CHAPITRE 1

Introduction

Ce mémoire est dédié à l'optimisation de code. L'optimisation de code fait partie de la phase finale d'un compilateur (Figure 1). On distingue, généralement, deux phases d'optimisation. La première est l'optimisation du code intermédiaire. Elle consiste à rendre la représentation interne fournie par le générateur de code intermédiaire la plus efficace possible. Le code intermédiaire est, plus au moins[1], indépendant de la machine cible.

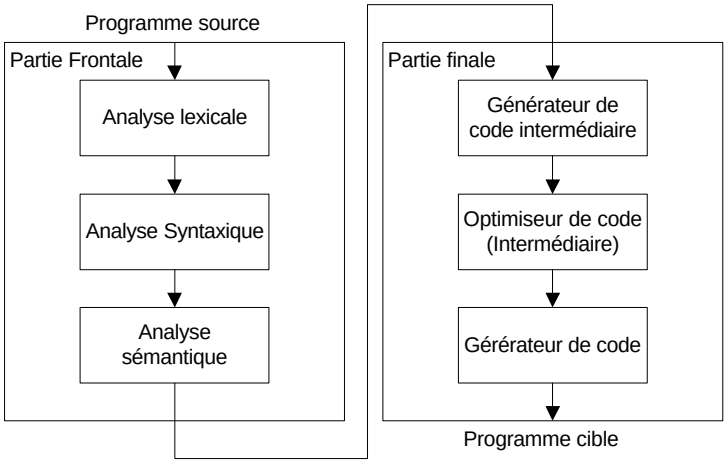

Figure 1 Structure d'un compilateur

La seconde phase d'optimisation s'effectue par le générateur de code final. Elle s'effectue dans l'optique de production de code cible. Les aspects architecturaux de la machine cible sont pris en compte dans le but de produire le code le plus efficace.

Dans un souci de généralité (ne pas s'attacher à une architecture particulière) et de concision dans les objectifs fixés en 1.1, nous développerons uniquement des optimisations dans le cadre de la production de code intermédiaire.

[1] Selon les besoins et les objectifs du compilateur.

1.1 OBJECTIFS

L'objectif principal est de développer les optimisations globales classiques dans un cadre moderne. Les optimisations abordées seront : la propagation conditionnelle des constantes, l'élimination du code inutile, l'élimination des redondances partielles et la réduction de force. Ces quatre optimisations ont été choisies de par leur quasi inévitabilité dans un compilateur optimisant.

Les optimisations présentées sont depuis longtemps étudiées. De nombreuses solutions exploitables ont été découvertes. Les fondements de ces différentes méthodes sont basés sur l'analyse de flot de données. L'analyse de flot de données consiste en l'évaluation des propriétés d'un programme. Un grand nombre de ces méthodes sont orientées vecteurs de bits. Elles consistent à partir d'un sous ensemble trivialement déterminé à la propagation de ce sous ensemble en tous les points du programme jusqu'à convergence. Elles effectuent une estimation itérative incrémentale de la propriété. Ces méthodes possèdent le défaut de propager les informations partout dans le programme même aux endroits auxquels l'information n'est pas nécessaire.

De cette constatation est née l'idée de l'évaluation éparse. Cette dernière est une évaluation qui détermine la propriété étudiée, uniquement aux endroits auxquels cela est nécessaire. La propagation ne peut plus se faire le long de la structure de contrôle du programme, mais le long d'une structure alternative dont les composants principaux sont les endroits du programme auxquels la propriété doit être évaluée.

Une représentation alternative du programme (que nous appellerons représentation éparse) contribue fortement à l'évaluation éparse. Nous développerons donc une telle représentation. Cette représentation comprendra intrinsèquement la structure sur laquelle les différentes optimisations s'appuieront pour effectuer leur(s) analyse(s) de flot respective(s).

Dans ce mémoire, nous établirons les tenants et les aboutissants de l'évaluation éparse des propriétés basée sur une représentation éparse du programme. Les implications seront illustrées par les différentes optimisations exposées.

L'évaluation des performances des optimisations, dans le cadre d'une implémentation particulière, ne sera pas développée. Nous nous baserons sur les complexités temporelles théoriques. Des références vers de telles analyses seront néanmoins dispersées dans le texte aux endroits nécessaires. Ce type d'analyse, trop vaste pour être développé dans ce mémoire, mériterait cependant de faire l'objet d'une étude approfondie, dans un travail ultérieur.

1.2 STRUCTURE DU MEMOIRE

Ce mémoire est composé de trois parties distinctes. Dans la première, nous exposerons les préliminaires mathématiques et informatiques nécessaires au développement des différents points abordés.

Dans la seconde partie, nous développerons la représentation éparse choisie (Single Static Assignment : SSA). Nous décrirons une méthode efficace de

construction de cette représentation. Nous donnerons également une méthode pour revenir de cette représentation après optimisation du programme.

La dernière partie sera consacrée aux méthodes d'optimisation proprement dites. La propagation conditionnelle des constantes, l'élimination du code inutile, l'élimination des redondances partielles et la réduction de force, y seront développées dans le cadre de la forme SSA.

Nous clôturerons par la synthèse des caractéristiques relevées et en tirerons les conclusions.