Langage de Programmation et Compilation

Jean-Cristophe Filliâtre

10 janvier 2024

Table des matières

Première partie

Cours 1 29/09

1 Introduction à la Compilation

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

1.1 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.

1.2 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*

1.3 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.

2 L'assembleur

2.1 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).

2.2 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*.

2.3 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).

2.4 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`.

2.5 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.

3 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.

3.1 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.

3.2 Sémantiques Useless

3.2.1 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

3.2.2 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.

3.2.3 Sémantique Par Traduction

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

3.3 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).

3.3.1 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 `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 3.3.1. *On définit les variables libres d'une expression e, noté $fv(e)$ par récurrence sur e avec :*

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

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

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 : $\overline{c \twoheadrightarrow c}$, $\overline{op \twoheadrightarrow op}$, $\overline{(\texttt{fun } x \to e) \twoheadrightarrow (\texttt{fun } x \to e)}$ et comme règles d'inférences :

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

et

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

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 3.3.0.1. *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 3.3.1. *Si $e \twoheadrightarrow v$, alors v est valeur. De plus si e est close alors v l'est également.*

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

Proposition 3.3.2. *Si $e \twoheadrightarrow v$ et $e \twoheadrightarrow v'$, alors $v = v'$.*

Démonstration. Par induction. ■

Remarque 3.3.0.2. *On a donc définit une fonction plus qu'une relation.*

3.3.2 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 : $(\texttt{fun}\ x \to e)\ v \to^\varepsilon e[x \leftarrow v]$ 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 :

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

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 3.3.2. *On appelle forme normale toute expression e telle qu'il n'existe pas e' telle que : $e \to e'$*

3.3.3 Equivalence des Sémantiques

Théorème 3.3.1. *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 3.3.2 (Passage au contexte des réductions)**.** *Supposons $e \to e'$, alors :*

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

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

Lemme 3.3.3 (Evaluation des Valeurs)**.** *On a $v \twoheadrightarrow v$.*

Lemme 3.3.4. *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.

■

3.3.4 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...

4 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 `let` $x = 1$ `in 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.

Troisième partie

Cours 3 : 13/10

Introduction

L'objectif est de partir du code source, une suite de caractère, pour obtenir une suite de lexèmes plus compréhensible et simple à analyse syntaxiquement

5 Blancs

Les blancs (espace, retour chariot, tabulation) jouent un rôle dans l'analyse lexicale, car ils permettent de séparer deux lexèmes. De nombreux blancs sont inutiles e.g. `x + 1`, et seront ignorés. Les conventions diffèrent selon les langages et certains des caractères blancs peuvent être significatifs, par exemple l'indentation en python ou en Haskell, ou les retours chariots transformés en points-virgules comme en python. Les commentaires jouent le rôle de blancs.

6 Outils pour l'analyse lexicale

On va utiliser des expressions régulières pour décrire les lexèmes et des automates finis pour les reconnaître. On exploite en particulier la capacité à construire un automate partant d'une ou plusieurs expressions régulières.

6.1 Expressions Régulières et Automates Finis [1]

6.1.1 Expressions Régulières

Définition 6.1.1 (Syntaxe)**.** *On définit la syntaxe des expressions régulières :*

$$\begin{array}{rl} r = & \varnothing \\ | & \varepsilon \\ | & a \in \Sigma \\ | & r \cdot r \\ | & r + r \\ | & r\star \end{array}$$

Définition 6.1.2 (Sémantique)**.** *On définit alors la sémantique basée sur cette syntaxe par les langages rationnels :*

$$\begin{array}{rcl} L(\varnothing) & = & \varnothing \\ L(a) & = & \{a\} \\ L(r_1 r_2) & = & L(r_1) \cdot L(r_2) \\ L(r_1 + r_2) & = & L(r_1) \cup L(r_2) \\ L(r\star) & = & \bigcup_{n \in \mathbb{N}} L^n(r) \end{array}$$

Pour les constantes flottantes de CamL on a par exemple, si $d = 0|1|\ldots|9$:

$$d\ d\star\ (.d\star \mid (\varepsilon \mid .d\star)(e \mid E)(\varepsilon \mid + \mid -)\ d\ d\star)$$

On peut alors écrire un algorithme pour savoir si une suite de caractère appartient à une regexp.

Définition 6.1.3 (Dérivée de Brzozowski)**.** *On pose, pour r une regexp et $c \in \Sigma$: $\delta(r, c) = \{w \mid cw \in L(r)\}$*

1. Voir Cours de LFCC

6.1.2 Automates Finis

Définition 6.1.4 (Syntaxe)**.** *Un automate fini est un quintuplet* (Q, Σ, I, F, T) *où :*

- Q *est un ensemble fini d'états*
- Σ *est un ensemble fini appelé alphabets*
- $I \subseteq Q$ *est un ensemble d'états initiaux*
- $F \subseteq Q$ *est un ensemble d'états finaux*
- $T \subseteq Q \times \Sigma \times Q$ *est un ensemble de transitions*

Théorème 6.1.1 (De Kleene)**.** *Les expressions régulières et les automates finis définissent les mêmes langages.*

Démonstration. Voir Cours de LFCC ■

6.2 Analyseur Lexical

6.2.1 Principe

Un analyseur lexical est un automate fini pour la réunion de toutes les expressions régulières définissant les lexèmes. Le fonctionnement de l'analyseur lexical est différent de la simple reconnaissance d'un mot par un automate car :

- Il faut décomposer un mot (le source) en une suite de mots reconnus
- Il peut y avoir des ambiguïtés
- Il faut construire les lexèmes (les états finaux contiennent des actions)

Ambiguïtés

Le mot `funx` est reconnu par l'expression régulière des identificateurs mais contient un préfixe reconnu par une autre expression régulière (`fun`) : ⇒ On choisit de reconnaître le lexème le plus long possible.
Le mot `fun` est reconnu par la regexp du mot clef `fun` mais aussi par celle des identificateurs : ⇒ On classe les lexèmes par ordre de priorité.

Retour en arrière

Un analyseur va échouer sur l'entrée abc avec les trois regexp a, ab, bc. L'analyseur lexical doit donc mémoriser le dernier état final rencontré, le cas échéant.

Lorqu'il n'y a plus de transition possible dans l'automate, de deux choses l'une :

- Soit aucun état final mémorisé : échec de l'analyse lexicale
- Soit on a lu le préfixe wv de l'entrée, avec w le lexème reconnu par le dernier état final recontré : on renvoie w et l'analyse lexicale redémarre avec v préfixé au reste de l'entrée

En pratique, on va renvoyer dans l'analyseur lexical une fonction de calcul du prochain lexème, puisque l'analyse lexicale est faite pour l'analyse syntaxique, cf. **??**

6.2.2 Construction

L'automate de Thompson

On construit par induction un automate compatible.

L'automate de Berry-Sethi

On met en correspondance les lettres d'un mot reconnu et celles apparaissant dans la regexp : On distingue les différentes lettres de la regexp puis on construit un automate dont les états sont des ensembles de lettres. Pour construire les transitions de s_1 à s_2, on détermine les lettres qui peuvent apparaître après une autre dans un mot reconnu : `follow`. Pour calculer `follow`, on a besoin de savoir calculer les premières et dernières lettres d'un mot reconnu (`first` et `last`). On a alors besoin d'une dernière notion : `null`, est-ce que ε appartient au langage reconnu. On obtient :

$\texttt{null}(\varnothing)$	$=$	`false`
$\texttt{null}(\varepsilon)$	$=$	`true`
$\texttt{null}(a)$	$=$	`false`
$\texttt{null}(r_1 r_2)$	$=$	$\texttt{null}(r_1) \wedge \texttt{null}(r_2)$
$\texttt{null}(r_1 \mid r_2)$	$=$	$\texttt{null}(r_1) \vee \texttt{null}(r_2)$
$\texttt{null}(r\star)$	$=$	`true`
On en déduit :		
$\texttt{first}(\varnothing)$	$=$	$\varnothing$
$\texttt{first}(\varepsilon)$	$=$	$\varnothing$
$\texttt{first}(a)$	$=$	$\{a\}$
$\texttt{first}(r_1 r_2)$	$=$	$\texttt{first}(r_1) \cup \texttt{first}(r_2)$ si $\texttt{null}(r_1)$
	$=$	$\texttt{first}(r_1)$ sinon
$\texttt{first}(r_1 \mid r_2)$	$=$	$\texttt{first}(r_1) \cup \texttt{first}(r_2)$
$\texttt{first}(r\star)$	$=$	$\texttt{first}(r)$
On définit `last` de même		
$\texttt{follow}(c, \varnothing)$	$=$	$\varnothing$
$\texttt{follow}(c, \varepsilon)$	$=$	$\varnothing$
$\texttt{follow}(c, a)$	$=$	$\varnothing$
$\texttt{follow}(c, r_1 r_2)$	$=$	$\texttt{follow}(c, r_1) \cup \texttt{follow}(c, r_2) \cup \texttt{first}(r_2)$ si $c \in \texttt{last}(r_1)$
	$=$	$\texttt{follow}(c, r_1) \cup \texttt{follow}(c, r_2)$ sinon
$\texttt{follow}(c, r_1 \mid r_2)$	$=$	$\texttt{follow}(c, r_1) \cup \texttt{follow}(c, r_2)$
$\texttt{follow}(c, r\star)$	$=$	$\texttt{follow}(c, r) \cup \texttt{first}(r)$ si $c \in \texttt{last}(r)$
	$=$	$\texttt{follow}(c, r)$ sinon

On construit alors l'automate reconnaissant r en ajoutant # à la fin de r :

1. L'état initial est l'ensemble $\texttt{first}(r\#)$.
2. Tant qu'il existe un état s dont on doit calculer les transitions, pour chaque c de l'alphabet, on pose s' l'état $\bigcup_{c_i \in s} \texttt{follow}(c_i, r\#)$
3. Les états acceptants sont ceux contenant #

7 L'outil `ocamllex`

7.1 Analyseur Lexical en `ocamllex`

Un fichier `ocamllex` porte le suffixe `.mll` et a la forme suivante :

— Code OCamL arbitraire

— `rule` $f_1 =$ `parse` | `regexp1` {`action 1`}

— ...

— Code OcamL arbitraire

A la compilation par `ocamllex file.mll`, on construit un fichier OCamL contenant des fonctions de types `Lexing.lexbuf` → `type` où le type de sortie dépend de l'action dans la fonction.

Les regexp en `ocamllex` s'écrivent sous la forme :

_	n'importe quel caractère
'a'	le caractère `'a'`
"foobar"	la chaîne `"foobar"` (en particulier ε = "")
[caractères]	ensemble de caractères
[^caractères]	complémentaire d'un ensemble de caractères
$r_1 \mid r_2$	alternative
$r_1 r_2$	concaténation
$r*$	étoile
$r+$	une ou plusieurs occurences de r, i.e. $rr\star$
$r?$	au plus une occurence de r
eof	fin du fichier

Pour remplacer la reconnaissance du lexème le plus long par celui le plus court, remplacer `parse` par `shortest`.
A longueur égale, c'est la règle qui apparaît en premier qui l'emporte.
On peut nommer la chaîne reconnue, ou des sous-chaînes reconnues par des sous-experssions régulières, à l'aide de la construction `as`.
On peut dans une action, rappeler récursivement l'analyseur lexical ou l'un des autres analyseurs simultanément définis. Le tampon d'analyse lexical doit être passé en argument, il est contenu dans une variable appelée `lexbuf`. Ceci est utile pour séparer les blancs, ou reconnaître les commentaires, même imbriqués.

Par défaut `ocamllex` contruit l'automate dans une table interprétée à l'exécution, mais l'option `-ml` construit du code CamL pur.

7.2 Efficacité

Pour des raisons de stockage, et même en utilisant une table, l'automate peut prendre beaucoup de place. Il est donc préférable d'utiliser une seule expression régulière pour les identificateurs et les mots-clefs, puis de les séparer ensuite grâce à une table des mots-clefs.
On peut de même ne stocker que les caractères en minuscule pour être insensible à la casse.

7.3 D'autres utilisations `ocamllex`

On peut utiliser `ocamllex` pour :

1. Réunir plusieurs lignes vides consécutives en une seule
2. Compter les occurences d'un mot dans un texte
3. Un petit traducteur OCamL vers HTML pour embellir le source mis en ligne (mettre les mots-clefs en vert, les commentaires en rouge, numéroter les lignes ...), le tout en moins de 100 lignes de code.

Quatrième partie

Cours 4 : 20/10

8 Analyse Syntaxique

L'objectif de l'analyse syntaxique est, à partir d'une suite de lexèmes, de construire la syntaxe abstraite, qu'on représente sous forme d'arbre. En particulier, l'analyse syntaxique doit détecter les erreurs de syntaxe et les localiser précisément, les identifier (parenthèse non fermée, etc...) voire reprendre l'analyse pour découvrir de nouvelles erreurs. On va utiliser : une grammaire non contextuelle pour décrire la syntaxe et un automate à pile pour la reconnaître.

9 Grammaires

Définition 9.0.1. *Une grammaire non contextuelle est un quadruplet (N, T, S, R) où :*

- *N est un ensemble fini de symboles non terminaux.*
- *T est un ensemble fini de symboles terminaux.*
- *$S \in N$ est le symbole de départ (axiome).*
- *$R \subseteq N \times (N \cup T)^*$ est un ensemble fini de règles de production.*

On note les règles de dérivation sous la forme :

$$\begin{aligned} E \quad &\to E + E \\ &\mid E * E \\ &\mid (E) \\ &\mid \texttt{int} \end{aligned}$$

Définition 9.0.2. *Un mot $u \in (N \cup T)^*$ se dérive en un mot $v \in (N \cup T)^*$ et on note $u \to v$ s'il existe une décomposition $u = u_1 X u_2$ avec $X \in N$, $X \to \beta \in R$ et $v = u_1 \beta u_2$. On appelle dérivation gauche le cas où u_1 ne contient pas de mots terminaux. On appelle langage défini par G l'ensemble des mots de T^* dérivés de l'axiome.*

Définition 9.0.3. *Un arbre de dériviation est un arbre dont les noeuds sont étiquetés par des symboles de la grammaire, de la manière suivante :*

- *La racine est l'axiome*
- *Tout noeud interne X non terminal dont les fils sont étiquetés par $\beta \in (N \cup T)^*$ avec $X \to \beta$ une règle de la dérivation.*

Pour un arbre de dérivation dont les feuilles forment le mot w dans l'arbre infixe, on a $S \to^\star w$. Inversement, si $S \to^\star w$, on a un arbre de dérivation dont les feuilles dans l'arbre infixe forment w.

Définition 9.0.4. *Une grammaire est ambigüe si au moins un mot admet au moins deux arbres de dérivation. Déterminer si une grammaire est ambigüe ou non n'est pas décidable.*

La grammaire précédente est ambigüe, comment interpréter `int` + `int` ∗ `int`

On va utiliser des critères décidables suffisants pour garantir qu'une grammaire est non ambigüe, et pour lesquels on sait en outre décider l'appartenance au langage efficacement.

10 Analyse ascendante

On va ici lire l'entrée de gauche à droite (pas toujours le cas, cf. `CYK`) puis reconnaître des membres droits de productions pour construire l'arbre de dérvation de bas en haut : bottom-up parsing.

10.1 Fonctionnement

On va manipuler une pile qui est un mot de $(N \cup T)^*$. A chaque instant, on a deux actions possibles :

— Une opération de lecture (*shift*) : on lit un terminal de l'entrée et on l'empile.

— Une opération de réduction (*reduce*) : on reconnaît en sommet de pile le membre droit β d'une production $X \to \beta$ et on remplace β par X en sommet de pile.

Comment prendre la décision lecture / réduction ? On se sert d'un automate fini et on examine les k premiers lexèmes de l'entrée, c'est l'analyse `LR(k)`, pour « Left to right scanning, Righmost derivation ». En pratique, $k = 1$.

10.2 Analyse LR

La pile est de la forme $s_0 x_1 s_1 x_2 \dots x_n s_n$ où s_i est un état de l'automate et $x_i \in T \cup N$ comme dans un automate. Soit a le premier lexème de l'entrée. Une table indexée par s_n et a nous indique l'action à effectuer :

— Si c'est un succès ou un échec, on s'arrête.

— Si c'est une lecture, on empile a et l'état s résultat de la transition $s_n \to^a s$ dans l'automate.

— Si c'est une réduction $X \to \alpha$, avec α de longueur p, on doit trouver α en sommet de pile :

$$s_0 x_1 \dots x_{n-p} s_{n-p} \mid \alpha_1 s_{n-p+1} \dots \alpha_p s_n$$

On dépile alors α et on empile Xs où $s_{n-p} \to^X s$ dans l'automate, i.e. $s_0 x_1 \dots x_{n-p} s_{n-p} X s$

En pratique on ne travaille pas avec l'automate mais avec deux tables :

— Une table d'actions ayant pour lignes les états et pour colonnes les terminaux. La case `action(s, a)` indique :

  — `shift s'` pour une lecture et un nouvel état s'

  — `reduce X` $\to \alpha$ pour une réduction

  — un succès

  — un échec

  Une table de déplacements ayant pour lignes les états et pour colonnes les non terminaux. La case `goto(s, X)` indique l'état résultat d'une réduction de X.

On ajoute aussi un lexème spécial # qui désigne la fin de l'entrée. On peut le voir comme l'ajout d'un nouveau non terminal S qui devient l'axiome et d'une règle $S \to E\#$. Pour créer ces tables, on utilise la famille de `yacc` (*Yet Another Compiler Compiler*)

11 L'outil Menhir

Menhir transforme une grammaire en un analyseur OCamL de type $LR(1)$. Chaque production de la grammaire est accompagnée d'une action sémantique, i.e. du code OCamL construisant une valeur sémantique (typiquement un arbre de syntaxe abstraite). Menhir s'utilise conjointement avec un analyseur lexical (typiquement ocamllex).
Un fichier Menhir a le suffixe `.mly` et a la forme suivante :

— %`code OCamL arbitraire` %

— `déclaration des lexèmes`

— %%

— `non-terminal-1 :` | `production` { `action` } | `production` { `action` } ;

— `non-terminal-2 :` | `production` { `action` }

— %%

— `code OCamL arbitraire`

En cas de conflits, Menhir produit deux fichiers pour expliquer d'où les conflits viennent. On peut les résoudre est d'indiquer comment choisir entre lecture et réduction. On peut donner des priorités aux lexèmes et aux productions, en fonction des règles d'associativité. Si la priorité de la production est supérieure à celle du lexème à lire, la réduction est favorisée. En cas d'égalité, l'associativité est consultée : un lexème associatif à gauche favorise la réduction et un lexème associatif à droite favorise la lecture. Par ailleurs, pour empêcher l'associativité, on peut aussi l'indiquer à Menhir, et éviter le *dangling else*.

Pour que les phases suivantes de l'analyse (typiquement le typage) puissent localiser les messages d'erreur, il convient de conserver une information de localisation dans l'arbre de syntaxe abstraite. Menhir fournit cette information dans $\$startpos$ et $\$endpos$, deux valeurs du type `Lexing.position`. Cette information lui a été transmise par l'analysur lexical. (`ocamllex` ne maintient automatiquement que la position absolue dans le fichier, il faut appeler Lexing.new_line pour chaque retour chariot.)

12 Derrière l'outil Menhir

12.1 First, Null, Follow

Définition 12.1.1 (Null). *Soit $\alpha \in (T \cup N)^*$. $\textsc{Null}(\alpha)$ est vrai si et seulement si on peut dériver ε à partir de α.*

Définition 12.1.2 (First). *Soit $\alpha \in (T \cup N)^*$. $\textsc{First}(\alpha)$ est l'ensemble de tous les premiers terminaux des mots dérivés de α, i.e. $\{a \in T \mid \exists\ w,\ \alpha \to^\star aw\}$*

Définition 12.1.3 (Follow). *Soit $X \in N$. $\textsc{Follow}(X)$ est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation, i.e. $\{a \in T \mid \exists\ u,\ w\ S \to^\star uXaw\}$.*

Théorème 12.1.1 (Tarski). *Soit A un ensemble fini muni d'une relation d'ordre $\leq$ et d'un plus petit élément ε. Toute fonction $f : A \to A$ croissante admet un plus petit point fixe.*

Démonstration. Comme ε est le plus petit élément, $\varepsilon \leq f(\varepsilon)$. Par croissance, $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$ pour tout k. Mais A étant fini, il existe un plus petit k_0 tel que $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$. En le notant a_0, on a bien un point fixe de f. Si b est un autre point fixe, par croissance, puisque $\varepsilon \leq b$, on a bien le résultat. ■

12.1.1 Principe du calcul de Null(X)

Proposition 12.1.1. *Pour calculer $\textsc{Null}(\alpha)$, il suffit de déterminer $\textsc{Null}(X)$ pour $X \in N$. On a $\textsc{Null}(X)$ ssi :*

- *Il existe une production $X \to \varepsilon$*
- *Il existe une production $X \to Y_1 \dots Y_m$ où $\textsc{Null}(Y_i)$ pour tout i.*

Il s'agit d'un ensemble d'équations mutuellement récursives. Autrement dit, on cherche la plus petite solution d'une équation de la forme : $\vec{V} = F(\vec{V})$.

Démonstration.
- $\Rightarrow$ Par récurrence sur le nombre d'étapes du calcul de point fixe, on montre que si $\textsc{Null}(X)$ alors $X \to^\star \varepsilon$
- $\Leftarrow$ Par récurrence sur le nombre d'étapes de la dérivation, on montre la réciproque.

■

Ici, on a $A = \{0,1\}^n$. On munit $\{0,1\}$ de l'ordre $0 \leq 1$ et A de l'ordre point à point. On a $\varepsilon = (0, \dots, 0)$. La fonction calculant $\textsc{Null}(X)$ à partir des $\textsc{Null}(X_i)$ est croissante, et le théorème de Tarski s'applique. On construit donc un point fixe à partir de ε.

12.1.2 Principe du calcul de First(X)

De même que pour Null : Les équations définissant First sont mutuellement récursives :

$$\textsc{First}(X) = \bigcup_{X \to \beta} \textsc{First}(\beta)$$

et :

$$\begin{aligned}
\textsc{First}(\varepsilon) &= \varnothing \\
\textsc{First}(a\beta) &= \{a\} \\
\textsc{First}(X\beta) &= \textsc{First}(X) \text{ si } \neg\textsc{Null}(X) \\
\textsc{First}(X\beta) &= \textsc{First}(X) \cup \textsc{First}(\beta) \text{ si } \textsc{Null}(X)
\end{aligned}$$

On applique alors le calcul de point fixe sur $A = \mathcal{P}(T)^n$ muni de $\subseteq$ point à point avec $\varepsilon = (\varnothing, \ldots, \varnothing)$

12.1.3 Principe du calcul de Follow(X)

Les équations sont :

$$\textsc{Follow}(X) = \left(\bigcup_{Y \to \alpha X \beta} \textsc{First}(\beta) \right) \cup \left(\bigcup_{Y \to \alpha X \beta, \textsc{Null}(\beta)} \textsc{Follow}(Y) \right)$$

On procède par calcul de point fixe sur le même domaine que pour First.

12.2 Automate LR

12.2.1 $LR(0)$

Fixons pour l'instant $k = 0$. On commence par contruire un automate asynchrone :

— Les états sont des items de la forme $[X \to \alpha \cdot \beta]$ où $X \to \alpha\beta$ est une production de la grammaire. L'intuition est : « je cherche à reconnaître X, j'ai déjà lu α et je dois encore lire β ».

— Les transitions sont étiquetées par $T \cup N$ et sont les suivantes :

$$\begin{aligned}
&[Y \to \alpha \cdot a\beta] \to^a [Y \to \alpha a \cdot \beta] \\
&[Y \to \alpha \cdot X\beta] \to^X [Y \to \alpha X \cdot \beta] \\
&[Y \to \alpha \cdot X\beta] \to^\varepsilon [X \to \cdot\gamma] \text{ pour toute production } X \to \gamma
\end{aligned}$$

On déterminise ensuite l'automate en regroupant les états reliés par des ε-transitions. Les états de l'automate déterministe sont donc des ensembles d'items.
Par construction : chaque état s est saturé par la propriété : si $Y \to \alpha \cdot X\beta \in s$ et si $X \to \gamma$ est une production, alors $X \to \cdot\gamma \in s$. L'état initial est celui contenant $S \to \cdot E\#$.

On construit alors la table `action` :

— $\texttt{action}(s, \#) =$ succès si $[S \to E \cdot \#] \in s$

— $\texttt{action}(s, a) = \texttt{shift}s'$ si $s \to^a s'$

— $\texttt{action}(s, a) = \texttt{reduce}X \to \beta$ si $[X \to \beta\cdot] \in s$ pour tout a.

— `échec` dans tous les autres cas

On construit alors la table `goto` : $\texttt{goto}(s, X) = s'$ si $s \to^X s'$.
La table $LR(0)$ peut contenir deux sortes de conflits : lecture/réduction et réduction/réduction.

Définition 12.2.1. *Une grammaire est dite $LR(0)$ so la table ainsi construite ne contient pas de conflit.*

La construction $LR(0)$ engendre très facilement des conflits. On va chercher à limiter les réductions : on pose $\texttt{action}(s, a) = \texttt{reduce}X\beta$ si et seulement si $[X \to \beta\cdot] \in s$ et $a \in \textsc{Follow}(X)$. On obtient la classe de grammaire $S(imple)LR(1)$.

12.2.2 $LR(1)$

Cette classe étant restrictive, on introduit une classe de grammaires encore plus large : $LR(1)$, au prix de tables encore plus grandes. Dans l'analyse $LR(1)$, les items ont la forme : $[X \to \alpha \cdot \beta, a]$ Les transitions de l'automate $LR(1)$ non déterministe sont :

$$
\begin{aligned}
[Y \to \alpha \cdot a\beta, b] &\to^{a} [Y \to \alpha a \cdot \beta, b] \\
[Y \to \alpha \cdot a\beta, b] &\to^{a} [Y \to \alpha a \cdot \beta, b] \\
[Y \to \alpha \cdot X\beta, b] &\to^{X} [Y \to \alpha X \cdot \beta, b] \\
[Y \to \alpha \cdot X\beta, b] &\to^{\varepsilon} [X \to \cdot\gamma, c] \text{ pour tout } c \in \textsc{First}(\beta b)
\end{aligned}
$$

L'état initial est celui qui contient $[S \to \cdot\alpha, \#]$. On peut déterminiser l'automate et construire la table correspondante : on introduit une action de réduction pour (s, a) seulement lorsque s contient un item de la forme $[X \to \alpha\cdot, a]$.

Pour des questions de puissances de calcul, on introduit la classe $LALR(1)$, lookahead LR, qui est une approximation beaucoup utilisée dans les outils de la famille `yacc`.

Cinquième partie
Cours 5 : 27/10

13 Localisations

L'outil `ocamllex` maintient, dans la structure de type `Lexing.lexbuf`, la position courante dans le texte source qui est analysé. On peut alors obtenir la localisation de la dernère chaîne reconnue par `ocamllex`. On peut utiliser ces informations pour localiser les erreurs de syntaxe, mais aussi de potentielles erreurs lexicales comme une chaîne ou un commentaire non fermé.
L'outil `Menhir` [2] récupère ces informations et les fournit dans deux valeurs `$startpos` et `$endpos`, qui, dans une action sémantique, correspondent au début et à la fin du texte reconnu par la règle de grammaire. On peut alors stocker cette information dans l'arbre de syntaxe abstraite.

14 Analyse Syntaxique Elementaire

On va ici construire un analyseur syntaxique pour des expressions arithmétiques incluant :

— des constantes
— des additions
— des multiplications
— des parenthèses

On part d'un analyseur lexical, par exemple écrit avec `ocamllex` :

```
type token =
    | CONST of int
    | PLUS
    | TIMES
    | LEFTPAR
    | RIGHTPAR
    | EOF
```

et on veut obtenir un arbre de syntaxe abstraite :

```
type expr =
    | Const of int
    | Add of expr * expr
    | Mul of expr * expr
```

On écrit un parser dans le fichier Menhir, puis, juste en dessous dans le fichier, l'analyseur syntaxique.

Remarque 14.0.0.1. *Conseil : Commencer par écrire un pretty-printer, ici, en OcamL avec la bibliothèque `Format` :*

```
open Format
let rec print_expr fmt = function
  | Add (e1, e2) -> fprintf fmt "%a +@ %a" print_expr e1 print_expr e2
  | e            -> print_term fmt e
and print_term fmt = function
  | Mul (e1, e2) -> fprintf fmt "%a *@ %a" print_term e1 print_term e2
  | e            -> print_factor fmt e
and print_factor fmt = function
  | Const n -> fprintf fmt "%d" n
  | e       -> fprintf fmt "(@[%a@])" print_expr e
```

2. L'outil `Cairn` permet de visualiser l'analyse de `Menhir`.

L'analyseur syntaxique suit la même structure que la fonction d'affichage [3]. C'est ça, le bon conseil de Gilles Kahn :

```
let rec parse_expr () =
    let e = parse_term () in
    if !t = PLUS then (next (); Add (e, parse_expr ())) else e
and parse_term () =
    let e = parse_factor () in
    if !t = TIMES then (next (); Mul (e, parse_term ())) else e
and parse_factor () = match !t with
    | CONST n -> next (); Const n
    | LEFTPAR -> next ();
        let e = parse_expr () in
        if !t <> RIGHTPAR then error ();
        next (); e
    | _ -> error ()
```

Remarque 14.0.0.2. *On pourrait inclure l'analyse lexicale dans un tel code, avec d'autres fonctions récursives pour lire les constantes entières, ignorer les blancs, etc...*
Pour des opérateurs associatifs à gauche, le code sera légèrement différent mais le principe reste le même.

15 Analyse Descendante

15.1 Fonctionnement

On va procéder par expansions successives du non-terminal le plus à gauche (dérivation gauche) en partant de S et en utilisant une table qui indique pour un non-terminal X à expanser et les k premiers caractères de l'entrée, l'expansion $X \to \beta$ à effectuer. Dans la suite, on va prendre $k = 1$, et on va noter $T(X, c)$ cette table. En pratique, on suppose qu'un symbole terminal # dénote la fin de l'entrée, et la table indique donc également l'expansion de X lorsqu'on atteint la fin de l'entrée. On utilise une pile qui est un mot de $(N \cup T)^\star$. Initialement, la pile est réduite au symbole de départ. A chaque instant, on va ensuite examiner le sommet de la pile et le premier caractère c de l'entrée :

— Si la pile est vide, on s'arrête ; il y a succès ssi c est #.

— Si le sommet de la pile est un terminal a, alors a doit être égal à c, on dépile alors a et on consomme c ; sinon on échoue.

— Si le sommet de la pile est un non terminal X, on remplace X par le mot $\beta = T(X, c)$ en sommet de pile, en empilant les caractères de β en partant du dernier ; sinon on échoue

15.2 Programmation

Un analyseur descendant se programme en introduisant une fonction pour chaque non terminal de la grammaire. Chaque fonction examine l'entrée, et selon le cas, la consomme ou appelle récursivement les fonctions correspondant à d'autres non terminaux, selon la table d'expansion.

Remarque 15.2.0.1.
— *La table d'expansion n'est alors pas explicite : elle est dans le code de chaque fonction.*
— *La pile n'est pas explicite, elle est réalisée par la pile d'appels.*
— *En pratique il faut construire l'arbre de syntaxe abstraite.*

3. Le code n'est pas complet, cf page du cours

15.3 Construction de la Table d'Expansion

Pour décider si on réalise l'expansion $X \to \beta$ lorsque le premier caractère de l'entrée est c, on va chercher à déterminer si c fait partie des premiers caractères des mots reconnus par β. Une difficulté se pose pour une production telle que $X \to \varepsilon$, et il faut alors considérer l'ensemble des caractères qui peuvent suivre X. On utilise donc à nouveau les fonctions `first` et `follow` :Pour chaque production $X \to \beta$

— On pose $T(X, a) = \beta$ pour tout $a \in \texttt{first}(\beta)$

— Si $\texttt{null}(\beta)$, on pose aussi $T(X, a) = \beta$ pour tout $a \in \texttt{follow}(X)$.

Il peut y avoir plusieurs éléments dans une même case :

Définition 15.3.1. *Une grammaire est dite $LL(1)$ si, dans la table précédente, il y a au plus une production dans chaque case.*

Une grammaire récursive gauche, i.e. contenant des productions de la forme : $X \to X\alpha|\beta$, ne sera jamais $LL(1)$. En effet, les `first` seront les mêmes pour ces deux productions. En particulier, la grammaire : $\begin{array}{rl} E & \to E + T \\ & \to \ T \\ T & \to T * F \\ & \to \ F \\ F & \to \ (E) \\ & \to \ \texttt{int} \end{array}$ n'est pas $LL(1)$. Plus généralement, si une grammaire contient $X \to a\alpha + a\beta$. Il faut alors factoriser (à gauche) les productions qui commencent par le même terminal.

16 Indentation comme Syntaxe

Dans certains langages, l'indentation (blancs de début de ligne/alignement vertical) est utilisée pour définir la syntaxe. L'analyseur lexical introduit des lexèmes `NEWLINE` (fin de ligne), `INDENT` (quand l'indentation augmente) et `DEDENT` (quand elle diminue). Il suffit alors de les utiliser dans la grammaire du langage.

Sixième partie

Cours 6 : 10/11

17 Typage

Que faire lorsqu'on ajoute deux éléments de deux types différents ? Par exemple "5" et 37. Si on additionne deux expressions arbitraires, $e1 + e2$, comment déterminer si cela est légal, et que doit on faire le cas échéant ? La réponse est le typage, une analyse qui associe un type à chaque valeur/expression dans le but de rejeter les programmes incohérents. Certains langages sont typés dynamiquement, les types sont associés aux valeurs et utilisés pendant l'exécution, et d'autres sont typés statiquement, les types sont associés aux expressions et utilisés pendant la compilation du programme.

> Well typed programs cannot go wrong - Milner

17.1 Objectifs du Typage

Le typage doit :

— être décidable

— rejeter les programmes absurdes dont l'évaluation échouerait, c'est la sûreté du typage

— pas rejeter trop de programmes non-absurdes, i.e. le système de types doit être expressifs.

On peut par exemple :

— Annoter toutes les sous-expressions par un type

— Annoter seulement les déclarations de variables

— Annoter seulement les paramètres de fonctions

— Ne rien annoter et tout inférer.

Un algorithme de typage doit avoir les propriétés de :

— Correction : si l'algorithme répond "oui", alors le programme est effectivement bien typé

— Complétude : si le programme est bien typé, alors l'algorithme doit répondre "oui"

— Principalité (pas obligatoire) : le type calculé pour une expression est le plus général possible

18 Typage Monomorphe

18.1 Définitions

On se donne des types simples, dont la syntaxe abstraite est :

$$\begin{array}{rl} \tau | & \texttt{int} \mid \texttt{bool} \mid \ldots \\ \mid & \tau \to \tau \\ \mid & \tau \times \tau \end{array}$$

Le jugement de typage se note : $\Gamma \vdash e : \tau$ et se lit : *dans l'environnement* Γ, *l'expression* e *a le type* τ. On posssède alors des règles d'inférences :

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{}{\Gamma \vdash n : \texttt{int}} \quad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$\Gamma + x : \tau$ est l'environnement Γ' défini par $\Gamma'(x) = \tau$ et $\Gamma'(y) = \Gamma(y)$ sinon.

En revanche, on ne peut pas typer le programme `1 2`, ni le programme `fun` $x \to x\ x$ car $\tau_1 = \tau_1 \to \tau_2$ n'a pas de solution, les types étant finis. On peut montrer $\varnothing \vdash$ `fun` $x \to x$: `int` $\to$ `int` mais aussi $\varnothing \vdash$ `fun` $x \to x$: `bool` $\to$ `bool`. Ce n'est tout de même pas du polymorphisme, pour une occurence donnée de `fun` $x \to x$, il faut choisir un type. Ainsi, le terme `let` f `in fun` $x \to x(f\ 1, f$ `true`$)$ n'est pas typable car il n'y a pas de type τ tel que : $f : \tau \to \tau \vdash (f\ 1, f$ `true`$) : \tau_1 \times \tau_2$. En particulier, on ne peut pas donner un type satisfaisant à une primitive, mais on peut donner une règle de typage pour l'application de celle-ci :

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash fste : \tau_1}$$

ou

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \texttt{opfix}e : \tau}$$

Si on souhaite se limiter à des fonctions, on peut modifier ainsi :

$$\frac{\Gamma \vdash e : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)}{\Gamma \vdash \texttt{opfix}\ e : \tau_1 \to \tau_2}$$

Quand on type `fun` $x \to e$, comment trouve-t-on le type à donner à x ? C'est toute la différence entre les règles de typage, qui définissent la relation ternaire $\Gamma \vdash e : \tau$ et l'algorithme de typage qui vérifie qu'une certaine expression e est bien typée dans un certain environnement Γ.

18.2 Implémentation sur Mini-ML

On définit une syntaxes abstraite des types, et donc dans le type des expressions, on demande que le constructeur `Fun` prenne également le type de l'entrée comme argument. L'environnement Γ est réalisé par une structure persistante, par exemple le module Map d'OCamL. (Insertion et recherche en $\mathcal{O}(\log n)$).

```
type typ =
    | Tint
    | Tarrow of typ * typ
    | Tproduct of typ * typ

type expression =
    | Var of string
    | Const of int
    | Op of string
    | Fun of string * typ * expression (* seul changement *)
    | App of expression * expression
    | Pair of expression * expression
    | Let of string * expression * expression
```

On peut alors passer à l'inférence de la vérification des types.

```
let rec type_expr env = function
    | Const _ -> Tint
    | Var x -> Smap.find x env
    | Op "+" -> Tarrow (Tproduct (Tint, Tint), Tint)
    | Pair (e1, e2) ->
       Tproduct (type_expr env e1, type_expr env e2)
    | Fun (x, ty, e) ->
       Tarrow (ty, type_expr (Smap.add x ty env) e)
    | Let (x, e1, e2) ->
       type_expr (Smap.add x (type_expr env e1) env) e2
    | App (e1, e2) -> begin match type_expr env e1 with
       | Tarrow (ty2, ty) ->
```

```
        if type_expr env e2 = ty2 then ty
        else failwith "erreur : argument de mauvais type"
    | _ ->
        failwith "erreur : fonction attendue"
    end
```

Les seules vérifications se trouvent dans la partie sur l'application de fonctions.
On ne renvoie pas un `failwith "erreur de typage"`, mais on indique l'origine du problème avec précision et on conserve les types pour les phases ultérieures du compilateur :

— D'une part, on ajoute aux noeuds des arbres en entrée du typage une localisation dans le fichier source :

```
type expression = {
    desc : desc;
    loc : loc;
}
and desc =
    | Var of string
    | Const of int
    | Op of string
    | Fun of string * typ * expression (* seul changement *)
    | App of expression * expression
    | Pair of expression * expression
    | Let of string * expression * expression


exception Error of loc * string
```

On peut la lever en modifiant le rendu en sortie :

```
let rec type_expr env e = match e.desc with
    | ...
    | App (e1, e2) -> begin match type_expr env e1 with
        | Tarrow (ty2, ty) ->
            if type_expr env e2 <> ty2 then
                raise (Error (e2.loc, "argument de mauvais type"));
            ...

try
    let p = Parser.parse file in
    let t = Typing.program p in
    ...
with Error (loc, msg) ->
    Format.eprintf "File '%s', line ...\n" file loc;
    Format.eprintf "error: %s@." msg;
    exit 1
```

— D'autre part, on décore les arbres en sortie du typage avec des types :

```
type texpression = {
    tdesc: tdesc;
    typ : typ;
    }
and tdesc =
    | Tvar of string
    | Tconst of int
    | Top of string
    | Tfun of string * typ * texpression
    | Tapp of texpression * texpression
```

```
    | Tpair of texpression * texpression
    | Tlet of string * texpression * texpression
```

C'est un *autre* type d'expressions.

La fonction de typage reconstruit des arbres, cette fois typés :

```
let rec type_expr env e =
    let d, ty = compute_type env e in
    { tdesc = d; typ = ty }

and compute_type env e = match e.desc with
    | Const n ->
        Tconst n, Tint
    | Var x ->
        Tvar x, Smap.find x env
    | Pair (e1, e2) ->
        let te1 = type_expr env e1 in
        let te2 = type_expr env e2 in
        Tpair (te1, te2), Tproduct (te1.typ, te2.typ)
    | ...
```

18.3 Sûreté du typage

Lemme 18.3.1 (Progression). *Si* $\varnothing \vdash e : \tau$ *alors soit* e *est une valeur, soit il existe* $e^{'}$ *tel que* $e \to e^{'}$.

Démonstration. On procède par récurrence sur la dérivation $\varnothing \vdash e : \tau$.
Supposons par exemple $e = e_1\ e_2$:

$$\frac{\varnothing \vdash e_1 : \tau^{'} \to \tau \quad \varnothing \vdash e_2 : \tau^{'}}{\varnothing \vdash e_1\ e_2 : \tau}$$

On applique l'HR à e_1 :

— Si $e_1 \to e_1^{'}$ alors $e_1\ e_2 \to e_1^{'} e_2$.

— Si e_1

■

Lemme 18.3.2 (Permutation). *Si* $\Gamma + x : \tau_1 + y : \tau_2 \vdash e : \tau$ *et* $x \neq y$ *alors* $\Gamma + y : \tau_2 + x : \tau_1 \vdash e : \tau$ *et la dérivation a même hauteur.*

Démonstration. Récurrence immédiate ■

Lemme 18.3.3 (Affaiblissement). *Si* $\Gamma e : \tau$ *et* $x \notin dom(\Gamma)$ *alors* $\Gamma + x : \tau^{'} \vdash e : \tau$ *et la dérivation a même hauteur.*

Démonstration. Récurrence Immédiate ■

Lemme 18.3.4 (Préservation par Substitution). *Si* $\Gamma + x : \tau^{'} \vdash e : \tau$ *et* $\Gamma \vdash e^{'} : \tau^{'}$ *alors* $\Gamma \vdash e\left[x \leftarrow e^{'}\right] : \tau$

Démonstration. Preuve par récurrence sur la hauteur de la dérivation $\Gamma + x : \tau^{'} \vdash e : \tau$:

— Cas d'une variable $e = y$

  — Si $x = y$ alors $e\left[x \leftarrow e^{'}\right] = e^{'}$ et $\tau = \tau^{'}$

  — Sinon, $e\left[x \leftarrow e^{'}\right] = e$ et $\tau = \Gamma(y)$

— Cas d'une abstraction $e = \texttt{fun}\ y \to e_1$.
On peut supposer, par α-conversion $y \neq x, y \notin dom(\Gamma)$ et y non libre dans e'. On a : $\Gamma + x : \tau' + y : \tau_2 \vdash e_1 : \tau_1$ et donc $\Gamma + y : \tau_2 + x : \tau' \vdash e_1 : \tau_1$ par permutation. D'autre part, $\Gamma \vdash e' : \tau'$ et donc par affaiblissement $\Gamma + y : \tau_2 \vdash e' : \tau'$.
Par HR on a donc : $\Gamma + y : \tau_2 \vdash e_1 \left[x \leftarrow e'\right] : \tau_1$ et donc :$\Gamma \vdash (\texttt{fun}\ y \to e_1) \left[x \leftarrow e'\right] : \tau_2 \to \tau_1$
i.e. $\Gamma \vdash e \left[x \leftarrow e'\right] : \tau$

■

Lemme 18.3.5 (Préservation)**.** *Si $\varnothing \vdash e : \tau$ et $e \to e'$ alors $\varnothing \vdash e' : \tau$.*

Démonstration. Par récurrence sur la dérivation $\varnothing \vdash e : \tau$. ■

Théorème 18.3.6. *Si $\varnothing \vdash e : \tau$, alors la réduction de e est infinie ou se termine sur une valeur. De manière équivalente : si $\varnothing \vdash e : \tau$ et $e \xrightarrow{\star} e'$ et e' irréductible, e' est une valeur.*

Remarque 18.3.6.1. *Ceci signifie qu'un programme bien typé ne plante pas, au sens de la sémantique à petits pas.*

Démonstration. On a $e \to e_1 \to \ldots \to e'$ et par applications répétées du lemme de préservation, on a donc $\varnothing \vdash e' : \tau$. Par le lemme de progression, e' se réduit ou est une valeur, c'est donc une valeur. ■

Il est toutefois contraignant de donner un type unique à $\texttt{fun}\ x \to x$ dans $\texttt{let fun}\ x \to x\ \texttt{in}\ \ldots$. De même on aimerait pouvoir donner plusieurs types aux primitives : c'est le polymorphisme paramétrique.

19 Polymorphisme Paramétrique

19.1 Système F

On étend l'algèbre des types :

$$\begin{array}{|l} \tau \\ | \\ | \\ | \\ | \\ | \end{array} \begin{array}{l} \texttt{int} \mid \texttt{bool} \mid \ldots \\ \tau \to \tau \\ \tau \times \tau \\ \alpha : \text{variable de type} \\ \forall \alpha.\tau : \text{type polymorphe} \end{array}$$

Définition 19.1.1. *On note $\mathcal{L}(\tau)$ l'ensemble des variables de types libres dans τ, défini par :*

$$\begin{aligned} \mathcal{L}(\textit{int}) &= \varnothing \\ \mathcal{L}(\alpha) &= \{\alpha\} \\ \mathcal{L}(\tau_1 \to \tau_2) &= \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\ \mathcal{L}(\tau_1 \times \tau_2) &= \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\ \mathcal{L}(\forall \alpha.\tau) &= \mathcal{L}(\tau) \setminus \{\alpha\} \end{aligned}$$

Pour un environnement Γ on définit aussi $\mathcal{L}(\Gamma) = \bigcup_{x \in dom(\Gamma)} \mathcal{L}(\Gamma(x))$

Définition 19.1.2. *On note* $\tau\left[\alpha \leftarrow \tau'\right]$ *la substitution de* α *par* τ' *dans* τ*, définie par :*

$$\begin{aligned}
\mathit{int}\left[\alpha \leftarrow \tau'\right] &= \mathit{int} \\
\alpha\left[\alpha \leftarrow \tau'\right] &= \tau' \\
\beta\left[\alpha \leftarrow \tau'\right] &= \beta \ \mathit{si}\ \beta \neq \alpha \\
(\tau_1 \to \tau_2)\left[\alpha \leftarrow \tau'\right] &= \tau_1\left[\alpha \leftarrow \tau'\right] \to \tau_2\left[\alpha \leftarrow \tau'\right] \\
\mathit{int}\left[\alpha \leftarrow \tau'\right] &= \mathit{int} \\
\mathit{int}\left[\alpha \leftarrow \tau'\right] &= \mathit{int} \\
\mathit{int}\left[\alpha \leftarrow \tau'\right] &= \mathit{int}
\end{aligned}$$

Les règles sont les mêmes qu'auparavant, plus :

$$\frac{\Gamma\vdash e{:}\tau \quad \alpha \not\in \mathcal{L}(\Gamma)}{\Gamma\vdash e{:}\forall\alpha.\tau} \text{ et } \frac{\Gamma\vdash e{:}\forall\alpha.\tau}{\Gamma e{:}\tau\left[\alpha\leftarrow\tau'\right]}$$

Le système obtenu s'appelle le système F.

Il permet de donner un type satisfaisant aux primitives : $\texttt{fst} : \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$. La condition $\alpha \notin \mathcal{L}(\Gamma)$ est cruciale, sinon on accepterait le programme $(\texttt{fun}\ x \to x)\ 1\ 2$.

Pour des termes sans annotations les deux problèmes d'inférence et de vérification ne sont pas décidables.

19.2 Système de Hindley-Milner

On limite la quantification $\forall$ en tête des types (quantification prénexe.)

$$\begin{array}{rl}
\tau| & \texttt{int} \mid \texttt{bool} \mid \ldots \\
| & \tau \to \tau \\
| & \tau \times \tau \\
| & \alpha : \text{variable de type} \\
| & \\
\sigma| & \tau : \text{schéma} \\
| & \forall\alpha.\sigma
\end{array}$$

L'environnement Γ associe un schéma de type à chaque identificateur et la relation de typage a maintenant la forme $\Gamma \vdash e : \sigma$.

On modifie toutes les règles :

$$\frac{}{\Gamma\vdash x{:}\Gamma(x)} \quad \frac{}{\Gamma\vdash n{:}\texttt{int}} \quad \frac{}{\Gamma\vdash +{:}\texttt{int}\times\texttt{int}\to\texttt{int}}$$

$$\frac{\Gamma+x{:}\tau_1\vdash e{:}\tau_2}{\Gamma\vdash \texttt{fun}\ x\to e{:}\tau_1\to\tau_2} \quad \frac{\Gamma\vdash e_1{:}\tau'\to\tau \ \Gamma\vdash e_2{:}\tau'}{\Gamma\vdash e_1\ e_2{:}\tau}$$

$$\frac{\Gamma\vdash e_1{:}\tau_1 \ \Gamma\vdash e_2{:}\tau_2}{\Gamma\vdash (e_1,e_2){:}\tau_1\times\tau_2} \quad \frac{\Gamma\vdash e_1{:}\sigma_1 \ \Gamma+x{:}\sigma_1\vdash e_2{:}\sigma_2}{\Gamma\vdash \texttt{let}\ x=e_1\ \texttt{in}\ e_2{:}\sigma_2}$$

$$\frac{\Gamma\vdash e{:}\sigma \quad \alpha\not\in\mathcal{L}(\Gamma)}{\Gamma\vdash e{:}\forall\alpha.\sigma} \quad \frac{\Gamma\vdash e{:}\forall\alpha.\sigma}{\Gamma e{:}\sigma\left[\alpha\leftarrow\tau'\right]}$$

On note que seule la construction `let` permet d'introduire un type polymorphe dans l'environnement.

Pour programmer une vérification ou une inférence de type, on procède récursivement sur la structure du programme, ce qui semble devoir se faire en temps exponentiel. Nous allons alors modifier la présentation du système de Hindley-Milner pour qu'il soit dirigé par la syntaxe (synta-directed), i.e., pour toute expression, au plus une règle s'applique. Les règles ont la même puissance d'expression : tout terme clos est typable dans un système si et seulement si il est typable dans l'autre :

$$\frac{\tau\leq\Gamma(x)}{\Gamma\vdash x{:}\tau} \quad \frac{}{\Gamma\vdash n{:}\texttt{int}} \quad \frac{\tau\leq\texttt{type}(op)}{\Gamma\vdash op{:}\tau}$$

$$\frac{\Gamma+x{:}\tau_1\vdash e{:}\tau_2}{\Gamma\vdash \texttt{fun}\ x\to e{:}\tau_1\to\tau_2}\quad \frac{\Gamma\vdash e_1{:}\tau^{'}\to\tau\ \Gamma\vdash e_2{:}\tau^{'}}{\Gamma\vdash e_1\ e_2{:}\tau}$$

$$\frac{\Gamma\vdash e_1{:}\tau_1\ \Gamma\vdash e_2{:}\tau_2}{\Gamma\vdash(e_1,e_2){:}\tau_1\times\tau_2}\quad \frac{\Gamma\vdash e_1{:}\tau 1\ \Gamma+x{:}\texttt{Gen}(\tau_1,\Gamma)\vdash e_2{:}\tau 2}{\Gamma\vdash \texttt{let}\ x{=}e_1\ \texttt{in}\ e_2{:}\tau 2}$$

Deux opérations apparaissent :

— L'instanciation : $\tau \leq \sigma$ se lit τ est une instance de σ et est définie par :

$$\tau \leq \forall\alpha_1 \dots \alpha_n.\tau^{'} \Leftrightarrow \exists\tau_1 \dots \exists\tau_n.\tau = \tau^{'}\,[\alpha_1 \leftarrow \tau_1, \dots, \alpha_n \leftarrow \tau_n]$$

— La généralisation :

$$\texttt{Gen}(\tau_1, \Gamma) = \forall\alpha_1 \dots \forall\alpha_n.\tau_1 \text{ où } \{\alpha_1, \dots, \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(\Gamma)$$

Pour inférer le type d'une expression il reste des problèmes : quel type donner dans la définition d'une fonction, quelle instance de $\Gamma(x)$ choisir ? Il existe une solution, l'algorithme W.

19.3 Algorithme W

On utilise de nouvelles variables de types pour représenter les types inconnus, et on détermine la valeur de ces variables plus tard par unification entre types au moment du typage de l'application.

Définition 19.3.1. *Si τ_1, τ_2 sont des types contenant des variables de types α_i. On appelle unification une instanciation f des variables α_i telle que $f(\tau_1) = f(\tau_2)$.*

Définition 19.3.2. *$\texttt{unifier}(\tau_1, \tau_2)$ détermine s'il existe une instance des variables de types de τ_1 et τ_2 telle que $\tau_1 = \tau_2$. :*

— $\texttt{unifier}(\tau, \tau) = \top$

— $\texttt{unifier}(\tau_1 \to \tau_1^{'}, \tau_2 \to \tau_2^{'}) = \texttt{unifier}(\tau_1, \tau_2); \texttt{unifier}(\tau_1^{'}, \tau_2^{'})$

— $\texttt{unifier}$

— $\texttt{unifier}(\alpha, \tau) =$ *si $\alpha \notin \mathcal{L}(\tau)$, remplacer α par τ partout, sinon échec*

— $\texttt{unifier}(\tau, \alpha) = \texttt{unifier}(\alpha, \tau)$

— *Sinon, échec.*

On définit une fonction W qui prend en arguments un environnement Γ et une expression e et renvoie le type inféré pour e.

— Si e est une variable x, renvoyer une instance triviale de $\Gamma(x)$.

— Si e est une constante c, renvoyer une instance triviale de son type (penser à $[] : \alpha\texttt{list}$)

— Si e est une primitive op, renvoyer une instance triviale de son type.

— Si e est une paire (e_1, e_2), calculer $\tau_1 = W(\Gamma, e_1)$, $\tau_2 = W(\Gamma, e_2)$, renvoyer $\tau_1 \times \tau_2$.

— Si e est une fonction $\texttt{fun}\ x \to e_1$, soit α une nouvelle variable, calculer $\tau = W(\Gamma + x : \alpha, e_1)$ et renvoyer $\alpha \to \tau$.

— Si e est une application $e_1\ e_2$, calculer τ_1, τ_2, soit α une nouvelle variable, $\texttt{unifier}(\tau_1, \tau_2 \to \alpha)$, renvoyer α.

— Si e est $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, calculer τ_1 et renvoyer $W(\Gamma + x : \texttt{Gen}(\tau_1, \Gamma), e_2)$.

Théorème 19.3.1 (Damas, Milner, 1982)**.** *L'algorithme W est correct, complet et détermine le type principal :*

— *Si $W(\varnothing, e) = \tau$ alors $\varnothing \vdash e : \tau$*

— *Si $\varnothing \vdash e : \tau$ alors $\tau \leq \forall\alpha.W(\varnothing e)$*

Théorème 19.3.2. *Le système de Hindley-Milner est sûr.*

19.4 Unification Algorithmique

Il existe plusieurs manières de réaliser l'unification :

— En manipulant explicitement une substitution :

```
type tvar = int
type subst = typ TVmap.t
```

— En utilisant des variables de types destructives :

```
type tvar = { id: int; mutable def: typ option; }
```

Il existe également plusieurs façons de coder l'algorithme W :

— Avec des schémas explicites et en calculant $\texttt{Gen}(\tau, \Gamma)$.

— Avec des niveaux.

20 Extensions

Ajouter de la récursion, ou des types construits (n-uplets, listes, types sommes et produits) se fait sans trop de difficulter, avec par exemple :

$$\frac{\Gamma + f : \tau \to \tau_1 + x; \tau \vdash e_1 : \tau_1 \quad \Gamma + f : \texttt{Gen}(\tau \to \tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec } fx = e_1 \texttt{ in } e_2 : \tau_2}$$

et

$$\frac{\Gamma \vdash e_1 : \tau\texttt{list} \; \Gamma \vdash e_2 : \tau_1 \; \Gamma + x : \tau + y : \tau \texttt{ list } \vdash e_3 : \tau_1}{\Gamma \vdash \texttt{ match } e_1 \texttt{ with } [] \to e_2 \mid :: (x, y) \to e_3 : \tau_1}$$

Pour les références toutefois, simplement les ajouters aux primitives ne fonctionne pas, c'est le problème des références polymorphes.

Définition 20.0.1. *Un programme satisfait le critère de la value restriction si toute sous-expression* `let` *dont le type est généralisé est de la forme* `let` $x = v_1$ `in` e_2 *où* v_1 *est une valeur.*

Ceci permet une solution très simple pour contourner le problème, c'est une restriction syntaxique.

Toutefois, dans ce cas là, après une première unification, on fixe le type. La solution est alors d'η-expanser pour faire apparaître une fonction, i.e. une valeur. En présence d'un système de modules, la réalité est encore plus complexe.

Septième partie

Cours 7 : 17/11

21 Stratégie d'évaluation et passage des paramètres

21.1 Evaluation

Définition 21.1.1. *Dans la déclaration d'une fonction* `function f(x1, ..., xn) = ...`*, les variables* `x1, ..., xn` *dont appelées paramètres formels de* `f`*. Dans l'appel de cette fonction* `f(e1, ..., en)`*, les expressions* `e1, ..., en` *sont les paramètres effectifs de* `f`*.*

Définition 21.1.2. *Dans un langage comprenant des modifications en place, une affectation* `e1 := e2` *modifie un emplacement mémoire désigné par l'expression* `e1`*. Celle-ci est limitée à certaines constructions, on parle de valeur gauche pour désigner les expressions légales à gauche d'une affectation.*

Définition 21.1.3. *La stratégie d'évaluation d'un langage spécifie l'ordre dans lequel les calculs sont effectués. On peut la définir à l'aide d'une sémantique formelle (cf cours 2). Le compilateur se doit de respecter la stratégie d'évaluation.*
On distingue :

— *L'évaluation stricte : les opérandes/paramètres effectifs sont évalués avant l'opération/l'appel, par exemple en C, C++, Python, Java, OCamL*

— *L'évaluation paresseuse : les opérandes/paramètres effectifs ne sont que si nécessaire, par exemple en Haskell, Clojure ou pour les connectives logiques.*

— Un langage impératif adopte systématiquement une évaluation stricte, pour garantir une séquentialité des effets de bord qui coïncide avec le texte source. On fait souvent exception des connectives logiques. La non-terminaison est également un effet.

— Un langage purement applicatif, c'est-à-dire sans effets de bord, peut en revanche adopter la stratégie d'évaluation de son choix, car une expression aura toujours la même valeur. On parle de transparence référentielle.

21.2 Passage des Paramètres

La sémantique précise également le mode de passage des paramètres lors d'un appel. On distingue :

— L'appel par valeur : de nouvelles variables représentant les paramètres formels reçoivent les valeurs des paramètres effectifs.

```
let f(x) =
    x := x + 1

main() =
    let v = 41
    f(v)
    print(v) (*affiche 41*)
```

— L'appel par référence : les paramètres formels désignent les mêmes valeurs gauches que les paramètres effectifs

```
let f(x) =
    x := x + 1

main() =
    let v = 41
    f(v)
    print(v) (*affiche 42*)
```

— L'appel par nom : les paramètres effectifs sont substitués aux paramètres formels, et donc évalués seulement si nécessaire

```
let f(x, y, z) =
    x*x + y*y

main() =
    int v := 41
    f(v)
    print(v) (*affiche 41*)
```

— L'appel par nécessité : les paramètres effectifs ne sont évalués que si nécessaire, mais au plus une fois

21.3 Exemples

21.3.1 Le Langage C

Le langage C est un langage impératif relativement bas niveau, notamment parce que la notion de pointeur, et d'arithmétique de pointeur, y est explicite. On peut le considérer inversement comme un assembleur de haut niveau.

Le langage C est muni d'une stratégie d'évaluation stricte avec appel par valeur. L'ordre d'évaluation n'est pas spécifié.

On trouve en C :

— Des types de base tels que `char, int, bool, float`...

— Un type $\tau*$ des pointeurs vers des valeurs de type τ. Si p est un pointeur de type $\tau*$, alors $*p$ désigne la valeur pointée par p, de type τ. Si e est une valeur gauche de type τ, alors $\&e$ est un pointeur sur l'emplacement mémoire correspondant de type $\tau*$.

— Des enregistrements appelés structures.

En C une valeur gauche est de la forme :

— `x`, une variable

— `*e`, le déréférencement d'un pointeur

— `e.x` l'accès à un champ de structure si `e` est elle-même une valeur gauche

— `t[e]` qui est en réalité `*(t+e)`

— `e -> x` qui n'est autre que `(*e).x`

L'appel par valeur implique que les structures sont copiées lorsqu'elles sont passées en paramètres ou renvoyées. Les structures sont également copiées lors des affectations de structures, i.e. des affectations de la forme `x = y`, où `x` et `y` ont le type `struct S` :

```
struct S { int a; int b; };

void f(struct S x) {
    x.b = x.b + 1;
}

int main() {
    struct S v = { 1, 2 };
    f(v);
    // v.b vaut toujours 2
}
```

On peut simuler un pasage par référence en passant un pointeur explicite :

```
void incr(int *x) {
    *x = *x + 1;
}

int main() {
    int v = 41;
    incr(&v);
    // v vaut maintenant 42
}
```

Mais ce n'est que le passage d'un pointeur par valeur.

Pour éviter le coût des copies, on passe des pointeurs sur les structures le plus souvent.

```
struct S { int a; int b; };

void f(struct S *x) {
    x->b = x->b + 1;
```

```
}

int main() {
    struct S v = { 1, 2 };
    f(&v);
    // v.b vaut maintenant 3
}
```

La manipulation explicite de pointeurs peut être dangereuse :

```
int* p() {
    int x;
    ...
    return &x;
}
```

Cette fonction renvoie un pointeur qui correspond à un emplacement sur la pile qui vient justement de disparaître (à savoir le tableau d'activation de p), et qui sera très probablement réutilisé rapidement par un autre tableau d'activation. On parle de référence fantôme (dangling reference).

On peut déclarer un tableau ainsi : `int t[10];`. La notation `t[i]` n'est que du sucre syntaxique pour `*(t+i)` où `t` désigne un pointeur sur le début d'une zone contenant 10 entiers et `+` désigne une opération d'arithmétique de pointeur. Le premier élément du tableau est donc `t[0]` i.e. `*t`. Quand on passe un tableau en paramètre, on ne fait que passer le pointeur. On ne peut pas affecter des tableaux, seulement des pointeurs : On ne peut pas écrire :

```
void p() {
    int t[3];
    int u[3];
    t = u; // <- erreur
}
```

car `t` et `u` sont des tableaux. Mais on peut écrire :

```
void q(int t[3], int u[3]) {
    t = u;
}
```

car c'est exactement pareil que :

```
void q(int *t, int *t) {
    t = u;
}
```

et l'affectation de pointeurs est autorisée.

21.3.2 Le langage C++

En C++ on trouve les types et constructions du C, avec une stratégie d'évaluation stricte. Le mode de passage est par valeur par défaut, mais on trouve aussi un passage par référence indiqué par le symbole `&` au niveau de l'argument formel :

```
void f(int &x) {
    x = x + 1;
}

int main() {
    int v = 41;
    f(v);
    // v vaut maintenant 42
}
```

En particulier, c'est le compilateur qui a pris l'adresse de `v` au moment de l'appel, et qui a déréférencé l'adresse `x` dans la fonction `f`. On peut aussi passer une structure par référence :

```
struct S { int a; int b; };
void f(struct S &x) {
    x.b = x.b + 1;
}
int main() {
    struct S v = { 1, 2 };
    f(v);
    // v.b vaut maintenant 3
}
```

et des pointeurs par référence, par exemple pour ajouter un élément dans un arbre :

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
    if (t == NULL ) t = create(NULL, x, NULL);
    else if (x < t->elt) add(t->left, x);
    else if (x > t->elt) add(t->right, x);
}
```

21.3.3 Le langage OCamL

OCamL est muni d'une stratégie d'évaluation stricte, avec appel par valeur. L'ordre d'évaluation n'est pas spécifié.

Une valeur est soit d'un type primitif (booléen, caractère, entier machine, ...), soit un pointeur vers un bloc mémoire (tableau, enregistrement, constructeur non constant, ...) alloué sur le tas en général. Les valeurs gauches sont les éléments de tableaux et les champs mutables d'enregistrements.

Une référence est un enregistrement :

```
type 'a ref = { mutable contents: 'a }
let (!) r = r.contents
let (:=) r v = r.contents <- v
```

Une référence est allouée sur le tas. C'est toujours un passage par valeur, d'une valeur qui est un pointeur implicite vers une valeur mutable. Un tableau est également alloué sur le tas.

On peut simuler l'appel par nom en OCamL, en remplaçant les arguments par des fonctions :

```
let f x y =
    if x = 0 then 42 else y + y
(*Peut être réécrite en : *)
let f x y =
    if x () = 0 then 42 else y () + y ()

let v = f (fun () -> 0) (fun () -> failwith "oups")
```

Plus subtilement, on peut aussi simuler l'appel par nécessité :

— On commence par introduire un type pour représenter les calculs paresseux :

```
type 'a value = Value of 'a
    | Frozen of (unit -> 'a)
type 'a by_need = 'a value ref
```

— et une fonction qui évalue un tel calcul si ce n'est pas déjà fait :

```
let force l = match !l with
    | Value v -> v
    | Frozen f -> let v = f () in l := Value v; v
```

C'est de la mémoïsation.

Remarque 21.3.0.1. *La construction* `lazy` *d'OCamL fait quelque chose de semblable.*

21.3.4 Le langage Java

Java est muni d'une stratégie d'évaluation stricte, avec appel par valeur. L'ordre d'évaluation est spécifié gauche-droite.
Une valeur est soit d'un type primitif (booléen, caractère, entier machine, ...), soit un pointeur vers un object alloué sur le tas.

```
void f(int x) {
    x = x + 1;
}
int main() {
    int v = 41;
    f(v);
    // v vaut toujours 41
}
```

Un objet est alloué sur le tas :

```
    class C { int f; }
        void incr(C x) {
        x.f += 1;
    }
    void main () {
        C r = new C();
        r.f = 41;
        incr(r);
        // r.f vaut maintenant 42
    }
```

C'est toujours un passage par valeur, d'une valeur qui est un pointeur implicite vers un objet.

21.3.5 Le Langage Python

Python est muni d'une stratégie d'évaluation stricte, avec appel par valeur. L'ordre d'évaluation est spécifié gauche-droite (mais droite-gauche pour une affectation). Une valeur est un pointeur vers un objet alloué sur le tas. Un entier est un objet immuable :

```
    def f(x) :
        x += 1
    v = 41
    f(v)
    print(v)
```

Un tableau est un objet mutable :

```
    def incr(x):
        x[1] += 1
    a = [0] * 17
    a[1] = 41
    incra(a)
```

Les modèles d'exécutions de Java, OCamL et Python sont très semblables : uniquement du passage par valeurs, de valeurs atomiques (64 bits), et ce, même si leurs langages de surface sont très différents.

22 Compilation du passage par valeur et par référence

22.1 Micro C++

Considérons la compilation d'un micro fragment de C++ avec :

— des entiers

— des fonctions (mais qui ne renvoient rien)

— du passage par valeur et par référence

On considère le fragment suivant :

$$
\begin{array}{lll}
E \to n & C \to E == E \mid E != E & \\
\mid x & \quad \mid E < E \mid E <= E \mid E > E \mid E >= E & \\
\mid E + E \mid E - E & \mid C & C \\
\mid E * E \mid E / E & \mid C \mid\mid C & \\
\mid -E & \mid !C &
\end{array}
$$

$$
\begin{array}{ll}
S \to x = E; & B \to \{\ S \ldots\ S\ \} \\
\quad \mid \texttt{if}\ (\ C\ )\ S & \\
\quad \mid \texttt{if}\ (\ C\ )\ S\ \texttt{else}\ S & F \to \texttt{void}\ f(X, \ldots, X)\ B \\
\quad \mid \texttt{while}\ (\ C\ )\ S & \\
\quad \mid f(E, \ldots, E); & X \to \texttt{int} x \\
\quad \mid \texttt{printf("\%d /n"}, E); & \quad \mid \texttt{int \&} x \\
\quad \mid \texttt{int}\ x, \ldots, x; & \\
\quad \mid B & P \to F \ldots F \\
 & \quad \texttt{int main()}\ B
\end{array}
$$

Par exemple, le programme suivant est correct et compile :

```
void fib(int n, int &r) {
if (n <= 1)
    r = n;
else {
    int tmp;
    fib(n - 2, tmp);
    fib(n - 1, r);
    r = r + tmp;
    }
}
int main() {
    int f;
    fib(10, f);
    printf("%d\n", f);
}
```

22.2 Variables et Portées

Définition 22.2.1. *La portée définit les portions du programme où une variable est visible. Ici, si le corps d'une fonction f mentionne une variable x alors :*

— *Soit x est un paramètre de f*

— *Soit x est déclarée plus haut dans un bloc englobant*

Par ailleurs, un train peut en cacher un autre.

Par exemple le programme suivant compile :

```c
void f(int n) {
    printf("%d\n", n); // affiche 34
    if (n > 0) {
        int n; n = 89;
        printf("%d\n", n); // affiche 89
    }
    if (n > 21) {
        printf("%d\n", n); // affiche 34
        int n; n = 55;
        printf("%d\n", n); // affiche 55
    }
    printf("%d\n", n); // affiche 34
}
int main() {
    f(34);
}
```

Ici, la portée ne dépend que du texte source, on parle de portée lexicale. On peut la réaliser avant ou pendant le typage. La syntaxe abstraite conserve une trace de cette analyse, en identifiant chaque variable de façon unique.

Avant	Après
Arbres de syntaxe abstraite issus de l'analyse syntaxique	Arbres de syntaxe abstraite après le typage
`type pint_expr =	PEconst of int	PEvar of string	... type pstmt =	PSvars of string list	PSblock of pstmt list	...`	`type int_expr =	Econst of int	Evar of ident	... type func = { locals: ident list; ... }`
Pour l'instant, les variables sont des chaînes de caractères.	Maintenant `ident` est un identifiant : entier, nom unique, enregistrement, etc.

Il existe des langages ou la portée est dynamique, i.e. dépend de l'exécution du programme, par exemple Bash.

Il faut choisir un emplacement mémoire pour chaque variable et être capable de calculer cet emplacement à l'exécution. Ici, les variables vont toutes être stockées sur la pile. A chaque fonction en cours d'exécution correspond une portion de la pile, appelée tableau d'activation, qui contient notamment ses paramères effectifs, l'adresse de retour et ses variables locales. On positionne le registre `%rbp` au milieu de la pile de sorte à retrouver facilement l'emplacement d'une variable en ajoutant 8, 16 etc... En effet, le sommet de pile peut bouger si on y stocke des calculs intermédiaires ou si on est en train de préparer un appel de fonction. Pour chaque variable, le compilateur détermine une position dans la pile. Par exemple, dans le type `ident` :

```
type ident = { offset : int; ... }
```

Pour les paramètres, ce sont +16, +24, etc... Pour les variables, ce sont -8, -16, etc, avec souvent plusieurs solutions possibles.

22.3 Passage Par Valeu

On utilise la bibliothèque `X86_64` pour produire du code assembleur. On concatène les morceaux d'assembleur avec une opération `++`. On suit un schéma de compilation simpliste, utilisant la pile pour stocker les résultats intermédiaires : A l'issue de l'exécution d'une fonction de compilation d'une expression arithmétique, la valeur de l'expression se trouve dans le registre `%rdi` :

```
let rec int_expr = function
    (*on commence par les constantes*)
    | Econst n ->
        movq (imm n) (reg rdi)
    (*et le opérations arithmétiques*)
    | Ebinop (Badd, e1, e2) ->
        int_expr e1 ++
        pushq (reg rdi) ++
        int_expr e2 ++
        popq rsi ++
        addq (reg rsi) (reg rdi)
    | Ebinop (Bsub, e1, e2) ->
        ...
```

Pour une variable, on utilise l'adressage indirect, car la position par rapport à `%rbp` est une constante connue :

```
| Evar { offset = ofs } ->
    movq (ind ~ofs rbp) (reg rdi)
```

De même, les expressions booléennes sont compilées d'une manière très analogue. On prend juste garde au fait que les opérateurs `&&` et `||` doivent être évalués paresseusement :

```
let rec bool_expr = function
| Bcmp (Beq, e1, e2) ->
    int_expr e1 ++ pushq (reg rdi) ++
    int_expr e2 ++ popq rsi ++
    cmpq (reg rdi) (reg rsi) ++
    sete (reg dil) ++ movzbq (reg dil) rdi
| ...
```

Les instructions sont compilées avec une fonction :

```
let rec stmt = function
    | Sprintf e ->
        int_expr e ++ call "print_int"
    | Sif (e, s1, s2) ->
        ...
    | Swhile (e, s) ->
        ...
    | Sblock sl ->
        List.fold_left (fun code s -> code ++ stmt s) nop sl
```

avec :

```
print int: # (en pratique, il faut aussi aligner la pile)
    movq %rdi, %rsi
    movq $.Sprint int, %rdi
    movq $0, %rax
    call printf
    ret
.data
.Sprint int:
    .string "%d\n"
```

Pour appeler une fonction f il faut :

1. Empiler les arguments
2. Appeler le code situé à l'étiquette f
3. Dépiler les arguments

```
| Scall (id, el) ->
    List.fold_left
        (fun acc e -> int_expr e ++ pushq (reg rdi) ++ acc)
        nop el ++
    call (symb id) ++
    addq (imm (8 * List.length el)) (reg rsp)
```

Reste l'affectation `x = e` : le membre de gauche est ici réduit à une variable `x` et on sait où cette variable est stockée sur la pile :

```
| Sassign ({ offset = ofs }, e) ->
    int_expr e ++
    movq (reg rdi) (ind ~ofs rbp)
```

22.4 Passage par Référence

Pour l'instant on a passé les paramètres par valeur, i.e. le paramètre formel est une nouvelle variable qui prend comme valeur initiale celle du paramètre effectif. En C++, le qualificatif `&` permet de spécifier un passage par référence. Dans ce cas, le paramètre formel désigne la même variable que le paramètre effectif, qui doit donc être une valeur gauche. Pour prendre en compte le passage par référence, on étend encore le type `ident` pour indiquer s'il s'agit d'une variable passée par référence. Dans un appel tel que $f(e)$, le paramètre effectif e n'est plus typé ni compilé de la même manière selon qu'il s'agit d'un paramètre passé par valeur ou par référence. Lorsque le paramètre est passé par référence le typage va donc :

1. Vérifier qu'il s'agit bien d'une valeur gauche (une variable ici)
2. Indiquer qu'elle doit être passée par référence

Une façon de procéder consiste à ajouter une construction de calcul de valeur gauche dans la syntaxe des expressions, et à remplacer le cas échéant, le paramètre effectif `e` par `Eaddr e`. On modifie alors :

```
type int_expr =
    ...
    | Eaddr of ident

let rec int_expr = function
    | Eaddr { offset = ofs; by_reference = br } ->
        leaq (ind ~ofs rbp) rdi ++
        if br then movq (ind rdi) (reg rdi) else nop
```

Il faut modifier aussi le calcul des valeurs droites et de l'affectation, mais pas celui de l'appel grâce à la nouvelle construction :

```
| Evar { offset = ofs; by_reference = br } ->
    movq (ind ~ofs rbp) (reg rdi) ++
    if br then movq (ind rdi) (reg rdi) else nop

| Sassign ({ offset = ofs; by_reference = br}, e) ->
    int_expr e ++
    (if br then movq (ind ~ofs rbp) (reg rsi)
     else leaq (ind ~ofs rbp) rsi) ++
    movq (reg rdi) (ind rsi)
```

Il reste à compiler les déclarations des fonctions :

```
type function_decl =
{ fname : string;
  formals: ident list;
```

```
    locals : ident list;
    body : stmt; }

let function_decl f =
    let fs = List.fold_left (fun fs x -> max fs (abs x.offset)) 0 f.locals in
    ++ stmt f.body ++
```

On obtient alors pour la fonction `swap` par exemple :

```
void swap(int &x, int &y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
swap:   pushq %rbp
        movq %rsp, %rbp
        subq $8, %rsp
        movq 16(%rbp), %rdi
        movq 0(%rdi), %rdi
        leaq -8(%rbp), %rsi
        movq %rdi, 0(%rsi)
        movq 24(%rbp), %rdi
        movq 0(%rdi), %rdi
        movq 16(%rbp), %rsi
        movq %rdi, 0(%rsi)
        movq -8(%rbp), %rdi
        movq 24(%rbp), %rsi
        movq %rdi, 0(%rsi)
        movq %rbp, %rsp
        popq %rbp
        ret
```

23 Correction de la Compilation

23.1 Définition de la Correction

Le compilateur doit respecter la sémantique du langage : Si le langage source est muni d'une sémantique $\rightarrow_s$ et le langage machine $\rightarrow_m$ et si e est compilée en $C(e)$ on doit avoir un diagramme commutatif : $\begin{array}{ccc} e & \xrightarrow{\star}_s & v \\ \downarrow & & \approx \\ C(e) & \xrightarrow{\star}_m & v' \end{array}$ où $v \approx v'$ exprime que les valeurs coïncident.

23.2 Une Preuve de Correction

On montre ici la correction de la compilation sur les expressions arithmétiques sans variable : $e \rightarrow n \mid e + e$

On se donne une sémantique à réductions pour le langage source :

$$v \rightarrow n$$
$$E \rightarrow \square \mid E + e \mid v + E$$

et $n_1 + n_2 \xrightarrow{\varepsilon} n$ avec $n = n_1 + n_2$.

On se donne aussi une sémantique à réductions pour le langage cible. Un état est la donnée de valeurs pour les registres R et d'un état de la mémoire M :

$$R = \{\texttt{\%rdi} \mapsto n; \texttt{\%rsi} \mapsto n; \texttt{\%rsp} \mapsto n\}$$
$$M = \mathbb{N} \rightarrow \mathbb{Z}$$

La sémantique d'une instruction m se définit par une réduciton :

$$R, M, m \xrightarrow{m} R^{'}, M^{'}$$

La réduction se définit aisément.

On souhaite montrer que si $e \xrightarrow{\star} n$ et si $R, M, \texttt{code}(e) \xrightarrow{m}{}^{\star} R^{'}, M^{'}$ alors $R^{'}(\texttt{\%rdi}) = n$, où `code` désigne la fonction de compilation. On procède alors par induction structurelle sur e. On établit alors un résultat plus fort :

Proposition 23.2.1. *Si $e \xrightarrow{\star} n$ et $R, M, \texttt{code}(e) \xrightarrow{m}{}^{\star} R^{'}, M^{'}$ alors :*

$$\begin{cases} R^{'}(\texttt{\%rdi}) = & n \\ R^{'}(\texttt{\%rsp}) = & R(\texttt{\%rsp}) \\ \forall a \geq R(\texttt{\%rsp}),\ M^{'}(a) = M(a) & \end{cases}$$

23.3 A Grande Echelle

Une telle preuve put être effectuée sur un vrai compilateur : CompCert, un compilateur C produisant du code efficace a été formellement vérifié avec Coq. Voir `http://compcert.inria.fr/`

Huitième partie
Cours 24/11

24 Fonctions comme Valeurs de Première Classe

On va étudier un fragment d'OcamL :

$$\begin{aligned} e =& c \\ &|x \\ &|\texttt{fun}\ x \to e \\ &|ee \\ &|\texttt{let [rec]}\ x = e \\ &|\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \end{aligned}$$

$$d = \texttt{let [rec]}\ x = e$$

$$p = d\ \ldots\ d$$

Remarque 24.0.0.1. *Les fonctions peuvent ici être imbriquées. La compilation des fonctions imbriquées exploite le fait que toute variable qui est référencée est contenue dans un tableau d'activation quelque part sur la pile. Le compilateur chaîne les tableaux d'activation, de façon à toujours pouvoir retrouver celui qui nous intéresse.*

24.1 Fonctions en Paramètres

OCamL permet de recevoir des fonctions en argument ainsi que d'en renvoyer. La solution consiste à utiliser une fermeture.

Définition 24.1.1. *Une fermeture est une structure de données allouée sur le tas contenant un pointeur vers le code de la fonction à appeler et les valeurs des variables susceptibles d'être utilisées par ce code, ensemble appelés l'environnement.*

On considère l'exemple suivant de fermeture :

```
let rec pow i x =
    if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn n =
    let f = pow n in
    let eps = 0.001 in
    let rec sum x =
        if x >= 1. then 0. else f x +. sum (x +. eps) in
    sum 0. *. eps
```

On peut faire apparaître la construction `fun` explicitement :

```
let rec pow =
    fun i ->
        fun x -> if i = 0 then 1. else x *. pow (i-1) x
```

Dans la première fermeture `fun i ->`, l'environnement est {`pow`}. Dans la seconde c'est {`i, pow`}. De même :

```
let integrate_xn = fun n ->
    let f = pow n in
    let eps = 0.001 in
    let rec sum =
        fun x -> if x >= 1. then 0. else f x +. sum (x+. eps) in
    sum 0. *. eps
```

Pour `fun n ->` la fermeture est {`pow`} et pour `fun x ->` c'est {`eps, f, sum`}.

La fermeture peut être représentée comme un unique bloc sur le tas dont le premier champ contient l'adresse du code, et les champs suivants contiennent les valeurs des variables libres et uniquement celle-là.

24.2 Compilation

Une façon relativement simple de compiler les fermetures consiste à procéder en deux temps :

1. On recherche dans le code toutes les constructions `fun x -> e` et on les remplace par une construction explicite de fermeture `clos f` $[y_1, \ldots, y_n]$ où les y_i sont des variables libres de `fun x -> e` et `f` le nom donné à une déclaration globale de fonction de la forme `let fun f` $[y_1, \ldots, y_n]$ $x = e'$ où e' est obtenu à partir de e en supprimant récursivement les constructions `fun`.
2. On compile le code obtenu qui ne contient plus que des déclarations de fonction de la forme `let fun`.

Sur notre exemple :

```
letfun fun2 [i,pow] x =
    if i = 0 then 1. else x *. pow (i-1) x
letfun fun1 [pow] i =
    clos fun2 [i,pow]
let rec pow =
    clos fun1 [pow]
letfun fun3 [eps,f,sum] x =
    if x >= 1. then 0. else f x +. sum (x +. eps)
letfun fun4 [pow] n =
    let f = pow n in
    let eps = 0.001 in
    let rec sum = clos fun3 [eps,f,sum] in
    sum 0. *. eps
let integrate_xn =
    clos fun4 [pow]
```

Toutefois, on a énormément agrandi notre grammaire :

Avant

```
type var = string

type expr =
    | Evar of var
    | Efun of var * expr
    | Eapp of expr * expr
    | Elet of var * expr * expr
    | Eif of expr * expr * expr
    | ...
type decl = var * expr

type prog = decl list
```

Après

```
type var =
    | Vglobal of string
    | Vlocal of int
    | Vclos of int
    | Varg
type expr =
    | Evar of var
    | Eclos of string * var list
    | Eapp of expr * expr
    | Elet of int * expr * expr
    | Eif of expr * expr * expr
    | ...
type decl =
    | Let of string * expr
    | Letfun of string * expr
type prog = decl list
```

En particulier, un identificateur peut représenter soit une variable globale, soit une variable locale, soit une variable contenue dans une fermeture, soit l'unique argument d'une fonction.
On a alors le schéma de compilation suivant :

1. Chaque fonction a un unique argument, passé dans `%rdi`
2. On passe la fermeture dans `%rsi`
3. Le tableau d'activation étant donc composé de l'adresse de retour, du `%rbp` sauvegardé, et d'emplacements pour des variables locales. Il est intégralement construir par l'appelé.

Pour compiler la construction `Eclos`$(f, [y_1, \ldots, y_n])$:

1. On alloue un bloc de taille $n + 1$ sur le tas.
2. On stocke l'adresse de f dans le champ 0.
3. On stocke les valeurs des bariables dans les champs 1 à n.
4. On renvoie le pointeur sur le bloc.

Pour compiler un appel de fonction `Eapp`(e_1, e_2) :

1. On compile e_1 dans le registre `%rsi`
2. On compile e_2 dans le registre `%rdi`
3. On appelle la fonction dont l'adresse est contenue dans le premier champ de la fermeture avec `call *(%rsi)`.

Pour compiler un accès à une variable :

— Si c'est une variable globale, elle se trouve à l'adresse donnée par l'étiquette.
— Si c'est une variable locale, elle se trouve à l'adresse donnée par lui même.
— Si c'est une variable dans la fermeture, la valeur se trouve à l'adresse donnée par lui même.
— Si c'est un argument, la valeur se trouve dans `%rdi`.

Pour compiler la déclaration, comme pour une déclaration usuelle :

1. On sauvegarde et positionne `%rbp`
2. On alloue le tableau d'activation
3. On évalue e dans `%rax`
4. On désalloue le tableau d'activation et on restaure `%rbp`
5. On exécute `ret`

Il est inutilement coûteux de créer des fermetures intermédiaires dans un appel où tous les arguments sont fournis. Un appel traditionnel pourrait être fait. En revanche, une application partielle produirait une fermeture. CamL ait cette optimisation. Une autre optimisation est possible : lorsqu'on crée une fermeture dans une fonction à laquelle elle ne survivra pas, elle peut être allouée sur la pile plutôt que sur le tas. Mais pour s'assurer que cette optimisation est possible, il faut effectuer une analyse statique non triviale.

25 Optimisation des appels terminaux.

25.1 Appel Terminal

Définition 25.1.1. *On dit qu'un appel $f\ e_1, \dots e_n$ qui apparaît dans le corps d'une fonction g est terminal si c'est la dernière chose que g calcule avant de renvoyer son résultat. Par extension, une fonction est récursive terminale s'il s'agit d'une fonction récursive dont tous les appels récursifs sont terminaux.*

Du point de vue de la compilation, on peut détruire le tableau d'activation de la fonction où se trouve l'appel avant de faire l'appel, puisqu'il ne servira plus ensuite. Mieux, on peut le réutiliser pour l'appel terminal que l'on doit faire (en particulier, l'adresse de retour sauvegardée y est la bonne). Dit autrement, on peut faire un saut (`jump`) plutôt qu'un appel (`call`). Le programme obtenu est alors plus efficace, en particulier car on accède moins à la mémoire. Une autre conséquence étant que l'espace de pile utilisé devient constant. En particulier, on évite ainsi tout débordement de pile qui serait dû à un trop grand nombre d'appels imbriqués.

25.2 Codé Main

Quand le compilateur n'optimise pas les appels terminaux, on peut toujours le faire soi même dans le code. Toutefois, il n'est pas toujours facile de remplacer les appels par des appels terminaux. Par exemple la fonction suivante fait déborder la pile pour de trop grands arbres :

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree

let rec height = function
    |Empty -> 0
    | Node (l, _, r) -> 1 + max (height l) (height r)
```

Au lieu de calculer la hauteur h de l'arbre, on calcule $k(h)$ pour une fonction k quelconque, appelée continuation. On appelle ça la programmation par continuations. On déduit le programme voulu par la continuation identité :

```
let rec height t k = match t with
    | Empty -> k 0
    | Node (l, _, r) ->
        height l (fun hl ->
        height r (fun hr ->
        k (1 + max hl hr)))
```

Tous les appels à `height` et `k` sont terminaux. Le calcul se fait donc en espace de pile constant.

25.3 Explication

On a remplacé l'espace sur la pile par de l'espace sur le tas, qui est occupé par des fermetures. Il y a bien sûr d'autres solutions, mais la méthode à base de CPS est mécanique.

Dans le cas où le langage optimise l'appel terminal mais n'a pas de fonctions anonymes (e.g. C), on construit des fermetures soi-même, à la main :

```
enum kind { Kid, Kleft, Kright };
struct Kont {
    enum kind kind;
    union { struct Node *r; int hl; };
    struct Kont *kont;
};
int apply(struct Kont *k, int v) {...}
```

Cela s'appelle la défonctionnalisation.

26 Filtrage

26.1 Définition

Dans les langages fonctionnels, on trouve une construction appelée filtrage, utilisée dans les définitions de fonctions, les conditionnelles généralisées et les gestionnaires d'exceptions. L'objectif du compilateur est de transformer ces constructions de haut niveau en des séquences de tests élémentaires. On considère dans la suite la construction

$$\texttt{match } x \texttt{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

Un motif est défini par $p = x \mid C(p, \ldots, p)$. où C est un constructeur qui peut être :

- Une constante telle que `false, true, 0, 1, "hello"`, etc...
- Un constructeur constant de type somme, tel que `[]`
- Un constructeur d'arité $n \geq 1$ tel que `::`
- Un constructeur de n-uplet avec $n \geq 2$.

Définition 26.1.1 (motif linéaire)**.** *On dit qu'un motif est linéaire si toute variable apparaît au plus une fois dans p.*

Dans ce qui suit, on ne considère que des motifs linéaires.

Définition 26.1.2. *Les valeurs sont ici $v = C(v, \ldots, v)$ où C est le même que dans la définition des motifs.*

Définition 26.1.3. *On dit qu'une valeur v filtre un motif p s'il existe une substitution σ de variables par des valeurs telle que $v = \sigma(p)$.*

Il est clair que toute valeur filtre $p = x$

Proposition 26.1.1. *Une valeur v filtre $p = C(p_1, \ldots, p_n)$ si et seulement si v est de la forme $v = C(v_1, \ldots, v_n)$ avec v_i qui filtre p_i pour tout $i = 1, \ldots, n$.*

Démonstration.
- Soit v qui filtre p. On a $v = \sigma(p)$ pour un certain σ soit $v = C(\sigma(p_1), \ldots, \sigma(p_n))$ et on pose donc $v_i = \sigma(p_i)$.
- Réciproquement, si v_i filtre p_i pour tout i alors il existe des σ_i telles que $v_i = \sigma_i(p_i)$. Comme p est linéaire, les domaines des σ_i sont deux à deux disjoints et on a donc $\sigma_i(p_j) = p_j$ si $i \neq j$. En prenant $\sigma = \sigma_1 \circ \ldots \circ \sigma_n$, on a bien le résultat.

■

Définition 26.1.4. *Dans le filtrage : `match x with` $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$, si v est la valeur de x on dit que v filtre le cas p_i si v filtre p_i et si v ne filtre aucun p_j pour $j < i$. Le résultat du filtrage est alors $\sigma(e_i)$ où σ est la substitution telle que $\sigma(p_i) = v$.*

26.2 Compilation

On cosidère un premier algorithme de compilation du filtrage. On suppose disposer de :

— $\texttt{constr}(e)$ qui renvoie le constructeur de la valeur e

— $\texttt{\#}_i(e)$ qui renvoie sa i-ème composante.

Autrement dit, si $e = C(v_1, \ldots, v_n)$ alors $\texttt{constr}(e) = C$ et $\texttt{\#}_i(e) = v_i$

On commence par la compilation d'une ligne de filtrage :

$$\mathit{code}(\texttt{match } e \texttt{ with } p \to \mathit{action}) = F(p, e, \mathit{action})$$

où la fonction de compilation F est définie ainsi :

— $F(x, e, \mathit{action}) = \texttt{let } x = e \texttt{ in } \mathit{action}$

— $F(C, e, \mathit{action}) = \texttt{if constr}(e) = C \texttt{ then } \mathit{action} \texttt{ else } \mathit{error}$

— $F(C(p), e, \mathit{action}) = \texttt{if constr}(e) = C \texttt{ then } F(p, \texttt{\#}_1(e), \mathit{action}) \texttt{ else } \mathit{error}$

— $F(C(p_1, \ldots, p_n), e, \mathit{action}) = \texttt{if constr}(e) = C \texttt{ then } F(p_1, \texttt{\#}_1(e), F(p_2, \texttt{\#}_2(e), \ldots F(p_n, \texttt{\#}_n(e), \mathit{action})) \ldots) \texttt{ else}$

Proposition 26.2.1. *Cette compilation est correcte*

Démonstration. On montre simplement par récurrence sur p que si $e \xrightarrow{\star} v$ alors $F(p, e, \mathit{action}) \xrightarrow{\star} \sigma(\mathit{action})$ s'il existe σ telle que $v = \sigma(p)$ et $F(p, e, \mathit{action}) \xrightarrow{\star} \mathit{error}$ sinon. ■

Pour filtrer plusieurs lignes, on remplace *error* par le passage à la ligne. Toutefois, cet algorithme est peu efficace car on effectue plusieurs fois les mêmes tests, et on effectue des tests redondants [4].

26.3 Un Algorithme plus Efficace

On proopose un algorithme qui considère le problème du filtrage des n lignes dans sa globalité. On représente le problème sous forme d'une matrice :

$$\left|\begin{array}{cccccc} e_1 & e_2 & \cdots & e_m & & \\ p_{1,1} & p_{1,2} & \cdots & p_{1,m} & \to & \mathit{action}_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,m} & \to & \mathit{action}_n \end{array}\right|$$

L'algorithme F procède récursivement sur la matrice : Pour $n = 0$ on renvoie *error* et pour $m = 0$ on renvoie action_1. Si toute la colonne de gauche se compose de variables :

$$M = \left|\begin{array}{cccccc} e_1 & e_2 & \cdots & e_m & & \\ x_{1,1} & p_{1,2} & \cdots & p_{1,m} & \to & \mathit{action}_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ x_{n,1} & p_{n,2} & \cdots & p_{n,m} & \to & \mathit{action}_n \end{array}\right|$$

On élimine cette colonne en introduisant des `let` :

$$F(M) = F\left|\begin{array}{cccccc} e_2 & \cdots & e_m & & & \\ p_{1,2} & \cdots & p_{1,m} & \to & \texttt{let } x_{1,1} = e_1 \texttt{ in } \mathit{action}_1 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ p_{n,2} & \cdots & p_{n,m} & \to & \texttt{let } x_{n,1} = e_1 \texttt{ in } \mathit{action}_n \end{array}\right|$$

Sinon, c'est que la colonnne de gauche contient au moins un motif construit. Pour chaque constructeur dans cette colonne, on construit la sous-matrice correspondant au filtrage d'une valeur pour

4. NDS : comme le fenouil

ce constructeur : si

$$M = \left|\begin{array}{cccccc} e_1 & e_2 & \cdots & e_m & & \\ C(q) & p_{1,2} & \cdots & p_{1,m} & \to & action_1 \\ D & p_{2,2} & \cdots & p_{2,m} & \to & action_2 \\ x & p_{3,2} & \cdots & p_{3,m} & \to & action_3 \\ E(r,s) & p_{4,2} & \cdots & p_{4,m} & \to & action_4 \\ y & p_{5,2} & \cdots & p_{5,m} & \to & action_5 \\ C(t) & p_{6,2} & \cdots & p_{6,m} & \to & action_6 \\ E(u,v) & p_{7,2} & \cdots & p_{7,m} & \to & action_7 \end{array}\right|$$

on pose pour le constructeur C :

$$M_C = \left|\begin{array}{cccccc} \#_1(e_1) & e_2 & \cdots & e_m & & \\ q & p_{1,2} & \cdots & p_{1,m} & \to & action_1 \\ - & p_{2,2} & \cdots & p_{2,m} & \to & \texttt{let } x = e_1 action_2 \texttt{ in} \\ - & p_{3,2} & \cdots & p_{3,m} & \to & \texttt{let } y = e_1 \texttt{ in } action_3 \\ t & p_{6,2} & \cdots & p_{6,m} & \to & action_6 \end{array}\right|$$

On fait de même pour D et E ainsi que pour les variables.

On pose alors :

$$\begin{aligned} F(M) = \texttt{case } & multlineconstr(e_1) \texttt{ in} \\ & C \Rightarrow F(M_C) \\ & D \Rightarrow F(M_D) \\ & E \Rightarrow F(M_E) \\ & \texttt{otherwise } \Rightarrow F(M_R) \end{aligned}$$

Proposition 26.3.1. *Cet algorithme termine et est correct*

Démonstration. La grandeur $\sum_{i,j} taille(p_{i,j})$ diminue strictement à chaque appel. ■

Le type de l'expression e_1 permet d'optimiser la construction dans de nombreux cas :

— Pas de test si un seul constructeur

— Pas de cas `otherwise` si on peut restreindre à des constructeurs.

— Un simple `if then else` s'il n'y a que deux constructeurs

— Une table de saut lorsqu'il y a un nombre fini de constructeurs.

— Un arbre binaire ou une table de hachage lorsqu'il y a une infinité de constructeurs.

On peut par ailleur détecter les cas redondants lorsqu'une action n'apparaît pas dans le code produit, et détecter les filtrages non exhaustifs lorsque *error* apparaît dans le code produit.

Neuvième partie
Cours 08/12

27 Phases du Compilateur

On décompose la production de code en plusieurs phases :

1. Sélection d'Instructions
2. Register Transfer Language
3. Explicit Register Transfer Language
4. Location Transfer Language
5. Code Linéarisé (assembleur)

Le point de départ est l'arbre de syntaxe abstraite issu du typage. Cet architecture de compilateur est valable pour tous les grands paradigmes de programmation. On va l'illustrer sur un fragment du langage C.

On se donne un langage mini-C avec :

— Des entiers

— Des structures allouées sur le tas avec `malloc` (uniquement des pointeurs sur ces structures et pas d'arithmétique de pointeurs)

— Des fonctions

— Une primitive pour afficher un entier.

On suppose l'analyse sémantique effectuée et le résultat :

```
type file = { gvars: decl_var list; funs: decl_fun list; }
and decl_var = typ * ident
and decl_fun = {
    fun_typ: typ; fun_name: ident;
    fun_formals: decl_var list; fun_body: block; }
and block = decl_var list * stmt list
and stmt =
    | Sskip
    | Sexpr of expr
    | Sif of expr * stmt * stmt
    | Swhile of expr * stmt
    | Sblock of block
    | Sreturn of expr
    | Sprintf of expr
and expr = { expr_node: expr_node; expr_typ: typ }
and expr_node =
    | Econst of int32
    | Eaccess_local of ident
    | Eassign_local of ident * expr
    | Eaccess_global of ident
    | Eassign_global of ident * expr
    | Eaccess_field of expr * int (* indice du champ *)
    | Eassign_field of expr * int * expr
    | Eunop of unop * expr (* - ! *)
    | Ebinop of binop * expr * expr (* + - == etc. *)
    | Ecall of ident * expr list
    | Emalloc of structure
```

28 Phase 1 : Sélection d'Instructions

Son objectif est de :

1. Remplacer les opérations arithmétiques du C par celles de `x86-64`
2. Remplacer les accès aux champs de structures par des opérations `mov`

28.1 Opérations Arithmétiques

On pourrait traduire naïvement chaque opération arithmétique de C par l'instruction correspondante de `x86-64`. Cependant, ce langage fournit des instructions permettant une plus grande efficacité, notamment :

— L'addition d'un registre et d'une constante

— Le décalage des bits vers la gauche ou la droit, correspondant à une multiplication ou à une division par une puissance de deux.

— La comparaison d'un registre avec une constante.

D'autre part, il est souhaitable d'évaluer autant d'expressions que possible pendant la compilation. On se donne de nouveaux arbres pour le résultat de la sélection d'instructions. L'objectif étant de traduire les précédents arbres. Les opérations sont maintenant celles de `x86-64` :

```
    type munop = Maddi of int32 | Msetei of int32 | ...
type mbinop = Mmov | Madd | Msub ... | Msete | Msetne ...
type expr =
| Emunop of munop * expr (* remplace Eunop *)
| Embinop of mbinop * expr * expr (* remplace Ebinop *)
| Eand of expr * expr
| Eor of expr * expr
| ...
```

Pour réaliser l'évaluation partielle on va utiliser des *smart constructors*. On introduit une fonction `mk_add` qui effectue d'éventuelles simplifications et se comporte comme le constructeur sinon :

```
let rec mk_add e1 e2 = match e1, e2 with
| Econst n1, Econst n2 ->
    Econst (Int32.add n1 n2)
| e, Econst 0l | Econst 0l, e ->
    e
| Emunop (Maddi n1, e), Econst n2
| Econst n2, Emunop (Maddi n1, e) ->
    mk_add (Econst (Int32.add n1 n2)) e
| e, Econst n | Econst n, e ->
    Emunop (Maddi n, e)
| _ ->
    Embinop (Madd, e1, e2)
```

Deux aspects sont essentiels concernant ces simplifications :

— La sémantique des programmes doit être préservée.

— La fonction de simplification doit terminer.

La traduction se fait alors mot à mot, et c'est un morphisme pour les autres constructions :

```
let rec expr e = match e.Ttree.expr_node with
| Ttree.Ebinop (Badd, e1, e2) ->
    mk_add (expr e1) (expr e2)
| Ttree.Ebinop (Bsub, e1, e2) ->
    mk_sub (expr e1) (expr e2)
| Ttree.Eunop (Unot, e) ->
    mk_not (expr e)
| Ttree.Eunop (Uminus, e) ->
    mk_sub (Econst 0l) (expr e)
| ...
```

28.2 Accès à la Mémoire

Une adresse mémoire est donnée par une expression et un décalage.
Ici, on transforme en accès à la mémoire les accès aux champs de structures. On adopte un schéma simple où chaque champ occupe exactement un mot. Pour le reste, rien à signaler, on en profite simplement pour oublier les types et regrouper les variables locales dans la fonction qui les a introduit :

```
type expr =
    | ...
    | Eload of expr * int
```

```
        | Estore of expr * int * expr

    let rec expr e = match e.Ttree.expr_node with
        | ...
        | Ttree.Eaccess_field (e, n) ->
            Eload (expr e, n * word_size)
        | Ttree.Eassign_field (e1, n, e2) ->
            Estore (expr e1, n * word_size, expr e2)

    type deffun = {
        fun_name : ident;
        fun_formals: ident list;
        fun_locals : ident list;
        fun_body : stmt list;
    }
        type file = {
        gvars: ident list;
        funs : deffun list;
}
```

29 RTL : Register Transfer Language

Ici, l'objectif est de :

— Détruire la structure arborescente des expressions et des instructions au profit d'un graphe de flot de contrôle, qui facilitera la suite. On supprime en particulier la distinction entre expressions et instructions.

— Introduire des pseudo-registres pour représenter les calculs intermédiaires. Ces pseudo-registres sont en nombre illimité et deviendront des registres ou des emplacements de pile.

29.1 Représentation

On se donne un module pour les pseudo-registres et un module pour des étiquettes représentant les sommes du graphe de flot de contrôle :

```
type t
val fresh: unit -> t
module S: Set.S with type elt = t
type set = S.t
module M: Map.S with type key = t
type 'a map = 'a M.t
type graph = instr Label.map
```

Inversement, chaque instruction RTL indique quelle est l'étiquette suivante : `Econstr (n, r, l)` signifie : charger n dans le pseudo-registre n et transférer les contrôle à l'étiquette l. Enfin, les opérations arithmétiques manipulent maintenant des pseudo-registres. Pour construire le graphe de flot on le stocke dans une référence et on se donne une fonction d'ajout d'instruction.

29.2 Traduction des Expressions

On traduit les expressions grâce à une fonction qui prend en arguments :

— Le registre de destination de la valeur de l'expression

— L'expression à traduire

— L'étiquette de sortie

et revoie l'étiquette d'entrée du calcul.

La traduction est relativement aisée, quitte à introduire des pseudo-registres :

```
let rec expr destr e destl = match e with
    | Istree.Econst n ->
        generate (Econst (n, destr, destl))
    | Istree.Embinop (op, e1, e2) ->
let tmp2 = Register.fresh () in
    expr destr e1 (
    expr tmp2 e2 (generate (
    Embinop (op, tmp2, destr, destl))))
```

— Pour les variables locales, on se donne une table indiquant quel pseudo-registre est associé à chaque variable :

```
        | Istree.Eaccess_local x ->
let rx = Hashtbl.find locals x in
generate (Embinop (Mmov, rx, destr, destl))
```

— Pour traduire les opérateurs `&&` et `||` et les instructions `if, while` il nous faut des instructions RTL de branchement :

```
        type instr =
...
| Emubranch of mubranch * register * label * label
| Embbranch of mbbranch * register * register
* label * label
| Egoto of label

type mubranch = Mjz | Mjnz | Mjlei of int32 | ...
type mbbranch = Mjl | Mjle | ...
```

On traduit alors les conditions ainsi :

```
        let rec condition e truel falsel = match e with
| Istree.Eand (e1, e2) ->
condition e1 (condition e2 truel falsel) falsel
| Istree.Eor (e1, e2) ->
condition e1 truel (condition e2 truel falsel)
| Istree.Embinop (Mjle, e1, e2) ->
let tmp1 = Register.fresh () in
let tmp2 = Register.fresh () in
expr tmp1 e1 (
expr tmp2 e2 (generate (
Embbranch (Mjle, tmp2, tmp1, truel, falsel))))
| e ->
let tmp = Register.fresh () in
expr tmp e (generate (
Emubranch (Mjz, tmp, falsel, truel)))
```

On pourrait traiter plus de cas particuliers.

— Pour traduire `return`, on se donne un pseudo-registre pour recevoir le résultat de la fonction et une étiquette `exitl` correspondant à la sortie de la fonction. Sinon, on se donne une étiquette correspondant à la suite du calcul.

```
        let rec stmt retr s exitl destl = match s with
            | Istree.Sskip ->
                destl
            | Istree.Sreturn e ->
                expr retr e exitl
```

```
        | Istree.Sif (e, s1, s2) ->
            condition e
                (stmt retr s1 exitl destl)
                (stmt retr s2 exitl destl)
```

— Pour une boucle `while`, on crée une boucle dans le graphe. Ceci peut se faire directement :

```
        | Istree.Swhile (e, s) ->
let l = Label.fresh () in
let entry = condition e (stmt retr s exitl l) destl in
graph := Label.M.add l (Egoto entry) !graph;
entry
```

ou en l'écrivant comme un opérateur de point fixe :

```
        let loop f =
let l = Label.fresh () in
let entry = f l in
graph := Label.M.add l (Egoto entry) !graph;
entry
(*Avec*)
| Istree.Swhile (e, s) ->
loop (fun l -> condition e (stmt retr s exitl l) destl)
```

— Les paramètres d'une fonction et son résultat sont désormais maintenus dans des pseudo-registres. La traduction d'une fonction se compose de 4 étapes :

1. On alloue des pseudo-registres frais pour ses arguments, son résultat et ses variables locales.
2. On part d'un graphe vide
3. On crée une étiquette fraîche pour la sortie de la fonction.
4. On traduit le corps de la fonction dans RTL et le résultat est l'étiquette d'*entrée* de la fonction

30 ERTL : Explicit Register Transfer Language

On cherche à expliciter les conventions d'appel, ici :

— Les six premiers arguments sont passés dans `%rdi, %rsi, %rdx, %rcx, %r8, %r9` et les suivants sur la pile.

— Le résultat est renvoyé dans `%rax`.

— Les registres *callee-saved* doivent être sauvegardés par l'appelé (`%rbx, %r12, %r13, %r14, %r15, %rbp`)

— La division `idivq` impose dividende et quotient dans `%rax`

— `malloc, printf` sont des fonctions de bibliothèques, avec argument dans `%rdi` et résultat dans `%rax`.

30.1 Formalisation

On suppose que le module `Register` décrit aussi les registres physiques. Dans ERTL, il ne reste plus dans les opérations d'appel de fonction que le nom de la fonction à appeler, car de nouvelles instructions vont être insérées pour charger les arguments dans des registres et sur la pile, et pour récupérer le résultat dans `%rax`. On conserve tout de même le nombre de paramètres passés dans des registres. Les autres instructions sont inchangées, mais on ajoute des instructions :

— Pour allouer et désallouer le tableau d'activation

— Pour lire/écrire sur la pile

— Pour rendre le retour explicite.

```
type t
...
val parameters: t list (* pour les n premiers arguments *)
val rax: t (* pour résultat, division *)
val callee_saved: t list
val rdi: t (* pour malloc et printf *)

| Ecall of ident * int * label

| Ealloc_frame of label
| Edelete_frame of label

| Eget_param of int * register * label
| Epush_param of register * label

| Ereturn
```

On ne change pas la structure du graphe de flot de contrôle. On se contente d'insérer de nouvelles instructions :

— Au début de chaque fonction, pour :
  1. Allouer le tableau d'activation
  2. Sauvegarder les registres *callee-saved*
  3. Copier les arguments dans les pseudo-registres correspondants

— A la fin de chaque fonction, pour :
  1. Copier le pseudo-registre contenant le résultat dans `%rax`
  2. Restaurer les registres *callee-saved*
  3. Désallouer le tableau d'activation

— A chaque appel, pour
  1. Copier les pseudo-registres contenant les arguments dans les registres d'entrée/sur la pile
  2. Copier le registre de résultat dans le pseudo-registre contenant le résultat après l'appel.

30.2 Traduction

Pour traduire les instructions de RTL vers ERTL, il y a peu de changement, si ce n'est concernant la division, les appels, les malloc et les printf :

```
| Rtltree.Emalloc (r, n, l) ->
    Econst (n, Register.rdi, generate (
    Ecall ("malloc", 1, generate (
    Embinop (Mmov, Register.rax, r, l)))))
| Rtltree.Eprintf (r, l) ->
    Embinop (Mmov, r, Register.rdi, generate (
    Ecall ("print_int", 1, l)))
| Rtltree.Embinop (Mdiv, r1, r2, l) ->
    Embinop (Mmov, r2, Register.rax, generate (
    Embinop (Mdiv, r1, Register.rax, generate (
    Embinop (Mmov, Register.rax, r2, l)))))
```

Pour ce qui est de l'appel, on commence par associer les premiers paramètres aux registres physiques correspondants :

```
let assoc_formals formals =
    let rec assoc = function
        | [] , _ -> [], []
```

```
        | rl , [] -> [], rl
        | r :: rl, p :: pl -> let a, rl = assoc (rl, pl) in
                              (r, p) :: a, rl
    in
    assoc (formals, Register.parameters)


let move src dst l = generate (Embinop (Mmov, src, dst, l))
let push_param r l = generate (Epush_param (r, l))
let pop n l =
if n = 0 then l else generate (Emunop (Maddi n, rsp, l))

| Rtltree.Ecall (r, x, rl, l) ->
    let frl, fsl = assoc_formals rl in
    let n = List.length frl in
    let l = pop (word_size * List.length fsl) l in
    let l = generate (Ecall (x, n, move rax r l)) in
    let l = List.fold_left (fun l t -> push_param t l) l fsl in
    let l = List.fold_right (fun (t, r) l -> move t r l) frl l in
    Egoto l
```

Il reste à traduire chaque fonction. On associe un pseudo-registre à chaque registre physique qui doit être sauvegardé :

```
let deffun f =
    graph := ...on traduit chaque instruction...
    let savers =
        List.map (fun r -> Register.fresh(), r) callee_saved in
    let entry = fun_entry savers f.Rtltree.fun_formals
        f.Rtltree.fun_entry in
    fun_exit savers f.Rtltree.fun_result f.Rtltree.fun_exit;
    { fun_name = f.Rtltree.fun_name;
    ...
    fun_body = !graph; }
```

Ensuite, on ajoute les instructions à l'entrée et à la sortie de la fonction :

```
let fun_entry savers formals entry =
    let frl, fsl = assoc_formals formals in
    let ofs = ref word_size in
    let l = List.fold_left
        (fun l t -> ofs := !ofs + word_size; get_param t !ofs l)
        entry fsl in
    let l = List.fold_right (fun (t, r) l -> move r t l) frl l in
    let l = List.fold_right (fun (t, r) l -> move r t l) savers l in
    generate (Ealloc_frame l)

let fun_exit savers retr exitl =
    let l = generate (Edelete_frame (generate Ereturn)) in
    let l = List.fold_right (fun (t, r) l -> move t r l) savers l in
    let l = move retr Register.rax l in
    graph := Label.M.add exitl (Egoto l) !graph
```

30.3 Imperfection

En appliquant les étapes ci-dessus, on est encore loin d'obtenir un bon code `x86-64`. En effet :

— L'allocation de registres tâchera d'associer des registres physiques aux pseudo-registres de manière à limiter l'usage de la pile mais aussi de supprimer certaines instructions.

— Le code n'est pas encore organisé linéairement.

Dans le cadre des fonctions récursives, c'est aussi à cette étape qu'il faut réaliser l'optimisation des appels terminaux. En effet, les instructions à produire ne sont pas les mêmes et ce changement aura une influence dans la phase suivante d'allocation de registres. Il y a une difficulté cependant, si la fonction appelée par un appel terminal n'a pas le même nombre d'arguments passés sur la pile ou de variables locales, car le tableau d'activation doit être modifié. On peut alors :

— Limiter l'optimisation de l'appel terminal aux cas où le tableau d'activation n'est pas modifié (c'est le cas notamment s'il s'agit d'un appel terminal d'une fonction récursive à elle-même)

— Faire de sorte que l'appelant modifie le tableau d'activation et transfère le contrôle à l'appelé *après* l'instruction de création de son tableau d'activation.

31 LTL : Location Transfer Language

L'objectif est ici de faire disparaître les pseudo-registres au profit de registres physiques et d'emplacements de pile.

31.1 Allocation de Registres

On décompose l'allocation de registres en plusieurs étapes :

1. Analyse de durée de vie : on détermine à quels moments précis la valeur d'un pseudo-registre est nécessaire pour la suite du calcul.
2. Construction d'un graphe d'interférence : on crée un graphe indiquant quels sont les pseudo-registres qui ne peuvent pas être réalisés par le même emplacement
3. Allocation de registres par coloration de graphe : on affecte des registres physiques et d'emplacements de pile aux pseudo-registres

Dans la suite, on appelle *variable* un pseudo-registre ou un registre physique :

Définition 31.1.1. *En un point de programme, une variable est dite vivante si la valeur qu'elle contient peut être utilisée dans la suite de l'exécution.*

La propriété *est utilisée* n'étant pas décidable, on se contente de l'approximation *peut être utilisée.*

On attache les variables vivantes aux arêtes du graphe de flot de contrôle. La notion de variable vivante se déduit de :

Définition 31.1.2. *Pour une instruction I du graphe de flot de contrôle, on note :*

— $def(I)$ *les variables définies par cette instruction*

— $use(I)$ *les variables utilisées par cette instruction*

Pour calculer les variables vivantes, il est commode de les associer non pas aux arêtes mais plutôt aux noeuds du graphe de flot de contrôle, c'est à dire aux instructions. On distingue alors :

Définition 31.1.3. *Pour une instruction I du graphe de flot de contrôle, on note :*

— $in(I)$ *les variables vivantes à l'entrée de cette instruction*

— $out(I)$ *les variables vivantes à la sortie de cette instruction*

On obtient :

$$\begin{cases} in(I) = & use(I) \cup (out(I) \setminus def(I)) \\ out(I) = & \bigcup_{s \in succ(I)} in(s) \end{cases}$$

On applique alors le théorème de Tarski pour trouver un plus petit point fixe. En supposant un graphe de flot de contrôle à N sommets et N variables, un calcul brutal a une complexité $\mathcal{O}(N^4)$ dans le pire des cas. On peut améliorer l'efficacité du calcul de plusieurs façons :

— En calculant dans l'ordre inverse du graphe, et en calculant *out* avant *in*.

— En fusionnant les sommets qui n'ont qu'un prédécesseur et qu'un successeur avec ces derniers.

— En utilisant un algorithme plus subtil qui ne recalcule que les valeurs de *in* et *out* qui peuvent avoir changées.

31.2 Algorithme de Kildall

Si $in(I)$ change, il n'est nécessaire de refaire le calcul que pour les prédécesseurs de I. D'où l'algorithme suivant :

```
soit WS un ensemble contenant tous les sommets
tant que WS n'est pas vide
    extraire un sommet l de WS
    old_in <- in(l)
    out(l) <- ...
    in(l) <- ...
    si in(l) est différent de old_in(l) alors
        ajouter tous les prédécesseurs de l dans WS
```

Le calcul des ensemble *def* et *use* est immédiat pour la plupart des instructions. Toutefois, pour les appels, il est un peu plus subtil : on exprime que les $\min(6, n)$ premiers registres de la liste des paramètres vont être utilisés et que tous les registres *caller-saved* peuvent être écrasés. Enfin, pour `return`, `%rax` et les registres *callee-saved* vont être utilisés.

Dixième partie
Cours : 22/12

32 Location Transfer Language

On a déjà fait l'analyse de durée de vie.

32.1 Construction du Graphe d'Interférence

Définition 32.1.1. *On dit que deux variables v_1 et v_2 intergèrent si elles ne peuvent pas être réalisées par le même emplacement.*

Soit une instruction qui définit une variable v : tout autre variable w vivante à la sortie de cette instruction peut interférer avec v. Cependant dans le cas particulier d'une instruction : `mov` w v, on ne souhaite pas déclarer que v et w interfèrent, car il peut être précisément intéressant de réaliser v et w par le même emplacement et d'éliminer ainsi une ou plusieurs instructions.

Définition 32.1.2. *Le graphe d'interférence d'une fonction est un graphe non orienté dont les sommets sont les variables et les sont de deux types : préférence ou interférence.*
Pour chaque instruction qui définit une variable v et dont les variables vivantes en sortie, autres que v, sont $w_1, , w_n$, on procède ainsi :

- *Si l'instruction n'est pas un `mov`, on ajoute les n arêtes d'interférence.*
- *Sinon, on met une préférence sur w et des interférences sur les autres.*

Si on a à la fois une arête de préférence et une d'interférence, on ne conserve que l'interférence.

On représente ce graphe ainsi :
On peut alors voir le problème de l'allocation de registres comme un problème de coloriage de graphe :

- Les couleurs sont les registres physiques
- Deux sommets liés par une arête d'interférence ne peuvent recevoir la même couleur.
- Deux sommets liés par une arête de préférence doivent, autant que possible, recevoir la même couleur.

Les registres physiques doivent être déjà coloriés.
Si un sommet ne peut être colorié, il correspondra à un emplacement de pile. On vide en mémoire le pseudo-registre.

On va colorier le graphe avec des heuristiques pour éviter l'explosion de la complexité. L'Algorithme de George et Appel est une solution.
Soit K le nombre de couleurs. Si un sommet a un degré $< K$, on peut le retirer du graphe, colorier le reste, et on pourra ensuite lui donner une couleur. Cette étape est appelée simplification. Retirer un sommet diminue le degré d'autres sommets et peut donc produire de nouveaux candidats à la simplification.
Lorsqu'il ne reste que des sommets de degré $\geq K$, on en choisit un comme candidat au vidage. On le retire du graphe, on le met sur la pile et le processus de simplification peut reprendre. On choisit un sommet peu utilisé à fort degré.
Lorsque le graphe est vide, on commence le processus de coloration, appelé sélection : On dépile les sommets un à un et :

— S'il s'agit d'un sommet de faible degré, on lui trouve une couleur
— Sinon : Soit il peut être tout de même colorié car ses voisins utilisent moins de K couleurs ; Soit il ne peut pas être colorié et doit être vidé en mémoire.

Enfin, on cherche à utiliser les arêtes de préférence. On utilise pour cela une technique dite de coalescence qui consiste à fusionner des sommets du graphe :

Définition 32.1.3. *Un sommet pseudo-registre v_2 peut être fusionné avec un sommet v_1 si tout voisin de v_1 qui est un registre physique ou de degré $\geq K$ est aussi voisin de v_2.*
De même, un sommet registre physique v_2 peut être fusionné avec un sommet v_1 si tout voisin de v_1 qui est un pseudo-registre ou de degré $\geq K$ est aussi voisin de v_2.

L'algorithme en lui même est alors le suivant : On peut choisir comme fonction de coût :

$$co\hat{u}t(v) = \frac{\text{nombre d'utilisations de } v}{d(v)}$$

Les pseudo-registres vidés en mémoire sont associés à des emplacements sur la pile, dans la zone basse du tableau d'activation, en dessous des paramètres. Plusieurs pseudo-registres peuvent occuper le même emplacement de pile s'ils n'interfèrent pas. On peut à nouveau minimiser le nombre d'emplacements de pile par coloriage de graphes, cette fois avec une infinité de couleurs possibles, avec l'algorithme suivant :

— On fusionne toutes les arêtes de préférence (coalescence).
— On applique ensuite l'algorithme de simplification en choisissant à chaque fois le sommet de degré le plus faible.

Toutefois, il peut se trouver qu'il reste des instructions de la forme `move v v`. On les élimine pendant la traduction vers **LTL**.

32.2 Traduction ERTL vers LTL

Les instructions LTL sont les mêmes que dans ERTL si ce n'est que les registres sont tous des registres physiques ou des emplacements de pile : On traduit chaque instruction ERTL à l'aide d'une fonction qui prend en argument le coloriage du graphe d'une part, et la structure du tableau d'activation d'autre part. Une variable r peut être un registre physique, un pseudo registre réalisé par un registre physique, un pseudo-registre réalisé par un emplacement de pile. Dans tous les cas : Dans d'autre cas, c'est plus compliqué car toutes les opérandes ne sont pas autorisées. C'est le cas de l'accès à une variable globale par exemple. En effet, on ne peut pas `mov` depuis la pile.
On utilise alors un registre intermédiaire. On adopte ici une solution simple : deux registres particuliers seront utilisés comme registres temporaires pour ces transferts avec la mémoire, et ne seront pas utilisés par ailleurs
En pratique, on n'a pas nécessairement le loisir de gâcher ainsi deux registres ; on doit alors modifier le graphe d'interférence et relancer une allocation de registres pour déterminer un registre libre pour le transfert. Heureusement, en pratique, cela converge en 2 ou 3 étapes.

Pour écrire dans la variable r, on se donne une fonction `write`, qui prend également en arguments le coloriage et l'étiquette où il faut aller après l'écriture :

On peut maintenant traduire facilement de ERTL vers LTL :

33 Linéarisation

Il reste une dernière étape : le code est toujours sous la forme d'un graphe de flot de contrôle et l'objectif est de produire du code assembleur linéaire. Plus précisément, les instructions de LTL contiennent une étiquette en cas de test positif et une autre étiquette en cas de test négatif alors que les instructions de branchement de l'assembleur contiennent une unique étiquette pour le cas positif et poursuivent l'exécution sur l'instruction suivante en cas de test négatif.

La linéarisation consiste à parcourir le graphe de flot de contrôle et à produire le code `x86-64` tout en notant dans une table les étiqurttes déjà visitées. Lors d'un branchement, on s'efforce autant que possible de produire le code assembleur naturel si la partie du code correspondant à un test négatif n'a pas encore été visitée. Dans le pire des cas, on utilise un branchement inconditionnel.
On utilise deux tables : une pour les étiquettes déjà visitées, et une pour celles qui devront rester dans le code assembleur. La linéarisation est effectuée par deux fonctions mutuellement récursives : `lin` qui produit le code à partir d'une étiquette donnée s'il n'a pas déjà été produit et une instruction de saut sinon et `instr` qui produit le code à partir d'une étiquette et de l'instruction correspondante sans condition :

Dans le cas d'un branchement, on considère d'abord le cas favorable où le code d'un test négatif n'a pas encore été produit, sinon, il est possible que le code correspondant à un test positif n'ait pas été produit et on peut alors inverser la condition de branchement. Enfin, dans le cas où le code des deux branches est produit, on n'a pas d'autre choix que de produire un branchement inconditionnel.
On rassemble ensuite tout dans un unique programme :

34 Résultats

Notre compilateur est de l'ordre de `gcc -O1`, et contient 1991 lignes.
L'architecture présentée est celle de CompCert où les optimisations sont réalisées au niveau RTL. `gcc` intercale un langage SSA et les optimisations sont réalisées au niveau SSA et au niveau RTL.

34.1 LLVM

Il s'agit d'une infrastructure pour aider à la construction de compilateurs optimisants. LLVM propose un langage intermédiaire, IR, et des outils d'optimisation et de compilation de ce langage. Le langage IR ressemble beaucoup à notre langage RTL, il contient :

— Des pseudo-registres

— Un graphe de flot de contrôle

— Des appels encore haut niveau

Mais il y a aussi des différences notables : c'est un langage typé et le code est en forme SSA : chaque variable n'est affectée qu'une fois. Le code d'origine étant susceptible d'affecter plusieurs fois une même variable. On recourt alors à un opérateur appelé φ pour réconcilier plusieurs branches du flot de contrôle.
L'intérêt de la forme SSA est que l'on peut maintenant attacher une propriété à chaque variable et l'exploiter ensuite partout où cette variable est utilisée.

On peut tirer parti de LLVM pour :

— Ecrire un nouveau compilateur pour un langage S en se contenant d'écrire la partie avant et la traduction vers IR

— Concevoir et réaliser de nouvelles optimisations sur le langage IR.