is the register operation regOp' obtained by setting $\text{regOp}'(r) = \text{regOp}(r)$ for all $r \neq r_{\max}$, and $\text{regOp}'(r_{\max}) = \text{instr}$. The construction for automata is similar; we do not spell it out as the focus is set on transducers.

A direct induction establishes that for all input $x \in \mathbb{N}^\omega$ and all runs of T on x we have, for all $i \in \mathbb{N}$, $v_i(r_{\max}) = \max x[:i]$, where $\text{valuations}(p) = v_0 v_1 \dots$.

Definition 12.21 Given a valuation $v : R \rightarrow \mathbb{N}$ such that $r_{\max} \in R$, we say that a register $r \in R$ is *orderly* if $v(r) \leq v(r_{\max})$. Otherwise, it is said to be *stray*.

We correspondingly generalise the notion of widening type to types that are widening over a subset $X \subseteq R$ of registers. In our characterisations, we later on ask for types that are widening over the subset of orderly registers.

Definition 12.22 (Widening type over a subset of registers) Let $R = \{r_1, \dots, r_k\}$ be a set of registers, and $R' = \{r'_1, \dots, r'_k\}$ be its primed copy.

Given a finite data word $w \in \mathbb{N}^*$, the notation $\max w$ is to be understood as $\max_{0 \leq j < |w|} w[j]$.

Let $\sigma(r_1, \dots, r_k, r'_1, \dots, r'_k)$ be a type of $2k$ -tuples that preserves types. For a set $X \subseteq R$, we say σ is widening over X if its subtype $\sigma|_{X \sqcup X'}$ is widening, where $\sigma|_{X \sqcup X'}$ is the only type τ over variables $X \sqcup X'$ such that $\sigma \wedge \tau$ is valid. We spell out what this means for clarity:

- (a) The value of each register in X increases between ν and λ : for all $r \in X$, $\sigma \Rightarrow r \leq r'$ is valid in $(\mathbb{Q}_+, <, 0)$, i.e. all valuations $\nu, \lambda : R \rightarrow \mathbb{Q}_+$ that satisfy σ are such that $\nu(r) \leq \lambda(r)$, for all $r \in X$.
- (b) If a given orderly register is constant between ν and λ , then all registers below are also constant: for all $r, s \in X$, if $\sigma \Rightarrow s = s'$ and $\sigma \Rightarrow r \leq s$ are valid in $\mathbb{Q}_+$, then $\sigma \Rightarrow r = r'$ is valid in $\mathbb{Q}_+$.

By extension, for two valuations ν and λ over R such that $\text{type}(\nu) = \text{type}(\lambda)$, we say that (ν, λ) is *pseudo-widening over X* if $\text{type}(\nu, \lambda)$ is widening over X .

We say that (ν, λ) is *widening over X* when $\text{type}(\nu) = \text{type}(\lambda)$ and:

1. for all $r \in X$, $\lambda(r) \geq \nu(r)$
2. for all $r, s \in X$, $|\lambda(s) - \lambda(r)| \geq |\nu(s) - \nu(r)|$

Observe that pair of valuations that is widening over X is in particular pseudo-widening over X .

The notions of candidate loop and of candidate synchronised loop in $\mathbb{N}$ can then be generalised, by restricting to orderly registers:

Definition 12.23 (Candidate quasi-loop) Let A be a register automaton over $(\mathbb{Q}_+, <, 0)$ with a distinguished register $r_{\max}$ maintaining the maximal value seen in the input (cf Assumption 12.20). A *candidate quasi-loop in N* is a partial run $C \xrightarrow{u} D$ such that (C, D) is pseudo-widening over O , where $O = \{r \in R \mid D(r) \leq D(r_{\max})\}$ i.e. $\text{type}(C, D)$ is widening over O .

Definition 12.24 (Candidate synchronised quasi-loop) Let T be a non-deterministic register transducer, and consider a pair of partial runs L :

$$\begin{aligned} C_1 &\xrightarrow{u|u_1} D_1 \\ C_2 &\xrightarrow{u|u_2} D_2 \end{aligned}$$

where $C_1 : R_1 \rightarrow \mathbb{Q}_+$, $D_1 : R'_1 \rightarrow \mathbb{Q}_+$ and $C_2 : R_2 \rightarrow \mathbb{Q}_+$, $D_2 : R'_2 \rightarrow \mathbb{Q}_+$. Let $O_2 = \{r \in R_2 \mid D_2(r') \leq D_2(r'_{\max})\}$ be the set of orderly registers of D_2 . The pair L is a *candidate synchronised quasi-loop* if the pair of pairs of configurations $((C_1, C_2), (D_1, D_2))$ is pseudo-widening over $R_1 \sqcup O_2$. Recall that this is the case whenever $\text{type}(C_1, C_2, D_1, D_2)$ is widening over $R_1 \sqcup O_2$.

Proposition 12.71 Let T be a transducer over $(\mathbb{N}, <, 0)$ (with guessing) that recognises a function $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$. f is not continuous if and only if T has a

Note that given a type τ over variables R represented as a function $f_\tau : R \times R \rightarrow \{<, >, =\}$, its subtype over $X \subseteq R$ is given by the restriction of f_τ to $X \times X$, i.e. $f_{\tau|_X} = (f_\tau)|_{X \times X}$.

When we write $C_1 : R_1 \rightarrow \mathbb{Q}_+$, we actually mean that $\nu_1 : R_1 \rightarrow \mathbb{Q}_+$, where $C_1 = (q_1, \nu_1)$ for some $q_1 \in Q$ (where Q is the state space of T). We use primed and indexed copies of registers to easily distinguish between the different configurations).

critical pattern

$$\begin{array}{c}
 C^i \xrightarrow{u|u_1} C_1 \xrightarrow{v|v_1} D_1 \xrightarrow{w|w_1} E_1 \\
 \underbrace{\hspace{10em}} \\
 \text{L} \\
 \underbrace{\hspace{10em}} \\
 C^i \xrightarrow{u|u_2} C_2 \xrightarrow{v|v_2} D_2 \xrightarrow{z|z_2} E_2
 \end{array}$$

such that L forms a candidate synchronised quasi-loop that is 1-final (i.e. C_1 is final), and E_1 and E_2 are each candidate co-reachable in $\mathbb{N}$.

Remark 12.12 If a transducer is non-guessing, there are no stray registers, so we obtain the pattern of Proposition 12.69.

Proof idea. We start by outlining the proof, and then detail the main arguments. The formal proof is omitted, so as not to trespass upon the reader's patience[†]. First, to exhibit the pattern, the central idea is that one can again extract a limit run when restricting to orderly registers. Indeed, for any input $x \in \mathbb{N}^\omega$, at each index $i \in \mathbb{N}$, orderly registers take their values in the bounded set $\{0, \dots, \max x[:i]\}$, which is sufficient for the extraction. Then, we can apply a Ramsey argument to get a subsequence of configurations that all have the same pairwise type σ , and show that if it were not widening over orderly registers, this would yield an infinite decreasing chain or an infinite increasing bounded chain, which is impossible in $\mathbb{N}$. The fact that the pattern is critical is proved as in the non-guessing case. The other direction also requires a bit more work than in the non-guessing case, where obtaining a discontinuity point from the pattern is reasonably easy. Indeed, since there is no condition on stray registers, we cannot simply iterate the loop, instead we have to find values for them for each number n of iterations of the quasi-loop $C_2 \xrightarrow{v|v_2} D_2$, which means leaving sufficient room between them to be able to iterate n times.

We now give a few more details. Let T be a functional transducer over $(\mathbb{N}, <, 0)$ recognising a function $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$ that is not continuous at some $x \in \mathbb{N}^\omega$, and let $i \geq 0$ and $(x_n)_{n \in \mathbb{N}}$ be a sequence of elements of $\text{dom}(f)$ that converges to x such that for all $n \in \mathbb{N}$, $|x \wedge x_n| \geq n$ but $|f(x_n) \wedge f(x)| \leq i$. Correspondingly, let ρ be an accepting run of T yielding output $f(x)$, and for each $n \in \mathbb{N}$, let ρ_n be an accepting run of T yielding output $f(x_n)$. We build by induction a sequence $\rho' = v'_0 t'_0 v'_1 t'_1 \dots$ such that for all $j \geq 0$, by letting $O_j = \{r \in R, v_j(r) \leq r_{\max}\}$, $\rho_n|_{O_j}[:j] = \rho'|_{O_j}[:j]$, where, given a run ρ and $X \subseteq R$, we define $\rho|_X$ as $v_{0|X} t_{0|X} v_{1|X} t_{1|X} \dots$. This construction relies on the same arguments as Proposition 12.69, since $Q \times \{0, \dots, \max x[:j]\}$ is finite, so for a given $j \geq 0$, there are only finitely many possible $\rho_n|_{O_j}[:j]$. Note that this construction does not work if we lift the restriction to orderly registers (e.g. for the transducer of Example 12.6), since then valuations take their values in the entire domain $\mathbb{N}$ (the transducer can guess arbitrarily large values). Consider the runs ρ and ρ' , we can extract, by applying the Ramsey argument of Proposition 12.56 to valuations $v_i \sqcup v'_i$, an infinite sequence of valuations that have the same pairwise type σ . In particular, they all have the same type, hence the same orderly registers O . If σ is not

[†] Note that the writer belongs to the set of readers.

widening over O , we are able to derive an infinite descending chain or an infinite increasing bounded chain in $\mathbb{N}$, which yields a contradiction, using the same arguments as in Proposition 12.56.

The other direction relies on the study of candidate synchronised loops: by isolating stray registers, we are able to iterate the loop for an arbitrary number of times over data words that are prefix of each other. As for the non-guessing case, it consists in repeatedly applying a morphism; however, we can only ensure that it is $\mathbb{N}$ -preserving over orderly registers. Since stray registers are only compared with tests $\star < r$ for any $r \notin O$, they can take arbitrarily large values, so we can always pick values for them that satisfy all tests of any finite prefix. This allows, from a candidate synchronised loop, to get an x and a family $(x_n)_{n \in \mathbb{N}}$ such that $(x_n)_{n \in \mathbb{N}}$ converges to x , but $(f(x_n))_{n \in \mathbb{N}}$ does not, in the same way as for Proposition 12.69; the fact that the pattern is critical ensures that the images $(f(x_n))$ mismatch with $f(x)$ at some position. $\square$

The above characterisation allows to extend Theorem 12.70 to register transducer with guessing.

Theorem 12.72 ([142, Theorem 4.28]) *The continuity problem for functions defined by non-deterministic register transducers over $(\mathbb{N}, <, 0)$ is PSPACE-complete.*

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

Proof. One first equips the input transducer with a special register $r_{\max}$, to ensure Assumption 12.20; this is a linear transformation. Then, the decision of the above pattern is done in the same way as for Theorem 12.70; excluding stray registers poses no difficulty.

PSPACE-hardness results from PSPACE-hardness of the non-guessing case (Theorem 12.70). $\square$

12.3.6. Transferring the Study to Similar Structures

We have extended our study to one non-oligomorphic data domain, namely $(\mathbb{N}, <, 0)$, which is a substructure of $(\mathbb{Q}, <, 0)$. To extend it to other substructures of $(\mathbb{Q}, <, 0)$, e.g. $(\mathbb{Z}, <, 0)$, we could do the same work again and study the properties of loops in $\mathbb{Z}$. However, a simpler way is to observe that $(\mathbb{Z}, <, 0)$ can be simulated by $\mathbb{N}$, using two copies of the structure. We show a quite simple yet powerful result: given a data domain $\mathcal{D}$, if a data domain $\mathcal{D}'$ can be defined as a quantifier-free interpretation over $\mathcal{D}$, then the problems of emptiness, functionality, continuity, etc reduce to the same problems over $\mathcal{D}$.

Definition 12.25 A *quantifier-free interpretation of dimension l* with signature Γ over a structure $(\mathcal{D}, \mathcal{S})$ is given by quantifier-free formulas over $\mathcal{S}$:

- A formula $\varphi_{\text{domain}}(x_1, \dots, x_l)$
- For each constant symbol c of Γ , a formula $\varphi_c(x_1, \dots, x_l)$. Note that we do not assume the encoding to be unique.
- For each relational symbol R of Γ of arity r (including $=$), a formula

$$\varphi_R(x_1^1, \dots, x_l^1, \dots, x_1^r, \dots, x_l^r)$$

The structure $\mathcal{D}'$ is defined by the following:

- Its domain is $\mathbb{D}' = \{(d_1, \dots, d_l) \mid \mathcal{D} \models \varphi_{\text{domain}}(d_1, \dots, d_l)\}$
- Each constant symbol c is interpreted by the tuples $(d_1^c, \dots, d_l^c) \in \mathbb{D}'$ that satisfy φ_c , i.e. such that $\mathcal{D} \models \varphi_c(d_1^c, \dots, d_l^c)$
- For each relational symbol, an interpretation

$$R^{\mathcal{D}'} = \{(d_1^1, \dots, d_l^1, \dots, d_1^r, \dots, d_l^r) \mid \mathcal{D} \models \varphi_R(d_1^1, \dots, d_l^1, \dots, d_1^r, \dots, d_l^r)\}$$

Remark 12.13 If the encoding of constants is unique, i.e. there is exactly one tuple that satisfies φ_c for all c , then $\varphi_{=}$ simply consists in checking that the coordinates are equal, i.e. $\varphi_{=}(x_1, \dots, x_l, x'_1, \dots, x'_l) := \bigwedge_{1 \leq i \leq l} x_i = x'_i$.

Example 12.7 The data domain $(\mathbb{Z}, <, 0)$ can be defined as a QF-interpretation of $(\mathbb{N}, <, 0)$, by using two copies of $\mathbb{N}$. One handles the positive numbers, the other, the negative ones, and its order is correspondingly reversed. Formally, this is defined as the following 2-dimensional interpretation over $(\mathbb{N}, <, 0)$ (where the first dimension stores negative numbers, and the second, positive ones):

- $\varphi_{\text{domain}}(x, y) := (x = 0) \vee (y = 0)$
- $\varphi_0(x, y) := (x = 0) \wedge (y = 0)$
- $\varphi_{=}(x_1, y_1, x_2, y_2) := x_1 = x_2 \wedge y_1 = y_2$
- $\varphi_{<}(x_1, y_1, x_2, y_2) := (x_1 = x_2 = 0 \wedge (y_1 < y_2)) \vee (y_1 = y_2 = 0 \wedge (x_1 > x_2)) \vee (x_2 = y_1 = 0 \wedge \neg(x_1 = y_2 = 0))$.

Since the encoding of 0 is unique, $\varphi_{=}$ simply checks that coordinates are equal, see Remark 12.13.

[142]: Exibard et al. (2021), ‘Computability of Data-Word Transductions over Different Data Domains’

Theorem 12.73 ([142, Theorem 4.29]) *Let $\mathcal{D}$ be a data adomain, and $\mathcal{D}'$ be a quantifier-free interpretation over $\mathcal{D}$. There is a polynomial space reduction from all problems we studied over $\mathcal{D}'$ to their analogous over $\mathcal{D}$.*

More precisely, there is a polynomial space reduction from the non-emptiness problem (respectively, the functionality, continuity, Cauchy-continuity and uniform continuity problems) over $\mathcal{D}'$ to the non-emptiness problem (respectively, the functionality, continuity, Cauchy-continuity and uniform continuity problems) over $\mathcal{D}$.

Proof. Let $\mathcal{D}$ be a data adomain, and $\mathcal{D}'$ be a quantifier-free interpretation over $\mathcal{D}$. Let $R' \subseteq \mathbb{D}'^\omega \times \mathbb{D}'^\omega$ be a relation given by a non-deterministic register transducer T' over $\mathcal{D}'$. Let $l \in \mathbb{N}$ be the dimension of $\mathcal{D}'$ w.r.t. $\mathcal{D}$. Then, we can view R as a relation $R \subseteq (\mathbb{D}^l)^\omega \times (\mathbb{D}^l)^\omega$, where R is empty (respectively, functional, continuous, Cauchy-continuous, uniformly continuous) if and only if R' is.

Moreover, since $\mathcal{D}'$ is an interpretation, one can construct a non-deterministic register transducer T which recognises R . It uses l registers for every register of T' plus $l - 1$ registers to store the input. Its transitions are grouped by blocks of length l . In the first $l - 1$ transitions, it simply stores the input. On the l -th transition, it simulates a transition of T' , by substituting the formulas of the interpretation for the predicates. This can be done, as they are quantifier-free, so they match the syntax of tests. As usual, we do not construct T explicitly, but we are able to simulate it using only polynomial space. $\square$

As a direct corollary, we can transfer our results over $(\mathbb{N}, <, 0)$ to $(\mathbb{Z}, <, 0)$.

Corollary 12.74 *The problems of non-emptiness, functionality, continuity, Cauchy-continuity and uniform continuity over $(\mathbb{Z}, <, 0)$ are PSPACE-complete.*

Proof. Membership in PSPACE follows from Theorem 12.73, since $(\mathbb{Z}, <, 0)$ can be defined as a 2-dimensional quantifier-free interpretation of $(\mathbb{N}, <, 0)$ (Example 12.7).

PSPACE-hardness is obtained by observing that $(\mathbb{Z}, <, 0)$ can simulate $(\mathbb{D}, =)$, for whom the corresponding problems are already PSPACE-hard (see Section 12.1). $\square$

The results can also be extended to $(\mathbb{N}, <, \mathfrak{C})$ for $\mathfrak{C} \subset_f \mathbb{N}$ a finite set of distinguished constants. Here, the dimension depends on the maximal constant, but the problems are PSPACE-complete if $\mathfrak{C}$ is fixed.

Corollary 12.75 *Let $\mathfrak{C} \subset_f \mathbb{N}$ be a finite number of distinguished constants. The problems of non-emptiness, functionality, continuity, Cauchy-continuity and uniform continuity for relations defined by transducers over $(\mathbb{N}, <, \mathfrak{C})$ are PSPACE-complete.*

Proof. The proof again relies on showing that $(\mathbb{N}, <, \mathfrak{C})$ can be built as a quantifier-free interpretation of $(\mathbb{N}, <, 0)$. One first saturates $\mathfrak{C}$ into $\mathfrak{C}' = \{0, 1, \dots, M-1, M\}$, where $M = \max \mathfrak{C}$. Then, one can build a $\lceil \log M \rceil + 1$ -dimensional interpretation of $(\mathbb{N}, <, \mathfrak{C}')$ by encoding each constant in binary using the first $m = \lceil \log M \rceil$ coordinates. The last component consists in a copy of $\mathbb{N}$ that is strictly above all constants. Intuitively, it is shifted by an M factor, i.e. 0 is interpreted as $M+1$, 1 as $M+2$ etc.

We encode constants in binary as it is the most succinct. One could also encode them in unary, or using any encoding since we do not use any properties of the binary encoding.

- The domain consists in tuples that either encode constants or elements that are above M : $\varphi_{\text{domain}}(a_0, \dots, a_m, x) := x \neq 0 \Leftrightarrow \bigwedge_{0 \leq i \leq m} a_i = 0$
- For $c \in \mathfrak{C}'$, let $b_0 \dots b_m = {}^2c$ be the binary encoding of c . We define a formula φ_c that holds whenever $x = 0$ and for all $0 \leq i \leq m$, $a_i \neq 0 \Leftrightarrow b_i = 1$. Formally,

$$\varphi_c(a_0, \dots, a_m, x) := x = 0 \wedge \bigwedge_{\substack{0 \leq i \leq m \\ b_i = 0}} a_i = 0 \wedge \bigwedge_{\substack{0 \leq i \leq m \\ b_i \neq 0}} a_i \neq 0$$

Whether the binary encoding is little or big endian does not matter here.

Technically, we do not have to specify that $x = 0$ as it is already handled by φ_{domain} .

- Here, the encoding of constants is not unique, so the definition of equality is non-trivial. As for the definition of order (see below), the idea is, for each constant, to explicitly state to what elements it is equal, and separately treat the case of non-constants. Thus, we let $\varphi_=(a_0, \dots, a_m, x, a'_0, \dots, a'_m, y)$ be defined as

$$\bigvee_{c \in \mathfrak{C}} (\varphi_c(a_0, \dots, a_m, x) \wedge \varphi_c(a'_0, \dots, a'_m, y)) \vee (x \neq 0 \wedge y \neq 0 \wedge x = y)$$

- The order is defined in a similar fashion. $\varphi_<(a_0, \dots, a_m, x, a'_0, \dots, a'_m, y)$ holds when either:
 - $(a_0, \dots, a_m)$ and (a'_0, a'_m) respectively encode constants c and c' such that $c < c'$
 - The left element $(a_0, \dots, a_m)$ encodes a constant and the right one does not, i.e. $y \neq 0$
 - Neither encode a constant ($x \neq 0$ and $y \neq 0$) and $x < y$.

Formally, we let $\varphi_{<}(a_0, \dots, a_m, x, a'_0, \dots, a'_m, y)$ be

$$\bigvee_{\substack{c, c' \in \mathbb{C}' \\ c < c'}} \varphi_c(a_0, \dots, a_m, x) \wedge \varphi_{c'}(a'_0, \dots, a'_m, y) \vee \\ \left(\bigvee_{c \in \mathbb{C}'} \varphi_c(a_0, \dots, a_m, x) \right) \wedge y \neq 0 \vee \\ x \neq 0 \wedge y \neq 0 \wedge x < y$$

Then, membership in PSPACE (if $\mathbb{C}$ is fixed) is obtained by Theorem 12.73.

PSpace-hardness once more follows from the hardness results over $(\mathbb{N}, <, 0)$. $\square$

Remark 12.14 This encoding of constants is quite wasteful, and one could define a finer-grained notion of quantifier-free interpretation, in the spirit of what is done in [108, Definition 1] through the notion of ‘polynomial orbit-finite set’: instead of considering a single set of copies of the original domain, one considers labelled sets of copies, e.g. $\mathbb{Z}$ can be defined as $+\mathbb{N} + -\mathbb{N}$, and constants simply consists in labels of nullary copies, e.g. $(\mathbb{N}, <, \{0, 1\})$ is $\mathbb{N} + 1$. One then defines the formulas over all copies, to allow interactions between them (e.g. to specify that $1 > 0$). Theorem 12.73 is easily adapted to this more general notion.

[108]: Bojańczyk and Stefanski (2020), ‘Single-Use Automata and Transducers for Infinite Alphabets’

Remark 12.15 Both constructions above can be combined to get $(\mathbb{Z}, <, \mathbb{C})$. The transfer result also applies to many other substructures of $(\mathbb{Q}, <, 0)$, such as the ordinals $\omega + \omega$, $\omega \times \omega$, etc.

Remark 12.16 This ‘transfer result’ does not hold for first-order interpretations, which are defined as quantifier-free interpretations except that we allow $\varphi_{\text{domain}}(x_1, \dots, x_l)$, $\varphi_c(x_1, \dots, x_l)$ and $\varphi_R(x_1^1, \dots, x_l^1, \dots, x_1^r, \dots, x_l^r)$ to be unrestricted first-order formulas. For instance, $(\mathbb{N}, +1, 0)$ can be obtained from $(\mathbb{N}, <, 0)$ as a 1-dimensional first-order interpretation, by letting $\varphi_{+1}(x, y) := y > x \wedge \neg \exists z \cdot x < z < y$. The emptiness problem for register automata over this data domain is undecidable since they can recognise runs of Minsky machines (Theorem 4.52), so there can be no effective reduction from the problems over $(\mathbb{N}, +1, 0)$ to those over $(\mathbb{N}, <, 0)$.

And indeed, the construction of Theorem 12.73 fails when one encodes φ_{domain} , $(\varphi_c)_{c \in \mathbb{C}}$ and $(\varphi_R)_{R \in \mathbb{R}}$ as tests, since tests are restricted to be quantifier-free formulas.

Remark 12.17 A corollary of the above observation is that allowing tests to be unrestricted first-order formulas would yield a model of register automata whose emptiness is undecidable, already over data domain $(\mathbb{N}, <, 0)$. Yet, this would be possible in $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, 0)$, as they admit quantifier elimination.

Lifting the equivalence between ω -computability and continuity, known for transducers over finite alphabets [51], to the case of data words allows to characterise ω -computability through a forbidden pattern that is decidable for $(\mathbb{D}, =, \mathbb{C})$ and more generally for decidable oligomorphic data domains, for functions defined by non-deterministic asynchronous register transducers. This class of functions is moreover closed under composition, and decidable in the sense that one can decide, given a transducer, whether it recognises a function. We were also able to characterise the finer notion of uniform computability, equivalent to uniform continuity, that guarantees a uniform bound on the number of data values to read before being able to output something.

The jump in abstraction induced by the generalisation to oligomorphic data domains highlights the fact that to abstract the behaviour of register automata, the key point is that the transition dynamics can be described in first-order logic, which explains why we get non-deterministic reassignment for free in our development. Note also that adding a guessing mechanism is a somehow maximal extension, in the sense that over $(\mathbb{D}, =, \mathbb{C})$, this exactly captures nominal automata [25, Theorem 6.4], which model all that can be done with FO-definable transitions (this results from [26, Lemma 4.11], which equates equivariant and FO-definable sets).

Further Generalisations to Other Data Domains We were also able to extend our results to $(\mathbb{N}, <, 0)$ and its quantifier-free interpretations, which include e.g. $(\mathbb{Z}, <, \mathbb{C})$ for any finite $\mathbb{C} \subseteq \mathbb{Z}$, and more generally $(\mathbb{Z}^k, <, \mathbb{C})$ for any fixed $k \geq 1$. This was obtained using two properties: first, that $\mathbb{N}$ is a substructure of $\mathbb{Q}$ and second, that we were able to characterize iterable paths in $\mathbb{N}$. One possible extension would be to investigate data domains which have a tree-like structure, e.g. the infinite binary tree. There exists a tree-like oligomorphic structure, of which the binary tree is a substructure. Studying the iterable paths of the binary tree may yield a similar result as in the linear order case. Of course we could also extend to more general structures, namely those that are substructures of an oligomorphic structure and for which we are able to characterize iterable paths.

Two-Way Models In [51], the authors also show decidability of continuity (and ω -computability) for regular functions, i.e. functions that can be recognised by non-deterministic two-way transducers with look-ahead. To the best of our knowledge, there does not yet exist a two-way model operating over data words that is closed under composition, whose expressiveness is comparable or greater than that of non-deterministic register transducers and whose emptiness is decidable. For instance, deterministic two-way register automata have an undecidable emptiness problem [101, Theorem 5.3]. We explored generalisations of bimachines [153, 154], which provide a canonical model for rational functions [155, Theorem 2]. However, contrary to the finite alphabet case, composing more and

[51]: Dave et al. (2020), ‘Synthesis of Computable Regular Functions of Infinite Words’

[25]: Bojańczyk, Klin, and Lasota (2014), ‘Automata theory in nominal sets’

[26]: Bojańczyk (2019), *Atom Book*

Functions recognised by non-deterministic one-way transducers are called rational.

[101]: Neven, Schwentick, and Vianu (2004), ‘Finite state machines for strings over infinite alphabets’

[153]: Schützenberger (1961), ‘A Remark on Finite Transducers’

[154]: Eilenberg (1974), *Automata, languages, and machines. A*

[155]: Reutenauer and Schützenberger (1991), ‘Minimization of Rational Word Functions’

more left-to-right and right-to-left machines induces an infinite hierarchy, which implies that the model is not closed under composition. Other natural restrictions were considered, e.g. reversibility, but to no avail. Indeed, two-wayness over data words, unless very restricted, easily allows to recognise inputs with pairwise distinct data values, hence opening the way to the simulation of computations of Turing machines by using them as landmarks to navigate back-and-forth; in contrast, restrictions tend to break closure under composition. A promising formalism is the extension of streaming string transducers [156] to data words; in [108, Theorem 11] the authors show that the single-use restriction of streaming string transducers over data words (called SST with atoms) is shown equivalent to two-way single-use transducers. The non-deterministic version of streaming string transducer with atoms thus forms a good candidate to model functions over data words, and one can wonder whether continuity is decidable for this model.

Beyond the Functional Case This study was first motivated by synthesis applications. There are several classical ways of extending synthesis results, for instance considering larger classes of specifications, larger classes of implementations or both. In particular, an interesting direction is to consider non-functional specifications. However, as mentioned earlier, and as a motivation for studying the functional case, enlarging both (non-functional) specifications and implementations to an asynchronous setting leads to undecidability. Indeed, already in the finite alphabet setting, the synthesis problem of deterministic transducers over ω -words from specifications given by non-deterministic transducers is undecidable [48, Theorem 17]. A simple adaptation of the proof of [48] allows to show that in this finite alphabet setting, enlarging the class of implementations to any ω -computable function also yields an undecidable synthesis problem. Still, an interesting case is yet unexplored, already in the finite alphabet setting: given a synchronous specification, as an ω -automaton, is it possible to synthesise an ω -computable function realising it? In [49, 157], this question has been shown to be decidable for specifications with *total domain* (any input word has at least one correct output by the specification). More precisely, it is shown that realisability by a continuous function is decidable, and it turns out that the synthesised function is definable by a sequential transducer (a.k.a. input-deterministic transducer), hence it is ω -computable. When the domain of the specification is partial, the situation changes drastically: sequential transducers may not suffice to realise a specification that we know is realisable by an ω -computable function. This can be seen by considering the function f_{swap} of Example 11.2, casted to a finite alphabet $\{a, b, c\}$ (where c plays the role of $\$$): it is not computable by a sequential transducer, since it requires an unbounded amount of memory to compute this function. It has been shown recently that the problem of deciding the existence of a continuous implementation for specifications given by rational relations (over finite alphabets) becomes undecidable in this setting [158, Theorem 8], but the authors identify a decidable subclass, namely *weakly deterministic rational relations* [158, Theorem 2]. This implies in particular that the problem is decidable for synchronous rational specifications.

[156]: Alur and Cerný (2010), ‘Expressiveness of streaming string transducers’

[108]: Bojańczyk and Stefanski (2020), ‘Single-Use Automata and Transducers for Infinite Alphabets’

[48]: Carayol and Löding (2015), ‘Uniformization in Automata Theory’

[49]: Holtmann, Kaiser, and Thomas (2012), ‘Degrees of Lookahead in Regular Infinite Games’

[157]: Fridman, Löding, and Zimmermann (2011), ‘Degrees of Lookahead in Context-free Infinite Games’

Recall that a relation is rational if it is recognised by a one-way non-deterministic transducer.

[158]: Filiot and Winter (2021), ‘Continuous Uniformization of Rational Relations and Synthesis of Computable Functions’

Finer-Grained Measures of Continuity It is also worth noting that the measure we have chosen for continuity and ω -computability is quite demanding: as soon as two data words differ at a given index, the proximity between them at later positions is irrelevant. This is consistent with a notion of exact computability, where machines unerringly output data values along their runs, but fails to capture the handling of possible errors in the input and output. For instance, it might be that the input stream comes from a lossy channel, and that some data values are erroneous. In that case, it is important to be able to pinpoint the behaviour of the model over inputs that are close with regards to some error measure, e.g. the Hamming distance. This is however much more challenging, as it requires to jointly abstract a set of behaviours, while we could focus on single runs in our study.

Contributions of this thesis are twofold. First, we provide a study of the synthesis problem for register automata with different semantics (non-deterministic, universal and deterministic), over data domains $(\mathbb{D}, =, \mathbb{C})$, $(\mathbb{Q}, <, 0)$ and $(\mathbb{N}, <, 0)$. The undecidability border is never far, and often crossed, so we also study the register-bounded synthesis problem, which targets implementations represented by sequential register transducers whose number of registers is given as input. In the non-deterministic semantics, the test-free restriction is needed to go back to the realm of decidable problems. Universal semantics are more easily tamed, which is lucky since they are also more suitable to express specifications, as universal register automata are trivially closed under intersection. Moreover, we argue that it is often easier to first specify a forbidden behaviour, which is more naturally done using non-determinism, and then complement by dualising to a universal automaton.

The second set of contributions consists in an examination of ω -computability for functions defined by non-deterministic asynchronous register transducers. We obtained decidability for a wide range of data domains, and our study led us to a detailed analysis of the behaviour of register machines that should prove useful in other contexts.

This work constitutes a step in the direction of lifting synthesis to the case of infinite alphabets and, as usual in research, it opens up more perspectives than those it closed. Thus, as Sisyphus, let us now draw the prospects that arise from the lessons we have learned.

14.1. What Is Essential for Abstracting the Behaviour of Register Automata?

Lifting Register-Bounded Synthesis to Oligomorphic Domains As pointed out in Chapter 9 (Conclusion), the key ingredient to abstract the behaviour of register automata in a finite way is that there are finitely many types, which is captured by the notion of oligomorphicity. Thus, already in this thesis, the jump in abstraction induced by the study of this class of domains bounces back on Part I, as it allows to lift decidability of register-bounded synthesis to decidable oligomorphic data domains (decidability of a data domain being a reasonable computability assumption). Indeed, in Section 12.2.1, we showed that projection over labels is effectively ω -regular for non-deterministic register automata over decidable oligomorphic data domains (Theorem 12.24). This is sufficient for mimicking our construction for register-bounded synthesis over $(\mathbb{D}, =, \mathbb{C})$ and $(\mathbb{Q}, <, 0)$, which essentially relies on Lemma 6.13 (resp., Lemma 6.15), which states ω -regularity of the associated finite alphabet specification.

Action Sequences and Constraint Sequences Another key observation is the key role of action sequences to show ω -regularity of the

projection over labels, which makes them a powerful tool to abstract the behaviour of register automata. For more complex data domains as $(\mathbb{N}, <, 0)$, global phenomena come into play (e.g. infinite descending chains are forbidden), which necessitates to shift the focus to constraint sequences, that successfully capture these dynamics.

Unifying the Constructions for $(\mathbb{N}, <, 0)$ The techniques employed in Part I and Part II to analyse the behaviours of machines with registers over $(\mathbb{N}, <, 0)$ (respectively Sections 7.3.4 and 12.3) bear some similarities, yet we were not able to unify them. This deserves further exploration, as this would yield a finer understanding of the way those machines operate over this data domain.

14.2. Beyond Register Automata: Exploring Other Formalisms

For Specifications We concluded Part I by insisting on the versatility of the transfer theorem for the decision of register-bounded synthesis. And indeed, this might be the way to go for obtaining a logical formalism for which this problem is decidable. This could be the case of the two-variable fragment of some first-order logics over data words [134, 135]. Positive results also exist for the Logic of Repeating Values [33] in the unbounded case. This first calls for a more precise examination of the memory structure of winning strategies in these games, as advocated by the authors. Second, as for specifications given by register automata, it might be possible to obtain positive results for more expressive fragments in the register-bounded case. Besides, we ruled out alternating register automata due to the undecidability of their emptiness and universality problem. However, they are tightly related to LTL with the freeze quantifier [28, Theorem 4.1], so it is worth considering their 1-register restriction, which does not falls victim of the above undecidability results over finite data words. Other operational models such as data automata (or, equivalently, class-memory automata) were not considered for similar reasons. Given that little is known about them from a theoretical perspective, it might be premature to examine variants of the synthesis problem over them. However, knowledge about models for infinite alphabets is blossoming, and one should consider giving a second look once more is discovered.

For Implementations Since they are restricted to outputting data values that appear in the input that they have read so far, and to remember only finitely many of them, register transducers might not always be a well-suited model for implementations, and another direction is to consider more expressive formalisms, as was done in [44], which allows to store infinitely many data values, with a queue discipline. Depending on the target model, the transfer theorem can be generalised, as it relies on the existence of a connection between syntax and semantics, embodied by action sequences in the case of machines with registers.

[134]: Bojanczyk et al. (2006), ‘Two-Variable Logic on Words with Data’

[135]: Schwentick and Zeume (2012), ‘Two-Variable Logic with Two Order Relations’

[33]: Figueira, Majumdar, and Praveen (2020), ‘Playing with Repetitions in Data Words Using Energy Games’

[28]: Demri and Lazic (2009), ‘LTL with the freeze quantifier and register automata’

[44]: Ehlers, Seshia, and Kress-Gazit (2014), ‘Synthesis with Identifiers’

A Pattern Logic over Data Words In Part II, we were able to characterise non-continuity by a forbidden pattern for all the data domains we consider. This calls for a unified framework for the decision of the existence of such patterns, that would extend what has been done in the finite alphabet case with the pattern logic [149]. The model-checking problem of transducers against this logic is shown decidable by a reduction to the emptiness problem of Parikh automata. Over data words, it seems that a decision procedure can be obtained by equipping Parikh automata with registers, as this feature does not interact too much with the counters of the automaton, and both can thus be abstracted independently. Besides, this would simplify the presentation of the decision of related problems as functionality and emptiness, which would provide a better understanding of the behaviour of machines with registers. Parikh's theorem has recently been lifted to infinite alphabets for 1-register automata [159]; this should constitute a useful block to build on for the study of Parikh register automata.

[149]: Filiot, Mazzocchi, and Raskin (2020), 'A Pattern Logic for Automata with Outputs'

[159]: Hofman et al. (2021), 'Parikh's theorem for infinite alphabets'

14.3. Minimisation and Learning

The main goal of synthesis is to 'liberate the programmer', to quote the words of Harel [160]. Another way to achieve this goal is to learn the behaviour of the system from sample outputs. The term of 'learning' has lately undergone a lot of fancy uses (especially when prefixed with 'machine'); here, it should be understood in a formal way, rather than in its statistical acception. It consists in finding an acceptor for a language by being able to ask queries like 'does this word belong to the language?' or 'did I finally learn the right model?'. This line of work was initiated for deterministic finite-state automata in [161], and furthered for non-deterministic ones in [162, 163]. Recently, the learning problem for register automata has been the subject of attentive scrutiny [164, 165]. This problem is tightly linked with that of defining canonical representatives, and in particular to the minimisation problem, that asks to find a minimal machine for a given (data) language, studied in [23, 166]. It is known that sequential transducers can be minimised [167]. Is it the case of register transducers? We already know that it is undecidable whether a given partial function over data words can be recognised by a sequential register transducer, as this implies deciding whether its domain is recognisable by a deterministic register automaton. However, this does not imply undecidability of the minimisation problem. On top of learning applications, being able to minimise sequential register transducers would be useful to reduce the size of implementations given by our synthesis algorithms, that suffer from the state space blowup but are, by construction, sequential.

[160]: Harel (2008), 'Can Programming Be Liberated, Period?'

[161]: Angluin (1987), 'Learning Regular Sets from Queries and Counterexamples'

[162]: Denis, Lemay, and Terlutte (2002), 'Residual Finite State Automata'

[163]: Bollig et al. (2009), 'Angluin-Style Learning of NFA'

[164]: Bollig et al. (2013), 'A Fresh Approach to Learning Register Automata'

[165]: Moerman et al. (2017), 'Learning nominal automata'

[23]: Benedikt, Ley, and Puppis (2010), 'What You Must Remember When Processing Data Words'

[166]: Cassel et al. (2015), 'A succinct canonical register automaton model'

[167]: Choffrut (2003), 'Minimizing subsequential transducers: a survey'

Nominal transducers extend transducers in the same way as nominal automata extend finite-state automata, and, over $(\mathbb{D}, =, \mathbb{C})$, they correspond to register transducers. For this model, the characterisation of sequentiality through a syntactic congruence given by Choffrut in [167, Theorem 1] again holds in the case of nominal transducers, by replacing 'finite' with 'orbit-finite'. Indeed, it does not rely on finiteness of the alphabet. We know that this criterion is undecidable, since sequentiality is undecidable; what remains to be seen is whether one can derive a

minimisation algorithm from it. For that, one needs a minimality criterion, which is not as obvious as in the finite alphabet case, since there are two relevant parameters, namely the number of states and of registers. In [23], it is shown that by constraining register automata to update their registers following a stack discipline, one gets a minimisation algorithm. Such constraint however lacks succinctness: in general, machines that follow it can be exponentially bigger. One could also resort to the construction of [25, Section 11], which yields canonical representations, although it is mathematically involved due to its genericity.

[25]: Bojańczyk, Klin, and Lasota (2014),
'Automata theory in nominal sets'

APPENDIX

Graphical Conventions

A.

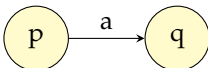
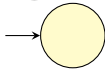
A.1. Relations and Functions, Input and Output

- ▶ Input is coloured **red**
- ▶ Output is coloured **green**
- ▶ Functions are coloured **green**
- ▶ Relations are coloured **blue**

A.2. Reactive Synthesis

- ▶ The environment is in **red**, as it is in charge of inputs.
- ▶ The system is in **green**, as it is in charge of outputs.
- ▶ Implementations as well, as they correspond to systems.
- ▶ Specifications are in **blue**, since they are relations.

A.3. ω -Automata

- ▶ States are depicted as circles, with their name inside, e.g. a state p is displayed as: 
- ▶ Transitions are depicted as arrows, with their label above. For instance, $p \xrightarrow{a} q$ is depicted as follows: 
- ▶ Initial states are marked with an incoming arrow: 
- ▶ States which belong to the set F of accepting (for the Büchi acceptance condition) or rejecting states (for the co-Büchi one) are doubly circled: 
- ▶ When the label of a transition can be any input, it is abbreviated as $*$.

The name can be omitted when irrelevant.

Labels can be positioned otherwise for graphical purposes.

A.3.1. Specification Automata

Consistently with the above conventions:

- ▶ Input states and input transitions are in **red**
- ▶ Output states and output transitions are in **green**
- ▶ In particular, a red star (*) denotes any input letter, and a green star (*), any output letter.

A.4. Register Automata

Conventions regarding states and transitions are inherited from ω -automata. Additionally:

- ▶ Labels are omitted when they are irrelevant, which is the case when the alphabet is unary
- ▶ Recall that $\top$ is a test which is satisfied by any data value
- ▶ $\text{asgn} = \{r\}$ is denoted $\downarrow r$, and more generally $\text{asgn} = \{r_1, \dots, r_n\}$ is denoted $\downarrow r_1, \dots, \downarrow r_n$.
- ▶ $\text{asgn} = \emptyset$ is omitted
- ▶ When reassignments are involved, we write $A := s$ when the content of s is copied into each register in the set $A \subseteq R$, which corresponds to the reassignment function $r \in A \mapsto s$ and $r \notin A \mapsto r$
- ▶ When non-deterministic reassignment is involved, we write $?r$ to mean that $r \in \text{ndasgn}$

A.5. Synchronous Sequential Register Transducers

Conventions regarding states, transitions and register operations on the input are inherited from register automata. Additionally, on transitions we write $\uparrow r$ when the output register is r .

A.6. Non-Deterministic Asynchronous Register Transducers

Conventions regarding states and transitions are inherited from ω -automata. Since the definition of the model slightly differ from the perspective adopted in Part I, we gain spell out its associated graphical conventions:

- ▶ When the data domain supports labels from a finite alphabet Σ , we write σ, ϕ for the test $\sigma(\star) \wedge \phi$
- ▶ Given a transition $p \xrightarrow{\phi | \text{regOp}, w} q$, we write $\downarrow r$ when $\text{regOp}(r) = \text{set}$, $?r$ when $\text{regOp}(r) = \text{guess}$ and nothing when $\text{regOp}(r) = \text{set}$.
- ▶ Finally, we write $\uparrow r_1 \cdot \dots \cdot \uparrow r_k$ when the output word is $r_1 \dots r_k$.

This is done to unify the notations with Part I. For $\sigma \in \Sigma$, the test $\sigma(\star) \wedge \phi$ asks that the label is σ and conducts test ϕ .

Details of Chapter 3

B.

B.1. Section 3.4.1 (Logics for Specifications)

B.1.1. The ω -language L_{par} is definable in MSO, but not in LTL

First, we give the details of Example 3.8, and more precisely show that $L_{\text{par}} = \{w_0b_0\$w_1b_1\$ \dots \in (\{0,1\}^*\$)^\omega \mid \text{for all } i \in \mathbb{N}, b_i = |w_i|_1 \bmod 2\}$ is definable by a MSO formula.

Example B.1 First, define a formula $\varphi_{\$}(Y)$ which holds whenever the set Y is a set of positions strictly between two \$, last bit excluded:

$$\begin{aligned} \varphi_{\$}(Y) := & \exists b. b \notin Y \wedge \$(b) \wedge b + 1 \in Y \wedge \exists e. e \notin Y \wedge \$(e + 1) \wedge e - 1 \in Y \\ & \wedge (\forall y. b < y < e \Leftrightarrow (y \in Y \wedge \neg \$(y))) \end{aligned}$$

b is thus the \$ at the *beginning* of Y , while e is the last bit of the sequence (thus its successor is a \$). Then, $\varphi_{\$}$ can be refined into φ_1 so as to characterise sets of positions between two \$ which are labelled 1:

$$\varphi_1(X) := \exists Y. \varphi_{\$}(Y) \wedge X \subseteq Y \wedge \forall y \in Y. 1(y) \Leftrightarrow y \in X$$

where $X \subseteq Y$ is a short for $\forall x. x \in X \Rightarrow x \in Y$.

Now, we have to express that X contains an even number of elements. To that end, we first introduce the formula $\varphi_{\sqcup}(X, Y, Z)$ which expresses that Y and Z partition X , i.e. $X = Y \sqcup Z$.

$$\varphi_{\sqcup}(X, Y, Z) := \forall t. (t \in Y \vee t \in Z) \Rightarrow t \in X \wedge \forall t \in X. (t \in Y \Leftrightarrow t \notin Z)$$

Then, X contains an even number of elements whenever it can be partitioned into two sets Y and Z which respectively contain the first and last element of X , and which alternate: if $x \in Y$, then the next element of X will belong to Z , and conversely.

$$\begin{aligned} \varphi_{\text{even}}(X) := & \exists Y. \exists Z. \varphi_{\sqcup}(X, Y, Z) \\ & \wedge \forall y, z \in X. \neg(\exists x \in X. y < x < z) \Rightarrow (y \in Y \Leftrightarrow z \in Z) \\ & \wedge \forall y \in X. (\forall t. t < y \Rightarrow t \notin X) \Rightarrow y \in Y \\ & \wedge \forall z \in X. (\forall t. t > z \Rightarrow t \notin X) \Rightarrow z \in Z \end{aligned}$$

Finally, one can state that a position is the last of a sequence by asking that the next position is labelled \$. Putting it together, one obtains the sought formula:

$$\varphi_{\text{par}} := \forall X. \varphi_1(X) \Rightarrow \forall y. (y \notin X \wedge y - 1 \in X) \Rightarrow (0(y) \Leftrightarrow \varphi_{\text{even}}(X))$$

Then, we show the claim of Example 3.12, namely that L_{par} is not definable in LTL. To that end, we use the fact that an ω -language is definable in LTL whenever it is definable in $\text{FO}[<]$, which is the case if and only

if it is aperiodic [62, Theorem 1.1]. An ω -language $L \subseteq \Sigma^\omega$ is *aperiodic* when there exists some $n \in \mathbb{N}$ such that for all $u, x \in \Sigma^*$ and all $v \in \Sigma^\omega$, $u \cdot x^n \cdot v \in L \Leftrightarrow u \cdot x^{n+1} \cdot v \in L$. Assume by contradiction that L_{par} is aperiodic, and let n be as in the definition. We treat the case where n is odd, the other case is similar. Let $u = \varepsilon$, $x = 1$ and $v = 0 \cdot \$ \cdot (11\$)^\omega$. We have that $w = u \cdot x^n \cdot v = 1^n \cdot 0 \cdot \$ \cdot (11\$)^\omega \in L$, since the number of 1 in the first block is even, and it is odd in the others, and the b_i correspond for each block. However, $u \cdot x^{n+1} \cdot v \notin L$, since b_0 is not correct anymore: there is an odd number of 1 in the first block, while $b_0 = 0$. This means that L_{par} is not aperiodic, so it is not definable in LTL.

[62]: Diekert and Gastin (2008), ‘First-order definable languages’

B.2. Section 3.4.3 (Automata)

B.2.1. Representation of Automata

When they are represented explicitly, automata are not extremely succinct. For instance, to recognise finite words of length at most i , an automaton needs i states, a number that is exponential in the size of the representation of i (namely, $\log i$). In our study, we sometimes need to represent automata in a concise way, to obtain PSPACE membership. In the following, we examine what is a representation of an automaton, and what conditions it must obey so that one can efficiently check whether it is empty (most of the decision problems we study reduce to checking emptiness). One should also be able to effectively compute union and intersection while not blowing up the representation. Note that the representation of finite-state automata and Büchi automata is the same, as they are syntactically equivalent. We describe the case of Büchi automata, the case of finite-state automata is simpler and can be deduced.

Definition B.1 Let $A = (Q, \Sigma, I, F, \Delta)$ be a non-deterministic Büchi automaton. A *representation* of A consist in an encoding of states, i.e. an injective function $\text{enc}_Q : Q \rightarrow \{0, 1\}^s$ and an encoding of the input alphabet $\text{enc}_\Sigma : \Sigma \times \{0, 1\}^a$ along with algorithms $(\text{alg}_Q, \text{alg}_\Sigma, \text{alg}_I, \text{alg}_F, \text{alg}_\Delta)$ and *maximum encoding lengths* $s, a \in \mathbb{N} \setminus \{0\}$ that satisfy the following conditions:

- The algorithm alg_Q takes as input a string $w \in \{0, 1\}^*$ and outputs 1 if and only if $w = \text{enc}_Q(q)$ for some $q \in Q$; similarly for alg_I and alg_F .
- The algorithm alg_Σ takes as input a string $w \in \{0, 1\}^*$ and outputs 1 if and only if $w = \text{enc}_\Sigma(\sigma)$ for some $\sigma \in \Sigma$.
- The algorithm alg_Δ takes as input a triple (w_p, w_q, w_σ) where $w_p, w_q, w_\sigma \in \{0, 1\}^*$, $w_p = \text{enc}(p)$, $w_q = \text{enc}(q)$ for some $p, q \in Q$ and $w_\sigma = \text{enc}(\sigma)$ for some $\sigma \in \Sigma$. It outputs 1 if and only if there exists a transition $p \xrightarrow[A]{\sigma} q$ in A (i.e., whether $(p, a, q) \in \Delta$).

It is moreover *succinct* if $s = P(\log Q)$ and $a = P(\log \Sigma)$ for some polynomial P , and, additionally, all algorithms run using a space that is polynomial in the size of their input.

B.2.2. The emptiness problem

By a simple pumping argument, we get:

Lemma B.1 *Let A be a non-deterministic Büchi automaton over alphabet Σ . A recognises at least a word (i.e. $L(A) \neq \emptyset$) if and only if there exists $u, v \in \Sigma^*$, an initial state $p \in I$ and a final state $q_f \in F$ such that $p \xrightarrow[A]{u} q_f \xrightarrow[A]{v} q_f$.*

As a consequence, we get:

Theorem 3.4 *The emptiness problem for non-deterministic Büchi automata is in $N\text{LOGSPACE}$.*

Proof. We give some details for completeness. We actually show that the non-emptiness problem is in $N\text{LOGSPACE}$. By Immerman–Szelepcsényi’s theorem [150, 151, Theorem 1, Theorem 1], it implies that the emptiness problem is in $N\text{LOGSPACE}$, since it states that $N\text{LOGSPACE} = \text{co}N\text{LOGSPACE}$.

The reader who is familiar with the pattern logic of [149, Definition 6] can notice that the characterisation of can be expressed and checked with it. Indeed, it amounts to model-checking the input automaton A with regard to $\exists \pi_1 : p \xrightarrow{u} q, \exists \pi_2 : q \xrightarrow{v} q \cdot \text{init}(p) \wedge \text{final}(q)$. By [149, Theorem 7], we get $N\text{LOGSPACE}$ membership.

We now detail an ad-hoc non-deterministic algorithm for the reader who does not know the logic, or who would like to know what happens under the hood. **guess** $v \in X$ means that the algorithm non-deterministically assigns the variable with a value of X . **halt** means that the computation fails. **accept** means that the computation succeeds and the input is accepted. Finally, **increment** c means $c := c + 1$.

Algorithm 3: A non-deterministic algorithm to decide the non-emptiness problem for non-deterministic Büchi automata.

Input: A succinct representation of a non-deterministic Büchi automaton with encoding lengths s and a

```

1  $p \leftarrow \varepsilon$                                 /* The current state */
2  $q \leftarrow \varepsilon$                         /* The next state */
3  $q_f \leftarrow \varepsilon$                      /* The repeating final state */
4  $\sigma \leftarrow \varepsilon$                    /* The label of the transition to take */
5  $\text{occ}_{q_f} := 0$                            /* Count occurrences of  $q_f$  */
6 guess  $p \in \{0, 1\}^s$                      /* Guess the initial state  $p$  */
7 if  $\neg \text{alg}_I(p)$  then halt
8 guess  $q_f \in \{0, 1\}^s$                    /* Guess the repeating final state  $q_f$  */
9 if  $\neg \text{alg}_F(q_f)$  then halt
10 while true do                             /* Explore the automaton */
11   if  $p = q_f$  then increment  $\text{occ}_{q_f}$  /* Track whether  $q_f$  occurs */
12   if  $\text{occ}_{q_f} \geq 2$  then accept             /*  $q_f$  repeats */
13   guess  $q \in \{0, 1\}^s$                    /* Guess the next transition to take */
14   guess  $\sigma \in \{0, 1\}^a$ 
15   if  $\text{alg}_\Delta(p, q, \Sigma)$  then  $p \leftarrow q$  /* Take the transition */
16   else halt

```

By the characterisation of Lemma B.2.2, there exists an accepting run of the above algorithm if and only if its input automaton is not empty. By definition of a succinct representation, it runs in space logarithmic in $|Q|$

[150]: Immerman (1988), ‘Nondeterministic Space is Closed Under Complementa-tion’

[151]: Szelepcsényi (1988), ‘The Method of Forced Enumeration for Nondeterministic Automata’

[149]: Filiot, Mazzocchi, and Raskin (2020), ‘A Pattern Logic for Automata with Out-puts’

and $|\Sigma|$ (all variables take logarithmic space, except for occ_{q_f} which takes constant space $\log 2$). This establishes that the non-emptiness problem for Büchi automata is in NLOGSPACE , hence the emptiness problem is as well. $\square$

Corollary B.2 *Assume some ambient object O of size k , typically a register automaton. As a consequence of the above, we get that if an automaton (related in one way or the other to O) can be represented using a space that is polynomial (not logarithmic) in k , one can decide whether it is empty in polynomial space. This is generally how we get PSPACE membership of the problems we study in Part II.*

B.2.3. Bounded Counters

It is often useful to equip automata with *bounded counters*, that allow them to count up to a certain bound. Formally, an automaton A can be equipped by a B -bounded counter by enriching its state space Q with $Q \times \{0, \dots, B\}$. It then has the ability to test whether a given value is equal to i for any $0 \leq i \leq B$. An important property is that such an automaton can be described using a space that is polynomial in $|A|$ and logarithmic in B . Indeed, one can encode a state (q, i) for some $i \in \{0, \dots, B\}$ as $\text{enc}_Q(q)\$2^{\bar{i}}$, where $2^{\bar{i}}$ is the binary encoding of i .

Proposition B.3 (Bounded counter) *Let A be some automaton. Equipping it with a constant number of counters that are bounded by some B that is exponential in the size of $|A|$ can be done in polynomial space.*

The separator $\$$ can in turn be encoded if one wants an encoding over $\{0, 1\}$.

Details of Part II

C.

C.0.1. Section 12.3.1 (Register Automata over a Discrete Domain)

Lemma C.1 Let $\mu \in \text{Aut}(\mathbb{Q}_+)$ be a morphism that preserves $\mathbb{N}$, i.e. $\mu(\mathbb{N}) \subseteq \mathbb{N}$.

For all $m, n \in \mathbb{N}$, $|\mu(n) - \mu(m)| \geq |n - m|$.

Proof. Up to terminology, this is an high school exercise. Let $\zeta \in \text{Aut}(\mathbb{Q}_+)$ be some $\mathbb{N}$ -preserving morphism. The result essentially follows from the fact that the image of any integer interval of size 1 is of size at least 1, since μ is increasing and bijective.

Let $m \in \mathbb{N}$. We show by induction that for all $n \geq m$, $\mu(n) - \mu(m) \geq n - m$. For $m = n$, the result is immediate. Assume that the results hold for some $n \geq m$. We have that $\mu(n+1) - \mu(m) = \mu(n+1) - \mu(n) + \mu(n) - \mu(m)$. Since μ is strictly increasing, we have that $\mu(n+1) > \mu(n)$, so $\mu(n+1) - \mu(n) \geq 1$. Thus, $\mu(n+1) - \mu(m) \geq 1 + \mu(n) - \mu(m) \geq 1 + n - m$ by the induction hypothesis, so the inductive invariant holds at step $n + 1$.

The result is then obtained by interverting m and n when $n < m$. $\square$

Observation 12.53 Let $\nu, \lambda : R \rightarrow \mathbb{Q}_+$. If (ν, λ) is widening, then so is $(K\nu, K\lambda)$, for all $K \in \mathbb{N} \setminus \{0\}$. Moreover, $\text{type}(K\nu, K\lambda) = \text{type}(\nu, \lambda)$.

Proof. Let $R = \{r_1, \dots, r_k\}$, and let (ν, λ) be a pair of $\mathbb{Q}_+$ -valuations over R that is widening, and let $K \in \mathbb{N} \setminus \{0\}$. Since the type is preserved when applying a morphism, we have that $\text{type}(K\nu, K\lambda) = \text{type}(\nu, \lambda)$. Observe that (ν, λ) is again widening. First, for all $r \in R$, $K\lambda(r) \geq K\nu(r)$ since $\lambda(r) \geq \nu(r)$. Then, for all $r, s \in R$, we have:

$$\begin{aligned} |\lambda(s) - \lambda(r)| &= |K\lambda'(s) - K\lambda'(r)| \\ &= K|\lambda'(s) - \lambda'(r)| \\ &\geq K|\nu'(s) - \nu'(r)| \quad \text{since } (\nu', \lambda') \text{ is widening} \\ &\geq |K\nu'(s) - K\nu'(r)| \\ &\geq |\nu(s) - \nu(r)| \end{aligned}$$

$\square$

Proposition 12.65 Let $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$ be a function defined by a non-deterministic register transducer over $(\mathbb{N}, <, 0)$. Its next-letter function Next_f is computable.

Proof. Let $f : \mathbb{D}^\omega \rightarrow \mathbb{D}^\omega$ be a function defined by a non-deterministic register transducer T over $(\mathbb{N}, <, 0)$. Given two input data words $u, v \in \mathbb{D}^*$, the next-letter problem asks to output $d \in \mathbb{D}$ such that $f(u \cdot \mathbb{N}^\omega) \subseteq v \cdot d \cdot \mathbb{N}^\omega$ if it exists, and answer No otherwise.

Solving the next-letter decision problem We first solve its decision version, or, more precisely, its complement. There is no next letter if and only if either v was already not a prefix of all $f(u \cdot x)$, i.e. $f(u \cdot \mathbb{N}^\omega) \not\subseteq v \cdot \mathbb{N}^\omega$, or if there exists $x_1, x_2 \in \mathbb{N}^\omega$ such that $f(u \cdot x_1)$ and $f(u \cdot x_2)$ mismatch right after v , i.e. $|f(u \cdot x_1) \wedge f(u \cdot x_2)| \leq |v|$.

We start by checking that $f(u \cdot \mathbb{N}^\omega) \not\subseteq v \cdot \mathbb{N}^\omega$, which is the case whenever $f(u \cdot \mathbb{N}^\omega) \cap (v \cdot \mathbb{N}^\omega)^c \neq \emptyset$. This reduces to checking emptiness of some register automaton. First, $f(u \cdot \mathbb{N}^\omega)$ is recognised by some register automaton over $\mathbb{N} \cup \text{elem}(u)$, where the elements of u are constants of the domain. Indeed, the automaton simulates T while checking that u is a prefix of the input. Similarly, $v \cdot \mathbb{N}^\omega$ is recognised by a *deterministic* register automaton over $\mathbb{N} \cup \text{elem}(v)$, which can be complemented. Their intersection is a register automaton over $\mathbb{N} \cup \text{elem}(u) \cup \text{elem}(v)$. By Corollary 12.75, it is decidable whether such an automaton is empty.

We now check whether there exists $x_1, x_2 \in \mathbb{N}^\omega$ such that $f(u \cdot x_1)$ and $f(u \cdot x_2)$ mismatch right after v . This is equivalent to asking the existence of two runs $C^l \xrightarrow{u|u_1} C_1 \xrightarrow{x_1|y_1} D_1$ and $C^l \xrightarrow{u|u_2} C_2 \xrightarrow{x_2|y_2} D_1$ such that $|u_1 \cdot x_1 \wedge u_2 \cdot x_2| \leq |v|$. Finding these runs can be done with a register automaton over $\mathbb{N} \cup \text{elem}(u)$. Such an automaton has a $|v|$ -bounded counter. It guesses two runs and checks that their input has u as prefix. In parallel, it guesses a mismatch position j that it stores in its counter, and checks that the outputs indeed mismatch at position j .

Both cases are decidable, so, overall, the next-letter decision problem is decidable.

Finding the next letter If there does not exist a next letter, we are done and the algorithm simply outputs No. Otherwise, we know that such a letter exists, so it remains to exhibit it. To that end, it suffices to enumerate all natural numbers $n \in \mathbb{N}$ and iteratively check whether $f(u \cdot \mathbb{N}^\omega) \subseteq v \cdot n \cdot \mathbb{N}^\omega$, which again reduces to the emptiness problem of some register automaton. $\square$

Bibliography

- [1] Alan M. Turing. 'On Computable Numbers, with an Application to the Entscheidungsproblem'. In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265 (cited on pages 1, 24, 121, 223).
- [2] Rajeev Alur et al. 'Syntax-guided synthesis'. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 1–8 (cited on pages 2, 24, 138).
- [3] Bernd Finkbeiner and Sven Schewe. 'Bounded synthesis'. In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 519–539. doi: 10.1007/s10009-012-0228-z (cited on pages 2, 8, 24, 30, 137).
- [4] Alonzo Church. 'Applications of recursive arithmetic to the problem of circuit synthesis'. In: *Summaries of the Summer Institute of Symbolic Logic Volume I* (1957), pp. 3–50 (cited on pages 6, 28, 29, 52, 53, 63).
- [5] Wolfgang Thomas. 'Facets of Synthesis: Revisiting Church's Problem'. In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Luca de Alfaro. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 1–14. doi: 10.1007/978-3-642-00596-1_1 (cited on pages 6, 7, 28, 29, 45, 48, 51, 53, 55, 222).
- [6] Robert McNaughton. *Finite-state infinite games*. Tech. rep. Project MAC Rep., MIT, Cambridge, Mass., 1965 (cited on pages 7, 29).
- [7] Robert McNaughton. 'Infinite Games Played on Finite Graphs'. In: *Ann. Pure Appl. Logic* 65.2 (1993), pp. 149–184. doi: 10.1016/0168-0072(93)90036-D (cited on pages 7, 29).
- [8] J.R. Büchi and L.H. Landweber. 'Solving sequential conditions finite-state strategies'. In: *Transactions of the American Mathematical Society* 138 (1969), pp. 295–311 (cited on pages 7, 29, 51, 55, 69, 70).
- [9] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. 'Graph Games and Reactive Synthesis'. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 921–962. doi: 10.1007/978-3-319-10575-8_27 (cited on pages 7, 8, 30, 70).
- [10] M.O. Rabin. *Automata on Infinite Objects and Church's Problem*. Conference Board of the mathematical science. Conference Board of the Mathematical Sciences, 1972 (cited on pages 7, 29).
- [11] Albert R. Meyer. 'Weak monadic second order theory of successor is not elementary-recursive'. In: *Logic Colloquium*. Ed. by Rohit Parikh. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 132–154 (cited on pages 7, 29, 55, 62).
- [12] Amir Pnueli. 'The Temporal Logic of Programs'. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. doi: 10.1109/SFCS.1977.32 (cited on pages 8, 29, 50, 55).
- [13] A. Pnueli and R. Rosner. 'On the Synthesis of a Reactive Module'. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 179–190. doi: 10.1145/75277.75293 (cited on pages 8, 29, 56, 69, 70, 195).
- [14] Bernd Finkbeiner. 'Synthesis of Reactive Systems'. In: *Dependable Software Systems Engineering*. Ed. by Javier Esparza, Orna Grumberg, and Salomon Sickert. Vol. 45. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2016, pp. 72–98. doi: 10.3233/978-1-61499-627-9-72 (cited on pages 8, 29).
- [15] Rüdiger Ehlers. 'Symbolic bounded synthesis'. In: *Formal Methods Syst. Des.* 40.2 (2012), pp. 232–262. doi: 10.1007/s10703-011-0137-x (cited on pages 8, 30).
- [16] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 'Antichains and compositional algorithms for LTL synthesis'. In: *Formal Methods Syst. Des.* 39.3 (2011), pp. 261–296. doi: 10.1007/s10703-011-0115-3 (cited on pages 8, 30, 201).

- [17] Swen Jacobs et al. ‘The first reactive synthesis competition (SYNTCOMP 2014)’. In: *Int. J. Softw. Tools Technol. Transf.* 19.3 (2017), pp. 367–390. doi: 10.1007/s10009-016-0416-3 (cited on pages 8, 30).
- [18] Borzoo Bonakdarpour and Bernd Finkbeiner. ‘Controller Synthesis for Hyperproperties’. In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 2020, pp. 366–379. doi: 10.1109/CSF49147.2020.00033 (cited on pages 8, 30).
- [19] Bernd Finkbeiner, Gideon Geier, and Noemi Passing. ‘Specification Decomposition for Reactive Synthesis’. In: *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*. Ed. by Aaron Dutle et al. Vol. 12673. Lecture Notes in Computer Science. Springer, 2021, pp. 113–130. doi: 10.1007/978-3-030-76384-8_8 (cited on pages 8, 30, 201).
- [20] Shaull Almagor and Orna Kupferman. ‘Good-Enough Synthesis’. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 541–563 (cited on pages 8, 30).
- [21] Michael Kaminski and Nissim Francez. ‘Finite-Memory Automata’. In: *Theor. Comput. Sci.* 134.2 (1994), pp. 329–363. doi: 10.1016/0304-3975(94)90242-9 (cited on pages 10, 31, 32, 71, 73, 76, 81, 83–89, 92, 95–97, 123, 161).
- [22] Luc Segoufin. ‘Automata and Logics for Words and Trees over an Infinite Alphabet’. In: *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*. Ed. by Zoltán Ésik. Vol. 4207. Lecture Notes in Computer Science. Springer, 2006, pp. 41–57. doi: 10.1007/11874683_3 (cited on pages 10, 31, 71, 86, 99).
- [23] Michael Benedikt, Clemens Ley, and Gabriele Puppis. ‘What You Must Remember When Processing Data Words’. In: *Proceedings of the 4th Alberto Mendelzon International Workshop on Foundations of Data Management, Buenos Aires, Argentina, May 17-20, 2010*. Ed. by Alberto H. F. Laender and Laks V. S. Lakshmanan. Vol. 619. CEUR Workshop Proceedings. CEUR-WS.org, 2010 (cited on pages 10, 31, 86, 311, 312).
- [24] Diego Figueira, Piotr Hofman, and Sławomir Lasota. ‘Relating timed and register automata’. In: *Math. Struct. Comput. Sci.* 26.6 (2016), pp. 993–1021. doi: 10.1017/S0960129514000322 (cited on pages 10, 31).
- [25] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. ‘Automata theory in nominal sets’. In: *Log. Methods Comput. Sci.* 10.3 (2014). doi: 10.2168/LMCS-10(3:4)2014 (cited on pages 10, 31, 73, 126, 241–243, 306, 312).
- [26] Mikołaj Bojańczyk. *Atom Book*. 2019 (cited on pages 10, 19, 20, 31, 39, 40, 82, 83, 131, 241, 243, 245, 246, 248, 252, 306).
- [27] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. ‘Alternation’. In: *J. ACM* 28.1 (1981), pp. 114–133. doi: 10.1145/322234.322243 (cited on pages 10, 32, 59, 68, 174).
- [28] Stéphane Demri and Ranko Lazic. ‘LTL with the freeze quantifier and register automata’. In: *ACM Trans. Comput. Log.* 10.3 (2009), 16:1–16:30. doi: 10.1145/1507244.1507246 (cited on pages 10, 12, 32, 33, 81, 92, 97, 102, 173, 174, 233, 239, 273, 282, 294, 310).
- [29] Diego Figueira. ‘Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données)’. PhD thesis. École normale supérieure de Cachan, France, 2010 (cited on pages 10, 32, 81).
- [30] Wolfgang Thomas. ‘Church’s Problem and a Tour through Automata Theory’. In: *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Ed. by Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 635–655. doi: 10.1007/978-3-540-78127-1_35 (cited on pages 12, 70).
- [31] Tim French. ‘Quantified Propositional Temporal Logic with Repeating States’. In: *10th International Symposium on Temporal Representation and Reasoning / 4th International Conference on Temporal Logic (TIME-ICTL 2003), 8-10 July 2003, Cairns, Queensland, Australia*. IEEE Computer Society, 2003, pp. 155–165. doi: 10.1109/TIME.2003.1214891 (cited on pages 12, 33).
- [32] Stéphane Demri, Deepak D’Souza, and Régis Gascon. ‘Temporal Logics of Repeating Values’. In: *J. Log. Comput.* 22.5 (2012), pp. 1059–1096. doi: 10.1093/logcom/exr013 (cited on pages 12, 33).

- [33] Diego Figueira, Anirban Majumdar, and M. Praveen. ‘Playing with Repetitions in Data Words Using Energy Games’. In: *Log. Methods Comput. Sci.* 16.3 (2020) (cited on pages 12, 15, 17, 33, 35, 38, 159, 166, 310).
- [34] Béatrice Bérard et al. ‘Parameterized Synthesis for Fragments of First-Order Logic Over Data Words’. In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Lecture Notes in Computer Science. Springer, 2020, pp. 97–118. doi: 10.1007/978-3-030-45231-5_6 (cited on pages 13, 34, 195, 200).
- [35] Ayrat Khalimov and Orna Kupferman. ‘Register-Bounded Synthesis’. In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Ed. by Wan J. Fokkink and Rob van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 25:1–25:16. doi: 10.4230/LIPIcs.CONCUR.2019.25 (cited on pages 13, 16, 34, 36, 137, 150).
- [36] Ayrat Khalimov, Benedikt Maderbacher, and Roderick Bloem. ‘Bounded Synthesis of Register Transducers’. In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 494–510. doi: 10.1007/978-3-030-01090-4_29 (cited on pages 14–16, 34–36, 93, 137, 138, 147, 150, 160).
- [37] Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Ed. by Wan Fokkink and Rob van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 24:1–24:15. doi: 10.4230/LIPIcs.CONCUR.2019.24 (cited on pages 14, 15, 34–36, 138, 159, 160).
- [38] Michael Kaminski and Daniel Zeitlin. ‘Finite-Memory Automata with Non-Deterministic Reassignment’. In: *Int. J. Found. Comput. Sci.* 21.5 (2010), pp. 741–760. doi: 10.1142/S0129054110007532 (cited on pages 14, 34, 37, 82, 98, 122, 123, 125, 126, 151, 155, 203, 211, 227, 231).
- [39] Luc Segoufin and Szymon Torunczyk. ‘Automata based verification over linearly ordered data domains’. In: *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*. Ed. by Thomas Schwentick and Christoph Dürr. Vol. 9. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011, pp. 81–92. doi: 10.4230/LIPIcs.STACS.2011.81 (cited on pages 14, 34, 84, 107, 110, 118, 251, 273, 282, 291).
- [40] Mikolaj Bojanczyk. ‘Transducers with Origin Information’. In: *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*. Ed. by Javier Esparza et al. Vol. 8573. Lecture Notes in Computer Science. Springer, 2014, pp. 26–37. doi: 10.1007/978-3-662-43951-7_3 (cited on pages 14, 35).
- [41] Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘Synthesis of Data Word Transducers’. In: *Logical Methods in Computer Science* 17.1 (2021) (cited on pages 14, 15, 35, 36, 138, 144, 150, 152, 153, 157, 159, 160, 175, 195, 197, 198).
- [42] Léo Exibard, Emmanuel Filiot, and Ayrat Khalimov. ‘Church Synthesis on Register Automata over Linearly Ordered Data Domains’. In: *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany*. Ed. by Benjamin Monmege and Markus Bläser. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021 (cited on pages 16, 36, 159, 178, 184, 193).
- [43] Rachel Faran and Orna Kupferman. ‘On Synthesis of Specifications with Arithmetic’. In: *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*. Ed. by Alexander Chatzigeorgiou et al. Vol. 12011. Lecture Notes in Computer Science. Springer, 2020, pp. 161–173. doi: 10.1007/978-3-030-38919-2_14 (cited on pages 16, 37).

- [44] Rüdiger Ehlers, Sanjit A. Seshia, and Hadas Kress-Gazit. ‘Synthesis with Identifiers’. In: *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*. Ed. by Kenneth L. McMillan and Xavier Rival. Vol. 8318. Lecture Notes in Computer Science. Springer, 2014, pp. 415–433. doi: 10.1007/978-3-642-54013-4_23 (cited on pages 17, 37, 119, 200, 310).
- [45] Antoine Durand-Gasselín and Peter Habermehl. ‘Regular Transformations of Data Words Through Origin Information’. In: *Proceedings of the 19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2016)*. Vol. 9634. Lecture Notes in Computer Science. Springer, 2016, pp. 285–300. doi: 10.1007/978-3-662-49630-5_17 (cited on pages 17, 37, 155).
- [46] Bernd Finkbeiner et al. ‘Temporal Stream Logic: Synthesis Beyond the Booleans’. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 609–629. doi: 10.1007/978-3-030-25540-4_35 (cited on pages 17, 37, 200).
- [47] Paul Krogmeier et al. ‘Decidable Synthesis of Programs with Uninterpreted Functions’. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 634–657. doi: 10.1007/978-3-030-53291-8_32 (cited on pages 17, 37).
- [48] Arnaud Carayol and Christof Löding. ‘Uniformization in Automata Theory’. In: *Logic, Methodology and Philosophy of Science - Proceedings of the 14th International Congress*. 2015 (cited on pages 18, 19, 38, 39, 62, 203, 307).
- [49] Michael Holtmann, Lukasz Kaiser, and Wolfgang Thomas. ‘Degrees of Lookahead in Regular Infinite Games’. In: *Logical Methods in Computer Science* 8.3 (2012). doi: 10.2168/LMCS-8(3:24)2012 (cited on pages 19, 39, 307).
- [50] Christophe Prieur. ‘How to decide continuity of rational functions on infinite words’. In: *Theor. Comput. Sci.* 276.1-2 (2002), pp. 445–447. doi: 10.1016/S0304-3975(01)00307-3 (cited on pages 21, 40, 203).
- [51] Vrunda Dave et al. ‘Synthesis of Computable Regular Functions of Infinite Words’. In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 43:1–43:17. doi: 10.4230/LIPIcs.CONCUR.2020.43 (cited on pages 21, 41, 203, 216, 219, 226, 229, 235, 239, 306).
- [53] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer, 2002. doi: 10.1007/3-540-36387-4 (cited on pages 48, 53, 54).
- [54] J. Richard Büchi. ‘On a decision method in restricted second order arithmetic’. In: *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*. Stanford University Press, 1962, pp. 1–11 (cited on pages 50, 55, 57, 61, 62, 70).
- [55] Patricia Bouyer et al. ‘Timed Temporal Logics’. In: *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*. Ed. by Luca Aceto et al. Vol. 10460. Lecture Notes in Computer Science. Springer, 2017, pp. 211–230. doi: 10.1007/978-3-319-63121-9_11 (cited on page 50).
- [56] Larry J. Stockmeyer. ‘The Complexity of Decision Problems in Automata Theory and Logic’. PhD thesis. Massachusetts Institute of Technology, 1974 (cited on page 53).
- [57] Mark Weyer. ‘Decidability of S1S and S2S’. In: *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Ed. by Erich Grädel, Wolfgang Thomas, and Thomas Wilke. Vol. 2500. Lecture Notes in Computer Science. Springer, 2001, pp. 207–230. doi: 10.1007/3-540-36387-4_12 (cited on pages 54, 62).
- [58] H. Comon et al. *Tree Automata Techniques and Applications*. release October, 12th 2007. 2007 (cited on page 54).

- [59] Mikołaj Bojańczyk and Thomas Colcombet. 'Bounds in ω -Regularity'. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, pp. 285–296. doi: 10.1109/LICS.2006.17 (cited on pages 54, 84, 108, 110, 299).
- [60] J. A. W. Kamp. 'Tense Logic and the Theory of Linear Order'. PhD thesis. University of California, Los Angeles (California), 1968 (cited on page 55).
- [61] Dov M. Gabbay et al. 'On the Temporal Basis of Fairness'. In: *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*. Ed. by Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne. ACM Press, 1980, pp. 163–173. doi: 10.1145/567446.567462 (cited on page 55).
- [62] Volker Diekert and Paul Gastin. 'First-order definable languages'. In: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*. Ed. by Jörg Flum, Erich Grädel, and Thomas Wilke. Vol. 2. Texts in Logic and Games. Amsterdam University Press, 2008, pp. 261–306 (cited on pages 55, 56, 317).
- [63] Nir Piterman and Amir Pnueli. 'Temporal Logic and Fair Discrete Systems'. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 27–73. doi: 10.1007/978-3-319-10575-8_2 (cited on page 55).
- [64] David E. Muller. 'Infinite sequences and finite machines'. In: *4th Annual Symposium on Switching Circuit Theory and Logical Design, Chicago, Illinois, USA, October 28-30, 1963*. IEEE Computer Society, 1963, pp. 3–16. doi: 10.1109/SWCT.1963.8 (cited on page 57).
- [65] Robert McNaughton. 'Testing and generating infinite sequences by a finite automaton'. In: *Information and Control* 9.5 (1966), pp. 521–530. doi: [https://doi.org/10.1016/S0019-9958\(66\)80013-X](https://doi.org/10.1016/S0019-9958(66)80013-X) (cited on pages 57, 60, 61).
- [66] Satoru Miyano and Takeshi Hayashi. 'Alternating Finite Automata on omega-Words'. In: *Theor. Comput. Sci.* 32 (1984), pp. 321–330. doi: 10.1016/0304-3975(84)90049-5 (cited on page 59).
- [67] Shmuel Safra. 'On the Complexity of ω -Automata'. In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 319–327. doi: 10.1109/SFCS.1988.21948 (cited on page 60).
- [68] Lawrence H. Landweber. 'Decision Problems for omega-Automata'. In: *Math. Syst. Theory* 3.4 (1969), pp. 376–384. doi: 10.1007/BF01691063 (cited on page 60).
- [69] Orna Kupferman and Salomon Sickert. 'Certifying Inexpressibility'. In: *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by Stefan Kiefer and Christine Tasson. Vol. 12650. Lecture Notes in Computer Science. Springer, 2021, pp. 385–405. doi: 10.1007/978-3-030-71995-1_20 (cited on page 60).
- [70] U. Boker. *Word-Automata Translations*. 2020. URL: <http://www.faculty.idc.ac.il/udiboker/automata> (cited on pages 60, 61).
- [71] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009 (cited on pages 61, 205, 212).
- [72] J. Richard Büchi. 'Weak Second-Order Arithmetic and Finite Automata'. In: *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 6 (1960), pp. 66–92 (cited on page 61).
- [73] C.C. Elgot. 'Decision problems of finite automata design and related arithmetics'. In: *Transactions of the American Mathematical Society*. Vol. 98. 1. 1961, pp. 21–51 (cited on page 61).
- [74] B.A. Trakhtenbrot. 'Finite automata and logic of monadic predicates'. In: *Doklady Akademii Nauk SSSR*. Vol. 149. In Russian. 1961, pp. 326–329 (cited on page 61).
- [75] Bakhadyr Khoussainov and Anil Nerode. 'Automatic Presentations of Structures'. In: *Logical and Computational Complexity. Selected Papers. Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, Indiana, USA, 13-16 October 1994*. Ed. by Daniel Leivant. Vol. 960. Lecture Notes in Computer Science. Springer, 1994, pp. 367–392. doi: 10.1007/3-540-60178-3_93 (cited on page 62).

- [76] Achim Blumensath and Erich Grädel. ‘Automatic Structures’. In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 2000, pp. 51–62. doi: 10.1109/LICS.2000.855755 (cited on page 62).
- [77] Edward F. Moore. ‘Gedanken-Experiments on Sequential Machines’. In: *Automata Studies*. Ed. by Claude Shannon and John McCarthy. Princeton, NJ: Princeton University Press, 1956, pp. 129–153 (cited on page 63).
- [78] G. H. Mealy. ‘A method for synthesizing sequential circuits’. In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079. doi: 10.1002/j.1538-7305.1955.tb03788.x (cited on page 63).
- [79] Donald A. Martin. ‘Borel Determinacy’. In: *Annals of Mathematics* 102.2 (1975), pp. 363–371 (cited on page 67).
- [80] Yuri Gurevich and Leo Harrington. ‘Trees, Automata, and Games’. In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*. Ed. by Harry R. Lewis et al. ACM, 1982, pp. 60–65. doi: 10.1145/800070.802177 (cited on page 67).
- [81] J. Richard Büchi. ‘State-Strategies for Games in $F_{\sigma\delta} \cap G_{\delta\sigma}$ ’. In: *The Journal of Symbolic Logic* 48.4 (1983), pp. 1171–1198. doi: 10.2307/2273681 (cited on page 67).
- [82] Feng-Hsiung Hsu, Murray Campbell, and A. Joseph Hoane Jr. ‘Deep Blue System Overview’. In: *Proceedings of the 9th international conference on Supercomputing, ICS 1995, Barcelona, Spain, July 3-7, 1995*. Ed. by Mateo Valero. ACM, 1995, pp. 240–244. doi: 10.1145/224538.224567 (cited on page 67).
- [83] David Silver et al. ‘A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play’. In: *Science* 362.6419 (2018), pp. 1140–1144 (cited on page 67).
- [84] Laurent Doyen and Jean-François Raskin. ‘Games with imperfect information: theory and algorithms’. In: *Lectures in Game Theory for Computer Scientists*. Ed. by Krzysztof R. Apt and Erich Grädel. Cambridge University Press, 2011, pp. 185–212 (cited on page 67).
- [85] A. W. Mostowski. ‘Regular expressions for infinite trees and a standard form of automata’. In: *Computation Theory*. Ed. by Andrzej Skowron. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 157–168 (cited on page 67).
- [86] E. Allen Emerson and Charanjit S. Jutla. ‘The Complexity of Tree Automata and Logics of Programs (Extended Abstract)’. In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 328–337. doi: 10.1109/SFCS.1988.21949 (cited on page 68).
- [87] A.W. Mostowski. *Games with Forbidden Positions*. Tech. rep. Technical Report 78. Uniwersytet Gdański. Instytut Matematyki, 1991 (cited on page 68).
- [88] Wiesław Zielonka. ‘Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees’. In: *Theor. Comput. Sci.* 200.1-2 (1998), pp. 135–183. doi: 10.1016/S0304-3975(98)00009-7 (cited on pages 68, 150, 194).
- [89] Cristian S. Calude et al. ‘Deciding parity games in quasipolynomial time’. In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*. Ed. by Hamed Hatami, Pierre McKenzie, and Valerie King. ACM, 2017, pp. 252–263. doi: 10.1145/3055399.3055409 (cited on pages 68, 150).
- [90] Marcin Jurdzinski and Ranko Lazic. ‘Succinct progress measures for solving parity games’. In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–9. doi: 10.1109/LICS.2017.8005092 (cited on page 68).
- [91] John Fearnley et al. ‘An ordered approach to solving parity games in quasi polynomial time and quasi linear space’. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*. Ed. by Hakan Erdogmus and Klaus Havelund. ACM, 2017, pp. 112–121. doi: 10.1145/3092282.3092286 (cited on page 68).
- [92] Karoliina Lehtinen. ‘A modal μ perspective on solving parity games in quasi-polynomial time’. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 639–648. doi: 10.1145/3209108.3209115 (cited on page 68).

- [93] Paweł Parys. 'Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time'. In: *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany*. Ed. by Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen. Vol. 138. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 10:1–10:13. doi: 10.4230/LIPIcs.MFCS.2019.10 (cited on page 68).
- [94] Shmuel Safra. 'Exponential Determinization for omega-Automata with a Strong Fairness Acceptance Condition'. In: *SIAM J. Comput.* 36.3 (2006), pp. 803–814. doi: 10.1137/S0097539798332518 (cited on page 69).
- [95] Sven Schewe and Thomas Varghese. 'Determinising Parity Automata'. In: *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*. Ed. by Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik. Vol. 8634. Lecture Notes in Computer Science. Springer, 2014, pp. 486–498. doi: 10.1007/978-3-662-44522-8_41 (cited on pages 69, 149).
- [96] Emmanuel Filiot et al. 'On Equivalence and Uniformisation Problems for Finite Transducers'. In: *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*. Vol. 55. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 125:1–125:14. doi: 10.4230/LIPIcs.ICALP.2016.125 (cited on page 69).
- [97] Thomas A. Henzinger and Nir Piterman. 'Solving Games Without Determinization'. In: *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*. Ed. by Zoltán Ésik. Vol. 4207. Lecture Notes in Computer Science. Springer, 2006, pp. 395–410. doi: 10.1007/11874683_26 (cited on pages 70, 111, 166).
- [98] Thomas Colcombet. 'The Theory of Stabilisation Monoids and Regular Cost Functions'. In: *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part II*. Ed. by Susanne Albers et al. Vol. 5556. Lecture Notes in Computer Science. Springer, 2009, pp. 139–150. doi: 10.1007/978-3-642-02930-1_12 (cited on pages 70, 166).
- [99] Denis Kuperberg and Michał Skrzypczak. 'On Determinisation of Good-for-Games Automata'. In: *Automata, Languages, and Programming*. Ed. by Magnús M. Halldórsson et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 299–310 (cited on page 70).
- [100] Wolfgang Thomas. 'Solution of Church's Problem: A tutorial'. In: *New Perspectives on Games and Interaction 4* (Jan. 2008) (cited on page 70).
- [101] Frank Neven, Thomas Schwentick, and Victor Vianu. 'Finite state machines for strings over infinite alphabets'. In: *ACM Trans. Comput. Log.* 5.3 (2004), pp. 403–435. doi: 10.1145/1013560.1013562 (cited on pages 71, 82, 86, 97, 118, 119, 131, 152, 154, 160, 196, 198, 200, 213, 306).
- [102] Marvin L. Minsky. 'Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines'. In: *Annals of Mathematics* 74.3 (1961), pp. 437–455 (cited on pages 74, 77, 121, 179).
- [103] Henrik Björklund and Thomas Schwentick. 'On notions of regularity for data languages'. In: *Theor. Comput. Sci.* 411.4-5 (2010), pp. 702–715. doi: 10.1016/j.tcs.2009.10.009 (cited on pages 76, 82, 86, 131, 145, 200).
- [104] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. 'Bisimilarity in Fresh-Register Automata'. In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. IEEE Computer Society, 2015, pp. 156–167. doi: 10.1109/LICS.2015.24 (cited on page 81).
- [105] Bartek Klin, Slawomir Lasota, and Szymon Torunczyk. 'Nondeterministic and co-Nondeterministic Implies Deterministic, for Data Languages'. In: *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by Stefan Kiefer and Christine Tasson. Vol. 12650. Lecture Notes in Computer Science. Springer, 2021, pp. 365–384. doi: 10.1007/978-3-030-71995-1_19 (cited on page 82).
- [106] Michael O. Rabin and Dana S. Scott. 'Finite Automata and Their Decision Problems'. In: *IBM J. Res. Dev.* 3.2 (1959), pp. 114–125. doi: 10.1147/rd.32.0114 (cited on page 82).

- [107] Mikołaj Bojańczyk and Rafał Stefanski. ‘Single use register automata for data words’. In: *CoRR abs/1907.10504* (2019) (cited on page 82).
- [108] Mikołaj Bojańczyk and Rafał Stefanski. ‘Single-Use Automata and Transducers for Infinite Alphabets’. In: *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*. Ed. by Artur Czumaj, Anuj Dawar, and Emanuela Merelli. Vol. 168. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 113:1–113:14. doi: 10.4230/LIPIcs.ICALP.2020.113 (cited on pages 82, 131, 305, 307).
- [109] Mikolaj Bojanczyk. ‘Data Monoids’. In: *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*. Ed. by Thomas Schwentick and Christoph Dürr. Vol. 9. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011, pp. 105–116. doi: 10.4230/LIPIcs.STACS.2011.105 (cited on page 82).
- [110] Thomas Colcombet, Clemens Ley, and Gabriele Puppis. ‘Logics with rigidly guarded data tests’. In: *Log. Methods Comput. Sci.* 11.3 (2015). doi: 10.2168/LMCS-11(3:10)2015 (cited on page 82).
- [111] Hiroshi Sakamoto and Daisuke Ikeda. ‘Intractability of decision problems for finite-memory automata’. In: *Theor. Comput. Sci.* 231.2 (2000), pp. 297–308. doi: 10.1016/S0304-3975(99)00105-X (cited on page 97).
- [112] Antoine Mottet and Karin Quaas. ‘The Containment Problem for Unambiguous Register Automata’. In: *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*. Ed. by Rolf Niedermeier and Christophe Paul. Vol. 126. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 53:1–53:15. doi: 10.4230/LIPIcs.STACS.2019.53 (cited on page 98).
- [113] Corentin Barloy and Lorenzo Clemente. ‘Bidimensional Linear Recursive Sequences and Universality of Unambiguous Register Automata’. In: *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*. Ed. by Markus Bläser and Benjamin Monmege. Vol. 187. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 8:1–8:15. doi: 10.4230/LIPIcs.STACS.2021.8 (cited on page 98).
- [114] Mikolaj Bojanczyk, Bartek Klin, and Joshua Moerman. ‘Orbit-Finite-Dimensional Vector Spaces and Weighted Register Automata’. In: *CoRR abs/2104.02438* (2021) (cited on page 98).
- [115] Annalisa Marcja and Carlo Toffalori. ‘Quantifier Elimination’. In: *A Guide to Classical and Modern Model Theory*. Dordrecht: Springer Netherlands, 2003, pp. 43–83. doi: 10.1007/978-94-007-0812-9_2 (cited on pages 104, 105, 215).
- [116] Stéphane Demri and Deepak D’Souza. ‘An automata-theoretic approach to constraint LTL’. In: *Inf. Comput.* 205.3 (2007), pp. 380–415. doi: 10.1016/j.ic.2006.09.006 (cited on pages 109, 183, 184).
- [117] Mikołaj Bojańczyk. ‘Weak MSO with the unbounding quantifier’. In: *Theory of Computing Systems* 48.3 (2011), pp. 554–576 (cited on pages 110, 116–118).
- [118] F. P. Ramsey. ‘On a Problem of Formal Logic’. In: *Proceedings of the London Mathematical Society* s2-30.1 (Jan. 1930), pp. 264–286. doi: 10.1112/plms/s2-30.1.264 (cited on pages 114, 115, 280).
- [119] A. M. Turing. ‘On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction’. In: *Proceedings of the London Mathematical Society* s2-43.1 (Jan. 1938), pp. 544–546. doi: 10.1112/plms/s2-43.6.544 (cited on pages 121, 223).
- [120] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013 (cited on pages 121, 173, 223).
- [121] Nikos Tzevelekos. ‘Fresh-register automata’. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 295–306. doi: 10.1145/1926385.1926420 (cited on pages 122, 127, 128).
- [122] D. Zeitlin. ‘Look-ahead finite-memory automata’. MA thesis. Technion - Israel Institute of Technology, 2006 (cited on pages 124, 126).

- [123] Mikolaj Bojanczyk et al. ‘Two-variable logic on data words’. In: *ACM Trans. Comput. Log.* 12.4 (2011), 27:1–27:26. doi: 10.1145/1970398.1970403 (cited on page 131).
- [124] Michael Benedikt, Clemens Ley, and Gabriele Puppis. ‘Automata vs. Logics on Data Words’. In: *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23–27, 2010. Proceedings*. Ed. by Anuj Dawar and Helmut Veith. Vol. 6247. Lecture Notes in Computer Science. Springer, 2010, pp. 110–124. doi: 10.1007/978-3-642-15205-4_12 (cited on pages 131, 200).
- [125] Loris D’Antoni. ‘In the Maze of Data Languages’. In: *CoRR abs/1208.5980* (2012) (cited on page 131).
- [126] J. Roitman. *Introduction to Modern Set Theory*. A Wiley-interscience publication. Wiley, 1990 (cited on page 151).
- [127] Bruno Courcelle. ‘Monadic Second-order Definable Graph Transductions: A Survey’. In: *Theor. Comput. Sci.* 126.1 (Apr. 1994), pp. 53–75. doi: 10.1016/0304-3975(94)90268-2 (cited on page 155).
- [128] Luc Dartois, Emmanuel Filiot, and Nathan Lhote. ‘Logics for Word Transductions with Synthesis’. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09–12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 295–304. doi: 10.1145/3209108.3209181 (cited on pages 155, 200).
- [129] Perdita Stevens. ‘Abstract interpretations of games’. English. In: *Proc. 2nd International Workshop on Verification, Model Checking and Abstract Interpretation, VMCAI’98*. 1998 (cited on page 166).
- [130] D. Berend and T. Tassa. ‘Improved bounds on bell numbers and on moments of sums of random variables’. In: *Probability and Mathematical Statistics* Vol. 30, Fasc. 2 (2010), pp. 185–205 (cited on page 173).
- [131] Bartek Klin and Mateusz Lelyk. ‘Scalar and Vectorial mu-calculus with Atoms’. In: *Log. Methods Comput. Sci.* 15.4 (2019). doi: 10.23638/LMCS-15(4:5)2019 (cited on page 177).
- [132] Mikołaj Bojańczyk. ‘Weak MSO+U with Path Quantifiers over Infinite Trees’. In: *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8–11, 2014, Proceedings, Part II*. 2014, pp. 38–49 (cited on page 183).
- [133] Shaull Almagor and Orna Kupferman. ‘Good-Enough Synthesis’. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 541–563. doi: 10.1007/978-3-030-53291-8_28 (cited on page 195).
- [134] Mikolaj Bojanczyk et al. ‘Two-Variable Logic on Words with Data’. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12–15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, pp. 7–16. doi: 10.1109/LICS.2006.51 (cited on pages 200, 310).
- [135] Thomas Schwentick and Thomas Zeume. ‘Two-Variable Logic with Two Order Relations’. In: *Log. Methods Comput. Sci.* 8.1 (2012). doi: 10.2168/LMCS-8(1:15)2012 (cited on pages 200, 310).
- [136] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. ‘Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems’. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 71–89. doi: 10.1007/978-3-662-49674-9_5 (cited on page 200).
- [137] Radu Iosif and Xiao Xu. ‘Abstraction Refinement for Emptiness Checking of Alternating Data Automata’. In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, Part II*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10806. Lecture Notes in Computer Science. Springer, 2018, pp. 93–111. doi: 10.1007/978-3-319-89963-3_6 (cited on page 200).

- [138] Rüdiger Ehlers. ‘Unbeast: Symbolic Bounded Synthesis’. In: *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Vol. 6605. Lecture Notes in Computer Science. Springer, 2011, pp. 272–275. doi: 10.1007/978-3-642-19835-9_25 (cited on page 201).
- [139] Edmund M. Clarke et al. ‘Counterexample-guided abstraction refinement for symbolic model checking’. In: *J. ACM* 50.5 (2003), pp. 752–794. doi: 10.1145/876638.876643 (cited on page 201).
- [140] Yoad Lustig and Moshe Y. Vardi. ‘Synthesis from component libraries’. In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 603–618. doi: 10.1007/s10009-012-0236-z (cited on page 201).
- [141] Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. ‘On Computability of Data Word Functions Defined by Transducers’. In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*. Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Lecture Notes in Computer Science. Springer, 2020, pp. 217–236. doi: 10.1007/978-3-030-45231-5_12 (cited on pages 204, 214, 232–235, 239).
- [142] Léo Exibard et al. ‘Computability of Data-Word Transductions over Different Data Domains’. In: *Logical Methods in Computer Science* (2021). Ed. by Jean Goubault-Larrecq and Barbara König (cited on pages 204, 222, 224, 252, 267, 269, 270, 282, 291, 292, 294, 302, 303).
- [143] Jacques Sakarovitch. *Éléments de théorie des automates*. fre. Les classiques de l’informatique. Paris: Vuibert informatique, 2003 (cited on page 205).
- [144] Marie-Pierre Béal et al. ‘Squaring transducers: an efficient procedure for deciding functionality and sequentiality’. In: *Theor. Comput. Sci.* 292.1 (2003), pp. 45–63. doi: 10.1016/S0304-3975(01)00214-6 (cited on page 210).
- [145] Klaus Weihrauch. *Computable Analysis - An Introduction*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000 (cited on page 216).
- [146] Françoise Gire. ‘Two Decidability Problems for Infinite Words’. In: *Inf. Process. Lett.* 22.3 (1986), pp. 135–140. doi: 10.1016/0020-0190(86)90058-X (cited on page 228).
- [147] Emmanuel Filiot, Nicolas Mazzocchi, and Jean-François Raskin. ‘A Pattern Logic for Automata with Outputs’. In: *Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10–14, 2018, Proceedings*. 2018, pp. 304–317 (cited on pages 228, 229, 239).
- [148] Christian Choffrut. ‘Une Caractérisation des Fonctions Séquentielles et des Fonctions Sous-Séquentielles en tant que Relations Rationnelles’. In: *Theor. Comput. Sci.* 5.3 (1977). in French, pp. 325–337. doi: 10.1016/0304-3975(77)90049-4 (cited on page 229).
- [149] Emmanuel Filiot, Nicolas Mazzocchi, and Jean-François Raskin. ‘A Pattern Logic for Automata with Outputs’. In: *Int. J. Found. Comput. Sci.* 31.6 (2020), pp. 711–748 (cited on pages 229, 230, 311, 318).
- [150] Neil Immerman. ‘Nondeterministic Space is Closed Under Complementation’. In: *SIAM J. Comput.* 17.5 (1988), pp. 935–938. doi: 10.1137/0217058 (cited on pages 256, 318).
- [151] Róbert Szelepcsényi. ‘The Method of Forced Enumeration for Nondeterministic Automata’. In: *Acta Informatica* 26.3 (1988), pp. 279–284. doi: 10.1007/BF00299636 (cited on pages 256, 318).
- [152] Walter J. Savitch. ‘Relationships Between Nondeterministic and Deterministic Tape Complexities’. In: *J. Comput. Syst. Sci.* 4.2 (1970), pp. 177–192. doi: 10.1016/S0022-0000(70)80006-X (cited on pages 256, 268).
- [153] Marcel Paul Schützenberger. ‘A Remark on Finite Transducers’. In: *Inf. Control.* 4.2-3 (1961), pp. 185–196. doi: 10.1016/S0019-9958(61)80006-5 (cited on page 306).
- [154] Samuel Eilenberg. *Automata, languages, and machines*. A. Pure and applied mathematics. Academic Press, 1974 (cited on page 306).
- [155] Christophe Reutenauer and Marcel Paul Schützenberger. ‘Minimization of Rational Word Functions’. In: *SIAM J. Comput.* 20.4 (1991), pp. 669–685. doi: 10.1137/0220042 (cited on page 306).

- [156] Rajeev Alur and Pavol Cerný. ‘Expressiveness of streaming string transducers’. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*. Ed. by Kamal Lodaya and Meena Mahajan. Vol. 8. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 1–12. doi: 10.4230/LIPIcs.FSTTCS.2010.1 (cited on page 307).
- [157] Wladimir Fridman, Christof Löding, and Martin Zimmermann. ‘Degrees of Lookahead in Context-free Infinite Games’. In: *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings*. Ed. by Marc Bezem. Vol. 12. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011, pp. 264–276 (cited on page 307).
- [158] Emmanuel Filiot and Sarah Winter. ‘Continuous Uniformization of Rational Relations and Synthesis of Computable Functions’. In: *CoRR abs/2103.05674* (2021) (cited on page 307).
- [159] Piotr Hofman et al. ‘Parikh’s theorem for infinite alphabets’. In: *CoRR abs/2104.12018* (2021) (cited on page 311).
- [160] David Harel. ‘Can Programming Be Liberated, Period?’ In: *Computer* 41.1 (2008), pp. 28–37. doi: 10.1109/MC.2008.10 (cited on page 311).
- [161] Dana Angluin. ‘Learning Regular Sets from Queries and Counterexamples’. In: *Inf. Comput.* 75.2 (1987), pp. 87–106. doi: 10.1016/0890-5401(87)90052-6 (cited on page 311).
- [162] François Denis, Aurélien Lemay, and Alain Terlutte. ‘Residual Finite State Automata’. In: *Fundam. Informaticae* 51.4 (2002), pp. 339–368 (cited on page 311).
- [163] Benedikt Bollig et al. ‘Angluin-Style Learning of NFA’. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 1004–1009 (cited on page 311).
- [164] Benedikt Bollig et al. ‘A Fresh Approach to Learning Register Automata’. In: *Developments in Language Theory - 17th International Conference, DLT 2013, Marne-la-Vallée, France, June 18-21, 2013. Proceedings*. Ed. by Marie-Pierre Béal and Olivier Carton. Vol. 7907. Lecture Notes in Computer Science. Springer, 2013, pp. 118–130. doi: 10.1007/978-3-642-38771-5_12 (cited on page 311).
- [165] Joshua Moerman et al. ‘Learning nominal automata’. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 613–625. doi: 10.1145/3009837.3009879 (cited on page 311).
- [166] Sofia Cassel et al. ‘A succinct canonical register automaton model’. In: *J. Log. Algebraic Methods Program.* 84.1 (2015), pp. 54–66. doi: 10.1016/j.jlamp.2014.07.004 (cited on page 311).
- [167] Christian Choffrut. ‘Minimizing subsequential transducers: a survey’. In: *Theor. Comput. Sci.* 292.1 (2003), pp. 131–143. doi: 10.1016/S0304-3975(01)00219-5 (cited on page 311).

Index of Notations

$f \circ g$	Composition of f and g , i.e. $x \mapsto f(g(x))$
$[\nu]$	Constraint associated with valuation ν , with regards to $(\mathbb{D}, =, \mathfrak{C})$
$[\nu]_{\mathbb{Q}}$	Constraint associated with valuation ν , with regards to $(\mathbb{Q}, <, 0)$
$\mathcal{D}$	Data domain
$\mathbb{D}$	Set of data values of $\mathcal{D}$
$\text{type}_{\mathcal{D}}(\nu)$	Type of ν in $\mathcal{D}$
$\text{Types}(\mathcal{D})$	Set of type of $\mathcal{D}$
$\text{Aut}(\mathcal{D})$	Set of automorphisms of $\mathcal{D}$
dom	Domain of a function
Im	Image of a function
FO	First-order formulas
LTL	Linear temporal logic formulas
MSO	Monadic second-order formulas
QF	Quantifier-free formulas
$\llbracket S \rrbracket$	Relation recognised by the specification automaton S
$\llbracket T \rrbracket$	Relation recognised by the transducer T
$\mathbb{N}$	Natural numbers
$\mathbb{Q}$	Rational numbers
$\mathbb{R}$	Real numbers
$\mathbb{Z}$	Integers
Ω_P	Omega-behaviour of P
$[x]_{\sim}$	Equivalence class of x with regards to $\sim$
$\sim$	Equivalence relation
$\cup$	Union
$\cap$	Intersection
A^c	Complement of A
$\sqcup$	Disjoint union

$A \subseteq B$ A is a subset of B

$A \subsetneq B$ A is a strict subset of B

$A \subset_f B$ A is a finite subset of B

$|A|$ Size of the set A

$\text{Tests}_{\mathcal{D}}(R)$ Set of tests over registers R in $\mathcal{D}$

$\text{locFresh} R \ \bigwedge_{r \in R} \star \neq r$

$\circledast$ Globally fresh data value (see fresh-register automata)

$\phi_E \ \bigwedge_{r \in E} \star = r \wedge \bigwedge_{r \notin E} \star \neq r$

ψ_τ Test associated with constraint τ

A^* Finite words over the set A

A^ω Infinite words over the set A

$A^\infty \ A^* \cup A^\omega$

ε Empty word

$w[i]$ i -th letter of w

$w[i:j]$ $w[i] \dots w[j]$

$w[:i]$ $w[0:i]$

$w[i:]$ $w[i: + \infty]$

$|w|$ Length of the word w

$\langle u, v \rangle$ Interleaving of u and v

$\langle R \rangle$ Interleaving of the relation R

$\rangle w \langle$ Uninterleaving of w

$u \otimes v$ Pointwise word product of u and v