Implémentation de la théorie des types dépendants

Delgado Roman

February 1, 2016

Abstract

Ce document représente la synthèse de mes premières lectures ainsi que de ma compréhension du sujet. Ce projet consistant en l'implémentation d'un système de type. Il sera donc question de définir les principaux concepts que j'ai pu aborder lors de mes lectures. La deuxième partie se concentre sur une implémentation du lambda calcul qui fait office de travail préliminaire au projet. Tout au long du projet il a été choisi de programmer en Ocaml language de programmation fonctionnelle.

Contents

1 Introduction

1.1 Historique

1.2 Qu'est - ce qu'un système de type

Tout programmeur a un jour fait face à un comportement anormal de son programme. Sans assistance il est très difficile pour un être humain de détecter d'où provient l'erreur, qu'elle soit simplement syntaxique ou d'ordre algorithmique. Pour les erreurs de type syntaxique les éditeurs de texte sont dotés de systèmes de coloration afin d'aider le programmeur à repérer les différents mots clés du

langage et à ne pas commettre d'erreur lors de leur écriture. Par analogie le système de type permet de définir des règles quand à la manipulation des différentes données du programme afin d'éviter le maximum de "bugs" lors de l'exécution du programme. Cette vérification peut survenir lors de la compilation, on parle alors de typage statique. Au contraire si cette vérification est effectuée lorsque le programme "opère" on parle de typage dynamique.

1.3 La théorie des types

2 Lambda Calcul

2.1 Présentation et conventions utilisées

La première étape du projet consiste en l'implémentation du modèle formel inventé par Alonzo Church dans les années 1930 le lambda calcul. Aussi appelé "le plus petit language de programmation du monde" il sert actuellement de meta language. Il peut être typé ou non, nous avons choisi de commencer par sa version non typée pour plus de souplesse. Afin de présenter rapidement le lambda calcul sans entrer dans les détails j'essayerai de présenter celui ci par l'exemple. Dans le lambda calcul non typé on peut définir un lambda terme de la façon suivante

```
lambdaterme :=
| variable x
| application u v (si u et v sont des lambda termes)
| abstraction Λx.y (si x est une variable et y un lambda terme)
```

Le premier problème lors de la représentation des lambda termes réside dans la dissociation des variables libres et des variables liées. Par exemple le terme suivant Λx.x représente la fonction identité et la variable x est liée. Le terme suivant représente aussi la fonction identité Λy.y . Dans le cas présent le nom de la variable ne pose pas de réel problème car le terme est petit mais lorsque l'on exprime des termes d'une taille plus importante. Prenons comme exemple le terme suivant (Λx.x)x qui est totalement équivalent au terme (Λx.x) y , on constate bien le problème de différencier les variables liées et les variables libres. Une des solutions est d'utiliser les indices de de Bruijn qui consiste a représenter les variables liées par des entiers. Le terme précédent s'écrira donc (Λ.0)y , afin de déterminer la valeur des variables liées il faut compter le nombre de "binder" avant celle ci. Le terme Λx. Λy. xy s'écrira donc Λ.Λ.1 0 .

Nous avons donc tous les éléments pour donner notre définition du type lambda term en Ocaml

```
type lambda_term =
| FreeVar of string
| BoundVar of int
| Abs of lambda_term
| Appl of (lambda_term * lambda_term)
```

2.2 Réduction

Un des principe fondamental consiste à reduire les termes. Tout comme il est possible de faire l'analogie entre Λ.0 comme étant la définition de la fonction identité, (Λ.0)y peut se rapporter à l'application de la fonction identité au paramètre y. En language c l'équivalent serait "identité(y);". Pour continuer l'analogie, la réduction se rapporte à executer cette fonction ce qui nous donne dans le cas précédent:

```
(Λ.0)y -> y
```

D'un point de vue plus formel dans le lambda calcul une application de la forme "(Λ...) t " c'est à dire une application dont le membre de gauche est une abstraction et le membre de droite un terme quelconque est appelé redex. Une redex peut donc se réduire grâce à une opération de substitution.

Pour les beta réductions nous allons utiliser la stratégie "call by value" c'est à dire que l'on cherche la redex la plus à gauche et que l'on ne réduit pas les abstractions car on les considère comme des valeurs.

2.3 Implémentation en Ocaml