

# Implémentation de la théorie des types Dépendants

Delgado roman

February 4, 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.

## Contents

<b>1</b>	<b>Théorie</b>	<b>2</b>
1.1	Lambda Calcul non typé . . . . .	2
1.1.1	Un modèle de calcul universel . . . . .	2
1.1.2	Syntaxe . . . . .	2
1.1.3	Réduction . . . . .	2
1.1.4	Les entiers de Church . . . . .	3
1.2	Lambda Calcul simplement typé . . . . .	3
1.2.1	règles de types . . . . .	3
1.2.2	La sécurité engendrée par les systèmes de type . . . . .	4
1.3	Les types dépendants . . . . .	4
1.3.1	Définition . . . . .	4
1.3.2	Règles de typage . . . . .	5
1.3.3	Exemple . . . . .	5
<b>2</b>	<b>Implementation</b>	<b>5</b>
2.1	Lieux . . . . .	5
2.2	Substitution et réduction . . . . .	6

# 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
$\lambda x.t$	abstraction
$t \ t$	application

Pour faire une analogie avec les langages 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  $\lambda x.x$ . 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 Réduction

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  $\lambda x.x \ y$  la variable  $x$  est liée et la variable  $y$  est libre. Voici un dernier exemple pour retirer toute ambiguïté,  $(\lambda x.\lambda y. \ x \ y)y$  ici la variable  $x$  est liée ainsi que la première occurrence 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 précédemment 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, on appelle cela une redex. Le faite de réduire 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  $(\lambda x.t)y \rightarrow t[x \rightarrow y]$ . On peut donc utiliser notre fonction identité définie précédemment à un terme  $t$  de la façon

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

#### 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 choses que sa simplicité pourrait laisser croire. Ici nous allons montrer un exemple de programmation à l'aide du lambda calcul, les entiers de Church. Comme dans le lambda calcul il n'est possible que d'appliquer une fonction à un argument il va nous falloir appliquer une fonction  $n$  fois pour obtenir le l'entier  $n$ .

En voici une définition formelle:

$$\begin{aligned} c0 &= \lambda s.\lambda z.z \\ c1 &= \lambda s.\lambda z.s\ z \\ c2 &= \lambda s.\lambda z.s(s\ z) \end{aligned}$$

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 noterait  $\text{successeur}(\text{successeur}(0))$ . Maintenant voici la définition de la fonction successeur ainsi que la fonction d'addition dans le lambda calcul:

$$\begin{aligned} \text{successeur} &= \lambda n.\lambda s.\lambda z. n\ s\ (s\ z) \\ \text{plus} &= \lambda m.\lambda n.\lambda s.\lambda z. m\ s\ (n\ s\ z) \end{aligned}$$

Pour la fonction plus,

Ces définitions sont extraites du livre “types and programming languages” de Pierce.

Pour conclure sur le lambda calcul non typé, on peut apparenter cela à un système de type dynamique.

## 1.2 Lambda Calcul simplement typé

En 1940 Alonzo Church publie un article “A Formulation of the Simple Theory of Types” où il définit le lambda calcul simplement typé. Celui ci perd en dynamisme et permet d'effectuer moins de calcul mais possède l'avantage de soulever les problèmes liés à la classification des données.

### 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 paraît 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 intéressants car ils permettent de palier à ces erreurs d'exécution. Nous allons donc nous intéresser à l'implémentation des booléens dans le lambda calcul. Avant toute chose il faut définir la notion de type.

$T ::=$	Type
Bool	Type booléens
$T \rightarrow T$	Type de fonction

Nous allons maintenant utiliser une notation explicitant le typage des variables,  $\lambda x:T.t$  signifie quand dans le terme  $t$  la variable liée  $x$  possède le type  $T$ . Nous allons maintenant définir les règles de typage pour les lambda termes. Pour déduire le type d'une variable à partir de son type est trivial:  $\frac{x:T}{x:T}$  (cette expression se lit en partant du numérateur, supposons une variable  $x$  de type  $T$ , alors le type de la variable  $x$  est  $T$ ). Les abstractions quant à elles nécessitent deux suppositions:  $\frac{x:T1 \ t:T2}{\lambda x:T1.t:T1 \rightarrow T2}$ . En effet si l'on possède le type de la variable  $x$  ainsi que celui du terme  $t$  on peut donc conclure sur le type de l'abstraction qui sera une bijection entre  $T1$  et  $T2$ . En suivant la même logique pour déterminer le type d'une application il nous faut le type de l'abstraction de gauche et le type du terme de droite:  $\frac{t1:T1 \rightarrow T2 \ t2:T1}{t1 \ t2:T2}$ .

### 1.2.2 La sécurité 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 exécution du programme. En effet lors de la compilation dans un système de type statique les erreurs sont interceptées. Il faut maintenant définir deux notions essentielles:

Progress: soit  $t$  un terme correctement typé alors  $t$  est une valeur ou il existe une dérivation telle que  $t \rightarrow t'$

Preservation: si un terme est correctement typé alors si il existe une règle de dérivation pour ce terme, le terme obtenu est lui aussi correctement typé.

Ces deux règles permettent d'affirmer que la dérivation d'un terme correctement typé ne peut pas se terminer sur un état autre qu'un terme équivalent à une valeur.

## 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 déterminer le type d'un terme à l'aide de la valeur d'un autre terme auquel il est associé,

Les types dépendants permettent de caractériser les termes en fonction de leur valeur. Si l'on pose l'expression  $a \Rightarrow b$  dans un typage statique pour déterminer le type de  $b$  on se base sur le type de  $a$  tandis que dans un système

de type dépendant, le type de  $b$  sera déterminé à l'aide de la valeur de  $a$ . Cela offre une souplesse supplémentaire dans l'évaluation des types en acceptant plus de programmes que si l'on utilisait un typage statique. Cependant la sécurité offerte par le système de type n'est pas altérée c'est pour cette raison que les types dépendants sont une bonne alternative.

### 1.3.2 Règles de typage

### 1.3.3 Exemple

Avec le vecteur faire un exemple

## 2 Implementation

Dans cette partie il est question de l'implémentation du lambda calcul non typé en précisant les choix de réalisation. Le langage choisi pour cette implémentation et qui le restera pour l'ensemble de ce projet est Ocaml, un langage fonctionnel dont la syntaxe permet de se rapprocher au plus des définitions mathématiques, ce qui rend le passage des définitions théoriques aux implémentations plus aisées.

### 2.1 Lieurs

Le premier problème auquel j'ai été confronté fut la représentation qu'il fallait donner des variables libres ainsi que des variables liées. D'un point de vue algorithmique lors du parcours d'un terme déterminé si une variable est liée peut s'avérer complexe et nécessiterait de nombreuses opérations supplémentaires. Dans le livre "Types and programming languages" l'auteur choisit de ne plus définir les variables par des caractères mais avec des entiers. C'est ce que l'on appelle les indices de de Bruijn. Prenons un exemple pour comprendre, le terme  $(\lambda x. \lambda y. xy)x$  peut se représenter de cette façon  $(\lambda. \lambda 1 0)2$ . Dans cette représentation les variables sont représentées en fonction du nombre de lieurs les précédant. Ici "y" est représentée par l'entier 0 car elle est liée au premier lieu à sa gauche. Même raisonnement pour la première occurrence de  $x$ , cependant la seconde occurrence de  $x$  est une variable libre elle est donc représentée par une valeur au dessus de toutes les valeurs possibles. Cette représentation permet donc de rendre l'implémentation plus aisée, cependant lorsque l'on souhaite effectuer une réduction il faut introduire une nouvelle opération, le "shift" comme expliqué dans l'ouvrage de Pierce.

Dans un article "A tutorial implementation of dependently typed lambda calculus" il est question de la représentation des variables avec les indices de de Bruijn mais cette fois ci on laisse les variables libres sous la forme de caractère. Le terme précédent s'écrirait donc  $(\lambda. \lambda. 1 \ 0)x$ . Cette notation permet de supprimer cette l'opération de "shift". Voici donc l'implémentation que nous avons choisi des lambda terme en Ocaml.

```

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

```

Afin de s'implifier la lecture des lambda terme voici une fonction permettant de convertir un lambda terme en chaîne de caractère.

```

let rec lambda_term_to_string t =
  match t with
  | FreeVar v -> v
  | BoundVar v -> string_of_int v
  | Abs x -> "[" ^ lambda_term_to_string x
  | Appl (x,y) -> "(" ^ lambda_term_to_string x ^ " " ^ lambda_term_to_string y ^ ")"

```

## 2.2 Substitution et réduction

Il faut maintenant définir l'opération de réduction, comme expliqué dans le chapitre 1.1.3 cela équivaut à substituer les occurrences des variables liées dans l'abstraction de gauche par les termes de droite. J'ai donc choisi de définir tout d'abord une fonction récursive de substitution afin de pouvoir l'utiliser plus tard dans la fonction de réduction. La subtilité de cette fonction réside dans le fait que à chaque fois que l'on rencontre un lieu il faut appeler la fonction substitution mais cette fois ci avec l'indice de la variable liée incrémenté de un.

```

let rec substitution t indice tsub =
  match t with
  | FreeVar v -> FreeVar v
  | BoundVar v -> if v = indice then tsub else BoundVar v
  | Abs x -> Abs(substitution x (indice+1) tsub)
  | Appl (x,y) -> Appl(substitution x indice tsub, substitution y indice tsub)

```

On ne peut appliquer une réduction qu'à un terme dont le membre de gauche est une abstraction. Cette fonction de réduction se contente simplement d'appeler la fonction substitution seulement sur les redex.

```

let reduction t =
  match t with
  | FreeVar v -> FreeVar v
  | BoundVar v -> BoundVar v
  | Abs x -> Abs(x)
  | Appl(Abs(x),y) -> substitution x 0 y
  | Appl(x,y) -> failwith "erreur"

```

Ici les applications n'étant pas considérées comme réductibles on renvoie une erreur. Ce choix d'implémentation sera expliqué par la suite lorsque nous aurons

défini la fonction d'évaluation. Afin de réduire un terme jusqu'à sa forme normale si elle existe peut être considérée comme une opération de normalisation, mais étant donné que certains termes ne peuvent pas se réduire nous avons décidé d'appeler cette fonction évaluation. Il existe de nombreuses stratégies pour calculer les lambda termes, nous avons choisis la stratégie "call by name" (c'est ainsi qu'elle est énoncée dans "Types and programming languages"). Cette stratégie consiste à considérer l'ensemble du lambda terme, si celui ci ....

call by name qui n'autorise pas l'évaluation a l'intérieur des abstractions.