Rapport de TIPE, version courte pour présentation aux professeurs encadrant

Samuel GARDELLE

Mai 2021

1 Introduction

L'étude de la compilation et des langages de programmation est utile du point de vue de la sécurité. S'assurer de la fiabilité d'un compilateur permet de vérifier qu'il ne déforme pas le sens de ce que le programmeur a souhaité exprimer. Cela est indispensable si l'on veut s'assurer qu'un système informatique fonctionne conformément à la manière dont on pense l'avoir conçu. S'il n'est pas fiable cela peut altérer la disponibilité, la confidentialité ou l'intégrité des données qu'il manipule.

Aussi, la simplicité du compilateur est importante. Pour deux raisons. D'abord parce que l'on peut éprouver plus facilement la fiabilité d'un compilateur simple que d'un compliqué.

Ensuite, parce que utiliser un compilateur ou un exécutable qui nous a été fourni par un tiers requiert obligatoirement qu'on lui fasse confiance. Pour ne pas dépendre d'un telle confiance il est obligatoire de soi-même re-faire son propre compilateur. Les raisons qui soutendent cette affirmation sont détaillés dans *Reflexions On Trusting Trust* de Ken Thomson.

1.1 But

L'informatique et la programmation fonctionelle permettent de mettre en oeuvre par des procédés physiques le calcul de fonctions mathématiques. Ce qui rend difficile ce calcul sont les limitations des machines qui l'effectuent (ordinateurs). Ces machines ne peuvent effectuer qu'un nombre limité d'opérations simples. Il faut donc traduire dans ce langage limité (appelé assembleur) la signification des fonctions mathématiques. Ce processus de traduction est la compilation.
Un programme qui automatise cette traduction est un compilateur. Pour rendre l'automatisation possible, il faut disposer d'un language standart pour exprimer des fonctions mathématiques.

Ce TIPE cherche donc à construire un compilateur simple.

Il faut choisir un langage qui sera compilé en assembleur. Ce langage doit permettre d'exprimer simplement des fonctions mathématiques calculables. On utilisera ici le λ-calcul qui se prette bien à la tâche (cf. Thèse de Church-Turing) dans sa version dite simplement typée. Il sera abrégé STLC. On ajoutera un opérateur d'addition et le type primitif $\mathbb{N}$ dont les termes sont les entiers naturels.

Il faut maintenant trouver une méthode pour le traduire en assembleur.

La théorie des catégories est une théorie qui permet d'étudier certains objets que les autres théories de l'algèbre ne traitent pas. Un tel objet est le STLC. Il est muni d'une structure particulière de catégorie cartésienne fermée (abrégé CCC). *Compiling to Categories* de Conal Elliott utilise déjà cette propriété pour compiler le STLC vers d'autres langages qui ont cette structure de CCC.

L'idée de ce TIPE est de rechercher dans l'assembleur une structure de CCC. Ensuite, une fois cette structure exibée, on peut espérer pouvoir en déduire une compilation/transformation "canonique" au sens où elle préserve cette structure (un foncteur cartésien fermé dans la théorie des catégories).

La définition du STLC et des CCCs sont détaillées dans *Category Theory* de Steve Awodey.

2 Structure de CCC de l'assembleur

L'assembleur que l'on va utiliser est LLVM. On ne peut pas donner de preuve du fait que l'assembleur ai la structure de CCC dans la mesure où LLVM n'est pas décrit par une sémantique formelle. La documentation officielle ne fait que donner un paragraphe explicatif pour chaque instruction qui fait office de "sémantique". Je décris ici la construction de la "catégorie assembleur" et avance des arguments justifiant sa structure de CCC.

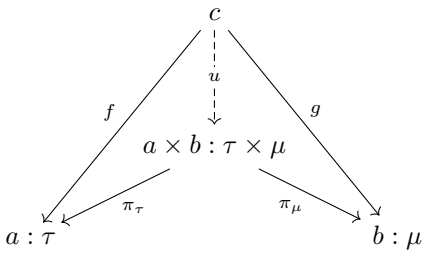
Les objets de la "catégorie de l'assembleur" sont des noms de *registre* (r) munis d'un type (τ) (*typed register* dans la terminologie LLVM) . On les note $r : \tau$. On suppose qu'il y en a une infinité.

Les flèches sont des procédures (des suites d'instructions) qui supposent l'existence d'une valeur d'entrée (resp. de sortie) présente dans leur registre domaine (resp. codomaine).

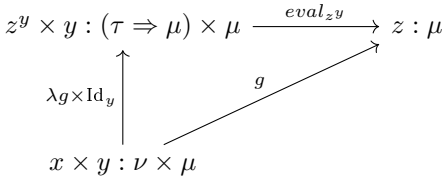
La composition de $g : (b : \mu) \to (c : \nu)$ et $f : (a : \tau) \to (b : \mu)$ est la flèche $g \circ f : (a : \tau) \to (c : \nu)$ où $g \circ f$ est la procédure concaténée de f puis g. Pour se convaincre du caractère associatif on comprend bien que la concaténation est associative (comme le monoide libre). L'identité est la procédure qui ne fait rien (no-op).

On a donc définit une catégorie. Exhibons son caractère cartésien puis fermé. On pourra se référer aux diagrammes commutatifs suivants qui décrivent les propriétés universelles recherchées.

Propriété universelle du produit binaire :



Propriété universelle de l'exponentielle :



Le produit de deux objets $a : \tau$ et $b : \mu$ est l'objet $a \times b : \tau \times \mu$ où $a \times b$ est un nouveau registre qui contient une paire de τ et μ. La projection π_τ (resp π_μ) est la procédure qui extrait la première composante (resp. seconde composante) de la paire et la place dans a (resp b).

Avant d'introduire l'exponentielle, il faut mentionner que LLVM permet la définition et l'appel de fonctions. On supposera que toutes les fonctions que l'on utilisera sont déjà définies dans le programme. La tâche incombe au compilateur de détecter quelles fonctions sont utilisées et de les insérer avant le reste du programme.

On dira qu'une fonction formelle est un couple contenant une adresse vers une fonction LLVM et l'adresse d'un pointeur (qui fait office de contexte comme pour une clôture lexicale). L'évaluation d'une fonction formelle est l'appel de cette fonction avec son paramètre mais aussi la valeur du pointeur. Son type est noté avec une flèche $\Rightarrow$.

Soient $y : \tau$ et $z : \mu$ deux objets : Quelque soit $x : \nu$ un objet, $g : (x \times y : \nu \times \tau) \to (z : \mu)$ une procédure : L'objet exponentiel $z^y : \tau \Rightarrow \mu$ est un registre qui contient une fonction formelle.

La transposée $\lambda g : (x : \nu) \to (z^y : \tau \Rightarrow \mu)$ est la procédure suivante : D'abord elle alloue un pointeur vers la valeur stockée dans le registre x. Ensuite elle écrit dans le registre z^y le couple contenant ce pointeur et l'adresse vers une fonction LLVM. Cette fonction LLVM est celle qui d'abord écrit dans le registre $x \times y$ le couple contenant la valeur de contexte x et celle du registre y. Ensuite cette fonction exécute g puis elle retourne la valeur contenue dans z.

L'évaluation $eval_{z^y} : (z^y \times y : (\tau \Rightarrow \mu) \times \tau) \to (z : \mu)$ est la procédure suivante : Elle lit la valeur du registre $z^y \times y$ et en extrait une fonction formelle et une valeur de type τ. Elle déréférence la valeur contenue dans le pointeur et appelle la fonction avec cette valeur et la valeut de type τ.

La transposée λg est en fait la version currifiée de g et $eval_{z^y}$ l'évaluation d'une fonction formelle.

3 Implémentation

Le fonctionnement du compilateur se décompose en plusieurs phases. La première phase est celle du parseur. Elle a pour tâche de transformer l'écriture d'un programme en un arbre syntaxique qui représente ce programme. La seconde est le calcul du type de ce programme et en même temps la vérification que ce programme est correctement typé. La troisième est la transformation de cet arbre qui représente un terme du STLC en un terme du langage des CCCs. La quatrième est la "génération de code" : c'est la transformation de ce second arbre en une suite d'instructions qui représentent le langage de l'assembleur.

3.1 Parseur monadique

Il y a de nombreuses manières de créer un parseur. Celle que j'ai choisi d'utiliser est connue sous le nom de "parseur monadique" ou "parser combinators". Les monades sont des constructions utilisées en Haskell qui sont particulièrement adaptées pour traiter ce genre de problème. Elles permettent d'écrire des parseurs de manière simple. Le programme ressemble dans sa structure à une définition en EBNF du STLC.

On retrouve dans le fichier *Parsing.hs* une librairie que j'ai fais en m'inspirant de *Parsec : Direct Style Monadic Parser Combinators For The Real World* qui permet de créer de tels parseurs.

L'idée derrière la librairie est de concevoir un parseur qui retourne un élément de type a comme une fonction d'une chaine de caractères vers soit une erreur soit un couple (x, s) où x est l'élément de type a qui a été parsé et s le reste de la chaine de caractère.

```
newtype Parser a =Parser{ parse::String->Either String (a,String) }
```

Les structures de monade, de foncteur, d'applicative et d'alternative qui sont définies dans le fichier permettent de préciser comment on peut composer séquentiellement ou parrallèlement des parseurs et comment promouvoir ("lift") une fonction en une transformation de parseur. Haskell propose des notations pour les monades qui rendent alors naturel l'usage de cette librairie.

Le fichier *Parser.hs* utilise cette librairie dans le cas particulier du STLC.

3.2 Catamorphismes

Le parcours d'arbre ("top-down") se fait en général par des fonctions récursives qui déconstruisent leurs arguments selon chaque constructeur de l'arbre. Souvent, la fonction est appliquée récursivement sur les sous-arbres de chaque constructeur. Un tel algorithme est appelé catamorphisme et peut s'écrire de manière plus générale. Ce qui est spécifique à un catamorphisme particulier n'est alors que la manière dont il assemble les résultats des appels récursifs sur ses sous-arbres : son algèbre.

La librairie *recursion−schemes* propose l'opérateur *cata* qui à une algèbre associe son catamorphisme. Une algèbre étant ici une manière d'assembler les résultats des sous-arbres en un seul résultat.

Exemple : l'évaluation d'un langage composé de l'addition et de constantes.

```
data Term = Add Term Term | Cst Int
data TermF f = AddF f f | CstF int
-- f est un foncteur, type généré automatiquement en pratique

alg :: Algebra Int Int
alg (AddF a b) = a + b
alg (CstF k) = k

eval :: Term -> Int
eval = cata alg
```

Les catamorphismes permettent de grandement simplifier la plupart des parcours de l'arbre syntaxique.

TODO : mentionner (ou pas ?) l'opérateur de point fixe pour les types ?

3.3 Implémentation

Pour ce qui est du fonctionnement du code, il découle directement de la théorie exposée précédement et n'utilise pas d'algorithme particulier ; à l'exception du passage du STLC en CCC appelé "élimination des abstractions" décrit dans *Compiling To Categories* de Conal Elliott.