Implémentation de la théorie des types Dépendants

Delgado roman

February 3, 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 Théorie

1.1 Lambda Calcul non typé

1.1.1 Un modèle de calcul universel

Dans les années 1930, Alonzo church commença son travail sur le lambda calcul. Le lambda calcul est un modèle de calcul universel au même titre que les machines de Turing. A l'aide d'une syntaxe simple il permet d'étudier les propriétés du calcul de façon formelle. Les fonctions ne portent pas de nom et ne sont applicables que à un seul argument. En 1936 Church publie dans son article "An unsolvable problem of elementary number theory" présentant le lambda calcul non typé. Celui ci est très souple car tout est fonction, cela permet donc de se concentrer sur la partie calculatoire.

1.1.2 Syntaxe

On peut définir le lambda calcul non typé de la façon suivante:

```
t::=                    lambda terme
      x                 variable
      Λx.t              abstraction
      t t               application
```

Pour faire une analogie avec les languages de programmation, on peut dire qu'une abstraction est une définition de fonction et que l'application consiste à appliquer une fonction un argument (qui dans le lambda calcul est aussi une fonction). On peut donc définir un lambda terme représentant la fonction identité de la façon suivante Λx.x . Il existe de nombreuses façons de représenter les lambda termes, ici il est question de celle utilisée par Pierce dans son ouvrage “Types and programming languages”

1.1.3 Reduction

Avant de parler de la réduction, il faut pouvoir déterminer si une variable est liée ou libre. Une variable est liée lorsque celle ci apparait à l'intérieur d'une abstraction. Dans le lambda terme suivant Λx.x y la variable x est liée et la variable y est libre. Voici un dernier exemple pour retirer toute ambiguïté, (Λx.Λy. x y)y ici la variable x est liée ainsi que la première occurence de la variable y, cependant la variable y à l'exterieur des parenthèse est libre. Un terme dont l'ensemble de ses variables sont liées est dit clos.

Le fait de réduire un lambda terme peut être vu en suivant la même analogie que précédement comme le fait de calculer celui ci. On ne peut effectuer de réduction que sur les applications dont le membre de gauche est une abstraction et cela consiste à substituer dans le membre de gauche l'ensemble des variables liées par le terme de droite.

On peut donc formaliser une réduction ainsi (Λx.t)y -> t[x->y] . On peut donc utiliser notre fonction identité définie précedement à un terme t de la façon

suivante (Λx.x)t -> t . Un lambda terme sous la forme normale est un terme ou plus aucune reduction n'est possible, certains termes peuvent ne pas avoir de forme normale comme le terme (Λx.xx)(Λx.xx)

1.1.4 Les entiers de Church

Ici il faut faire une transition avec le fait que le lambda calcul non typé correspond au typage dynamique, alors que le lambda calcul est la version typage statique

Le lambda calcul permet de définir beaucoup plus de chose que sa simplicité pourrait laisser croire. Ici nous allons montrer un exemple de programmation a l'aide du lambda calcul, les entiers de Church.

En voici une définition formelle:

```
c0 = Λs.Λz.z
c1 = Λs.Λz.s z
c2 = Λs.Λz.s(s z)
```

Ici c0 correspond à l'entier 0, c1 à l'entier 1 et ainsi de suite. Cette définition est très proche de la définition des entiers de Peano ou l'entier 2 se notterait successeur(successeur(0)). Maintenant voici la défintion de la fonction successeur ainsi que la fonction d'addition dans le lambda calcul:

```
successeur = Λn.Λs.Λz. n s (s z)
plus = Λm.Λn.Λs.Λz. m s (n s z)
```

Ces défintions sont extraites du livre "types and programming languages" de Pierce

1.2 Lambda Calcul simplement typé

Les citations dans cette section sont Church Curry Pierce

Par la suite en 1940 il publie "A Formulation of the Simple Theory of Types" ou apparait le lambda calcul typé.

1.2.1 règles de types

Le lambda calcul non typé permet de formaliser une infinité de .... mais l'absence de type ne lui permet pas de vérifier la validité de ses calculs. Pour tout programmeur il parrait logique que l'expression "if <entier> then <booléens> else <booléens> n'a pas de sens car il faudrait que le résultat de l'expression de la condition soit un booléen et non un entier. C'est pour cette raison que les types sont intérressants car ils permettent de palier à ces erreurs d'éxecution. Nous allons donc nous intérresser à l'implémentation des booléens dans le lambda calcul. Avant toute chose il faut définir la notion de type.

```
T ::=           Type
        Bool    Type booléens
        T->T    Type de fonction
```

/* Ici je vais mettre les règles de dérivation qui sont dans le livre de Pierce */

1.2.2 La securité engendrée par les systèmes de type

Une des notions évoquée par Pierce dans son ouvrage est la sécurité offerte par les systèmes de types quant à la bonne éxecution du programme. En effet lors de la compilation dans un système de type statique les erreurs sont interceptées. Il faut maintenant définir deux notions escentielles,

1.3 Les types dépendants

Dire que c'est plus dynamique que les statiques mais que cela s'effectue quand même lors de la compilation

1.3.1 Définition

Les types dépendants permettent de determiner le type d'un terme a l'aide de la valeur d'un autre terme auquel il est associé,

1.3.2 Règles de typage

Ici mettre des règles de typage que je ne connais pas encore

1.3.3 Exemple

Avec le vecteur faire un exemple

2 Implementation

2.1 Lieurs