Langage de Programmation et Compilation

Jean-Cristophe Filliâtre

6 octobre 2023

Table des matières

Première partie

Cours 1 29/09

Introduction

Maîtriser les mécanismes de la compilation, transformation d'un langage dans un autre. Comprendre les aspects des langages de programmation.

0.1 Un Compilateur

Un compilateur est un traducteur d'un langage source vers un langage cible. Ici le langage cible sera l'asembleur.
Tous les langages ne sont pas compilés à l'avance, certains sont interprétés, transpilés puis interprétés, compilés à la volée, transpilés puis compilés... Un compilateur prend un programme P et le traduit en un programme Q de sorte que : $\forall P, \exists Q, \forall x,\ P(x) = Q(x)$. Un interpréteur effectue un travail simple mais le refait à chaque entrée, et donc est moins efficace.
Exemple : le langage *lilypond* va compiler un code source en fichier .pdf.

0.2 Le Bon et le Mauvais Compilateur

On juge un compilateur à :

1. Sa correction
2. L'efficacité du code qu'il produit
3. Son efficacité en tant que programme

> « Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free ! Thus, the most important objective in writing a compiler is that it is correct »- *Dragon Book, 2006*

0.3 Le Travail d'un Compilateur

Le travail d'un compilateur se compose :

— d'une phase d'analyse qui :
   1. reconnaît le programme à traduire et sa signification
   2. signale les erreurs et peut donc échouer

— d'une phase de synthèse qui :
   1. produit du langage cible
   2. utilise de nombreux langages intermédiaires
   3. n'échoue pas

Processus : source → analyse lexicale → suite de lexèmes (tokens) → analyse syntaxique → Arbre de syntaxe abstraite → analyse sémantique → syntaxe abstraite + table des symboles → production de code → langage assembleur → assembleur → langage machine → éditeur de liens → exécutable.

1 L'assembleur

1.1 Arithmétique des ordinateurs

On représente les entiers sur n bits numérotés de droite à gauche. Typiquement, n vaut 8, 16, 32 ou 64. On peut représenter des entiers non signés jusqu'à $2^n - 1$. On peut représenter les entiers en définissant b_{n-1} comme un bit de signe, on peut alors représenter $\left[-2^{n-1}, 2^{n-1} - 1\right]$. La valeur d'une suite de bits est alors : $-b_{n-1}2^{n-1} + \sum_{k=0}^{n-2} b_k 2^k$. On ne peut pas savoir si un entier est signé sans le contexte.
La machine fournit des opérations logiques (bit à bit), de décalage (ajout de bits 0 de poids fort, 0 de poids faible ou réplication du bit de signe pour interpréter une division), d'arithmétique (addition, soustraction, multiplication).

1.2 Architecture

Un ordinateur contient :

— Une unité de calcul (CPU) qui contient un petit nombre de registres et des capactités de calcul

— Une mémoire vive (RAM), composée d'un très grand nombre d'octets (8 bits), et des données et des instructions, indifférenciables sans contexte.

L'accès à la mémoire coûte cher : à 1B instructions/s, la lumière ne parcourt que 30cm entre deux instructions.
En réalité, il y a plusieurs (co)processeurs, des mémoires cache, une virtualisation de la mémoire...
Principe d'exécution : un registre (%rip) contient l'adresse de l'instruction, on lit (fetch) un ou plusieurs octects dans la mémoire, on interprète ces bits (decode), on exécute l'instruction (exectue), on modifie (%rip) pour l'instruction suivante. En réalité, on a des pipelines qui branchent plusieurs instructions en parallèle, et on essaie de prédire les sauts conditionnels.
Deux grandes familles d'Architectures : CISC (complex instruction set), qui permet beaucoup d'instructions différentes mais avec assez peu de registres, et RISC (Reduced Instruction Set) avec peu d'instruction effectuées très régulièrements et avec beaucoup de registres. Ici, on utilisera l'architecture *x86-64*.

1.3 L'architecture `x86-64`

Extension 64 bits d'une famille d'architectures compatibles Intel par AMD adoptée par Intel. Architecture à 16 registres, avec adressage sur 48 bits au moins et de nombreux modes d'adressage. On ne programme pas en langage machine mais en assembleur, langage symbolique avec allocation de données globales, qui est transformé en langage machine par un assembleur qui est en réalité un compilateur. On utilise l'assembleur GNU avec la syntaxe AT&T (la syntaxe Intel existe aussi).

1.4 L'assembleur `x86-64`

Pour assembler un programme assembleur, appeler `as -o file.o` puis appeler l'édition de lien avec `gcc -no-pie file.s -o exec-name`. On peut débuguer en ajoutant l'option `-g`. La machine est petite boutiste (little-endian) si elle stocke les valeurs dans la RAM en commençant par le bit de poids faible, gros boutiste (big-endian) pour le poids fort.
Commandes : Dans cette liste, $\%(r)$ désigne l'adresse mémoire stockée dans r

— `movq $a %b` permet de mettre la valeur a dans le registre b

— `movq %a %b` permet de copier le registre a dans le registre b

— `movq $label %b` permet de changer l'adresse de l'étiquette dans le registre b

— `addq %a %b` permet d'additionner les registres a et b.

— `incq %r` permet d'incrémenter le registre r, de même pour `decq`.

— `negq %r` permet de modifier la valeur de r en sa négation

— `notq %r` permet de modifier la valeur de r en sa négation logique.

— `orq %r1 %r2` (resp. `andq` et `xorq`) permet d'affecter à $r2$, $or(r1, r2)$ (resp. and, xor)

— `salq $n %r`/`salq %cl %r` décale la valeur de r de n (ou $\%cl$) zéros à gauche.

— `sarq` est le décalage à droite arithmétique, `shrq` le décalage à droite logique.

— Le suffixe `q` désigne une opération sur 64 bits. `b` désigne 1 octet, `w` désigne 2 octets, `l` désigne 4 octets. Il faut préciser les deux extensions si celles-ci diffèrent.

— `jmp label` permet de jump à une étiquette.

La plupart des opérations fositionnent des drapeaux selon leur résultat.
Certaines instructions : `j(suffixe)` (jump), `set(suffixe)` et `cmov(suffixe)`(move) permettent de tester des drapeaux et d'effectuer une opération selon leur valeur.

On ne sait pas combien il y a d'instructions en `x86-64`.

1.5 Le Défi de la Compilation

C'est de traduire un programme d'un langage de haut niveau vers ce jeu d'instruction.
Constat : les appels de fonctions peuvent être arbitrairement imbriqués et les registres ne suffisent pas ⇒ on crée alors une pile car les fonctions procèdent majoritairement selon un mode LIFO.
La pile est stockée tout en haut, et croît dans le sens des adresses décroissantes, `%rsp` poînte sur le sommet de la pile. Les données dynamiques sont allouées sur le tas, en bas de la zone de données.
Chaque programme a l'illusion d'avoir toute la mémoire pour lui tout seul, illusion créée par l'OS.
En assembleur on a des facilités d'utilisation de la pile :

— `pushq $a` push a dans la pile

— `popq %rdi` dépile

Lorsque f (caller) appelle une fonction g (callee), on ne peut pas juste `jmp g`. On utilise `call g` puis une fois que c'est terminé `ret`.
Mézalor tout registre utilisé par g sera perdu par f. On s'accorde alors sur des **conventions d'appel**. Des arguments sont passés dans certains registres, puis sur la pile, la valeur de retour est passéee dans `%rax`. Certains registres sont *callee-saved* i.e. l'appelé doit les sauvegarder pour qu'elle survivent aux appels. Les autres registres sont dit *caller-saved* et ne vont pas survivre aux appels.
Il faut également qu'en entrée de fonction `%rsp + 8` doît être multiple de 16, sinon des fonctions peuvent planter.
Il y a quatres temps dans un appel :

1. Pour l'appelant, juste avant l'appel :
2. Pour l'appelé, au début de l'appel :
   (a) Sauvegarde `%rbp` puis le positionne.
   (b) Alloue son tableau d'activation.
   (c) Sauvegarde les registres *callee-saved*.
3. Pour l'appelé, à la fin de l'appel :
   (a) Placer le résultat dans `%rax`
   (b) Restaure les registres sauvegardés
   (c) Dépile sont tableau d'activation
   (d) Exécute `ret`
4. Pour l'appelant, juste après l'appel :
   (a) Dépile les éventuels arguments
   (b) Restaure les registres *caller-saved*

Deuxième partie
Cours 2 : 6/10

Introduction

La signification des programmes est définie souvent de manière informelle, en langue naturelle, e.g. le langage Java.

2 Sémantique Formelle

La sémantique formelle caractérise mathématiquement les calculs décrits par un programme. C'est utile pour la réalisation d'outils (interprètes, compilateurs), et nécessaire aux raisonnements sur les programmes.

2.1 Syntaxe Abstraite

On ne peut pas manipuler un programme en tant qu'object syntaxique, on préfère utiliser la syntaxe abstraite (se déduit lors de la compilation à l'analyse syntaxique et sémantique.). On construit un arbre de syntaxe abstraite pour comprendre.
On définit la syntaxe abstraite par une grammaire. En OCamL, on réalise la syntaxe abstraite par des types construits. Il n'y a pas de parenthèses dans la syntaxe abstraite. On appelle sucre syntaxique toute construction de la syntaxe concrète qui n'existe pas dans la syntaxe abstraite.

C'est sur la syntaxe abstraite qu'on va définir la sémantique.

2.2 Sémantiques Useless

2.2.1 Sémantique Axiomatique - Logique de Floyd-Hoare

Tony Hoare, An axiomatic basis for computer programming, 1969, article le plus cité de l'histoire de l'informatique.
On caractérise les programmes par l'intermédiaire des propriétés satsfaites par les variables. On introduit le triplet $\{P\}\ i\ \{Q\}$ qui signifie, si P est vraie avant l'instruction i, après, Q sera vraie

2.2.2 Sémantique Dénotationelle

A chaque expression e, on associe sa définition $\|e\|$ qui est un objet représentant le calcul désigné par e. On définit cet objet récursivement.

2.2.3 Sémantique Par Traduction

On définit la sémantique d'un langage en le traduisant vers un langage dont la sémantique est connue.

2.3 Sémantique Opérationnelle

La sémantique opérationnelle décrit l'enchaînement des calculs élémentaires qui mènent de l'expression à son résultat.
Il y a deux formes de sémantique opérationelle :

1. La sémantique naturelle (big-steps semantics) : $e \twoheadrightarrow v$
2. La sémantique par réduction (small steps semantics) $e \to e_1 \to e_2 \to \ldots \to v$

On applique ça à Mini-ML (qui est Turing-Complet).

2.3.1 Sémantique Opérationnelle à Grand Pas

On cherche à définir $e \twoheadrightarrow v$. On définit les valeurs parmi les constantes, les primitives non appliquées, les fonctions et les paires.
Une relation peut être définie comme la plus petite relation satisfaisant un ensemble d'axiomes notés $\overline{P}$ et des règles d'inférences (implications). On définit $\mathsf{Pair(n)}$ par $\overline{Pair(0)}$ et $\frac{Pair(n)}{Pair(n+2)}$. La plus petitre relation qui vérifie ces deux propriétés coïncide avec la propriété « n est un entier naturel pair ».
Une dérivation est un arbre dont les noeuds correspondent aux règles et les feuilles aux axiomes (les arbres croissent vers le haut). L'ensemble des dérivations possibles caractérise exactement la plus petite relation satisfaisant les règles d'inférence.

Définition 2.3.1. *On définit les variables libres d'une expression e, noté $fv(e)$ par récurrence sur*

e avec :

$$\begin{array}{rcl} fv(x) & = & \{x\} \\ fv(c) & = & \varnothing \\ fv(op) & = & \varnothing \\ fv(\mathsf{fun}\ x \to e) & = & fv(e) \setminus \{x\} \\ fv(e_1 e_2) & = & fv(e_1) \cup fv(e_2) \\ fv((e_1, e_2)) & = & fv(e_1) \cup fv(e_2) \\ fv(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\ fv(x) & = & \{x\} \end{array}$$

On défnit la substitution de toute occurence libre de x dans e par v définie par : $\begin{array}{rcl} x[x \leftarrow v] & = & v \\ y[x \leftarrow v] & = & y \ si\ y \neq x \end{array}$

On fait le choix d'une stratégie d'appel par valeur, i.e. l'argument est complètement évalué avant l'appel.
On a ici comme axiomes : $\overline{c \twoheadrightarrow c}$, $\overline{op \twoheadrightarrow op}$, $\overline{(\mathsf{fun}\ x \to e) \twoheadrightarrow (\mathsf{fun}\ x \to e)}$ et comme règles d'inférences :

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2 \twoheadrightarrow v_2}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)} \qquad \frac{e_1 \twoheadrightarrow v_1 \quad e_2[x \leftarrow v_1] \twoheadrightarrow v}{let\ x = e_1\ \mathsf{in}\ e_2 \twoheadrightarrow v}$$

et

$$\frac{e_1 \twoheadrightarrow (\mathsf{fun}\ x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

On ajoute ensuite des règles pour les primitives, dépendant de la forme de chacune, e.g. :

$$\frac{e_1 \twoheadrightarrow +\quad e_2 \twoheadrightarrow (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \twoheadrightarrow n}$$

Partant, on peut montrer qu'un programme s'évalue en une valeur en écrivant l'arbre de dérivation de celui-ci.

Remarque 2.3.0.1. *Il existe des expressions sans valeur : $e = 12$ par exemple.*

On peut établir une propriété d'une relation définie par un ensemble de règles d'inférence, en raisonnant par induction sur la dérivation. Cela signifie par récurrence structurelle.

Proposition 2.3.1. *Si $e \twoheadrightarrow v$, alors v est valeur. De plus si e est close alors v l'est également.*

Démonstration. Par induction : $\frac{e_1 \twoheadrightarrow (\mathsf{fun}\ x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1 e_2 \twoheadrightarrow v}$. ■

Proposition 2.3.2. *Si $e \twoheadrightarrow v$ et $e \twoheadrightarrow v'$, alors $v = v'$.*

Démonstration. Par induction. ■

Remarque 2.3.0.2. *On a donc définit une fonction plus qu'une relation.*

2.3.2 Sémantique à Petits Pas

La sémantique opérationnelle à petits pas remédie aux problèmes de programmes qui ne terminent pas, en introduisant une notion d'étape élémentaire de calcul $e_1 \to e_2$
On commence par définir une relation $\to^{\varepsilon}$ correspondant à une réduction en tête, au sommet de l'expression, par exemple : $(\mathsf{fun}\ x \to e)\ v \to^{\varepsilon} e[x \leftarrow v]$ On se donne également des règles pour les primitives. On réduit en profondeur en introduisant la règle d'inférence : $\frac{e_1 \to^{\varepsilon} e_2}{E(e_1) \to E(e_2)}$ où E est un contexte défini par la grammaire suivante :

$$\begin{array}{rcl} E & ::= & \square \\ & | & Ee \\ & | & vE \\ & | & \mathsf{let}\ x = E\ \mathsf{in}\ e \\ & | & (E, e) \\ & | & (v, E) \end{array}$$

Un Contexte est un terme à trou où $\square$ représente le trou. $E(e)$ dénote le contexte E dans lequel $\square$ a été remplacé par e. La règle d'inférence permet donc d'évaluer une sous-expression. Tels que définis, les contextes impliquent ici une évaluation en appel par valeur et de gauche à droite.
On note $\to^*$ la cloture réflexive et transitive de $\to$.

Définition 2.3.2. *On appelle forme normale toute expression e telle qu'il n'existe pas* $e^{'}$ *telle que :* $e \to e^{'}$

2.3.3 Equivalence des Sémantiques

Théorème 2.3.1. *Les deux sémantiques opérationnelles sont équivalentes pour les expressions dont l'évaluation termine sur une valeur i.e. :*

$$e \twoheadrightarrow v \Leftrightarrow e \to^* v$$

Démonstration.

Lemme 2.3.2 (Passage au contexte des réductions)**.** *Supposons* $e \to e'$*, alors :*

1. $ee_2 \to e^{'}e_2$
2. $ve \to ve^{'}$
3. $\mathsf{let}\ x = e\ \mathsf{in}\ e_2 \to \mathsf{let}\ x = e'\ \mathsf{in}\ e_2$

— ($\Rightarrow$) On procède par induction sur la dérivation.

— ($\Leftarrow$) :

Lemme 2.3.3 (Evaluation des Valeurs)**.** *On a* $v \twoheadrightarrow v$*.*

Lemme 2.3.4. *Si* $e \to e'$ *et* $e' \twoheadrightarrow v$ *alors* $e \twoheadrightarrow v$*.*

Démonstration. On commence par les réductions de tête, puis on procède par induction aux applications de contexte. ■

On a alors, par récurrence sur le nombre de pas, l'implication souhaitée.

■

2.3.4 Langages Impératifs

Pour un langage impératif les sémantiques ci-dessus sont insuffisantes. On associe alors typiquement un état S à l'expression évaluée. L'état peut être décomposé en plusieurs éléments pour modéliser par exemple une pile (des variables locales), des tas...

3 Interprète

On peut programmer un interprète en suivant les règles de la sémantique naturelle. On se donne un type pour la syntaxe abstraite des expressions et on définit une fonction correspondant à la relation $\twoheadrightarrow$
Un interprète renvoie la (ou les) valeur(s) d'une expression, souvent récursivement.
On peut éviter l'opération de substitution, en interprétant l'expression e à l'aide d'un environnement donnant la valeur courante de chaque variable (un dictionnaire). Ceci pose problème car le résultat de $\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{fun}\ y \to +(x, y)$ est une fonction qui doit « mémoriser »que $x = 1$.
On utilise alors le module Map pour les environnements (c'est une *fermeture*). On représente alors la valeur d'une fonction avec son environnement.
Pour un interprète de la sémantique à petits pas, il vaut mieux utiliser un *zipper* que de recalculer le contexte tout le temps.