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

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

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 noterait successeur(successeur(0)). Maintenant voici la définition 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)
```

Pour la fonction plus,

Ces défintions 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 parait 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 intéresser à l'implémentation des booléens dans le lambda calcul. Avant toute chose il faut définir la notion de type et de contexte:

```
T ::=            Type
    bool         booléens
    T->T         Type de fonction
Γ::=
     ∅           Contexte vide
     Γ,x:T       Contrainte de type sur la variable
```

Nous allons maintenant utiliser une notation explicitant le typage des variables, λx: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 assez trivial:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

(cette expression se lit en partant du numérateur, supposons une variable x de type T dans le contexte Γ, alors on peut déduire d'après le contexte Γ que le type de la variable x est T).

Il faut maintenant définir les règles de typage pour l'abstraction (R-Type Abs) et l'application (R-Type Appl). Pour les construire il est plus simple de commencer par poser la conclusion puis de chercher les hypothèses nécéssaires dans le contexte. Par exemple pour l'abstraction le type à obtenir sera de la forme A⇒B il faut donc que la variable associée au lambda terme soit de type A et le résultat de type B

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t \ : \ A \to B} \quad \texttt{R-Type Abs}$$

$$\frac{\Gamma \vdash t1 : A \to B \quad \Gamma \vdash t2 : A}{\Gamma \vdash t1\, t2 : B} \ \texttt{R-Type Appl}$$

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 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 essentielles:

Progress: soit t un terme correctement typé alors t est une valeur ou il existe une dérivation telle que t->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 a une valeur.

1.3 Les types dépendants

1.3.1 Définition

Afin d'introduire les types dépendants prenons l'exemple suivant: soit la fonction ayant pour but d'effectuer un "et logique" entre deux vecteurs de même taille. Dans un système de type statique on ne peut pas faire de restriction quant à la taille que feront les deux vecteurs. Grâce à la théorie des types dépendants il est possible de demander explicitement deux vecteurs de taille n et donc d'effectuer le "et logique" sans problème. Les types dépendants permettent donc de caractériser un objet non pas seulement par son type mais aussi par sa valeur et ainsi de créer des règles de typage plus complexes et offrant plus de souplesse que le typage dynamique.

1.3.2 Règles de typage

Dans cette partie il sera question de présenter un système de type dépendant comprenant quelques types de base simple ainsi que leurs règles d'introduction et d'élimination.

Commençons tout d'abord par définir les types de base que nous allons manipuler qui sont: le type booléen (bool), l'ensemble vide (void), l'ensemble unitaire (unit) ainsi que l'implication ($\Rightarrow$).

$$\frac{}{\Gamma \vdash bool \ \in Type} \qquad \frac{}{\Gamma \vdash unit \ \in Type}$$

$$\frac{}{\Gamma \vdash void \ \in Type} \qquad \frac{\Gamma \vdash A \in Type \ B \in Type}{\Gamma \vdash (x : A) \Rightarrow B \ \in Type}$$

Prenons pour exemple le type booléen, la règle d'introduction est assez simple puisque peu importe le contexte "true" et "false" sont des booléens. La règle d'élimination des booléens va être le "if ... then ... else" puisque cette règle permet d'utiliser nos booléens précédement définis. De même pour les autres types voici l'ensemble des règles de typage que nous définiron:

`Introduction`	`Elimination`
$\dfrac{}{\Gamma \vdash true \in bool \qquad false \in bool}$	$\dfrac{\Gamma \vdash t \in bool \ \Gamma \vdash u, v \in A}{\Gamma \vdash if\ t\ then\ u\ else\ v}$
$\dfrac{}{\Gamma \vdash i() \in unit}$	
	$\dfrac{\Gamma \vdash t \in void}{\Gamma \vdash A \in A}$

$$\frac{\Gamma, x : A \vdash b \in B}{\Gamma \vdash \lambda x : B \in (x : A) \Rightarrow B} \qquad \frac{f(a : A) \Rightarrow B \quad s \in A}{f(s) \in B[s/a]}$$

1.3.3 Exemple

Essayons de formaliser l'exemple exprimé lors de la définition de la théorie des types dépendants avec l'aide du langage Agda.

2 Implémentation

Dans cette partie il est question de l'implémentation du lambda calcul non typé en précisant les choix de réalisation. Le language choisi pour cette implémentation et qui le restera pour l'ensemble de ce projet est Ocaml, un language 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 determiné si une variable est liée peut s'avérer complexe et nécéssiterait de nombreuses opérations supplémentaires. Dans le livre "Types and programming languages" l'auteur choisit de ne plus définir les variables par des charactères mais avec des entiers. C'est ce que l'on appel les indices de de Bruijn. Prenons un exemple pour comprendre, le terme (Λx.Λy.xy)x peut se représenter de cette façon (Λ.Λ10)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 lieur à sa gauche. Même raisonnement pour la première occurence de x, cependant la seconde occurence 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, cette notion est aussi évoquée dans la publication "I'am not a number i'am a free Variable". Le terme précédent s'écrirait donc (Λ.Λ.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 lieur 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 de reduction call by name (c'est ainsi qu'elle est énoncée dans "Types and programming languages"). Cette stratégie consite à réduire le membre gauche des redex au maximum mais à ne jamais modifier le membre de gauche. Voici donc une implémentation de cette stratégie:

```
let rec evaluation t =
   match t with
   | FreeVar v -> FreeVar v
   | BoundVar v -> BoundVar v
   | Abs x -> Abs(x)
   | Appl(Abs(x),y) -> evaluation(reduction t)
   | Appl(BoundVar x,y) -> Appl(BoundVar x,y)
   | Appl(FreeVar x,y) -> Appl(FreeVar x,y)
   | Appl(x,y) -> evaluation(Appl(evaluation x, y))
```

Avec l'ensemble de ces fonctions nous avons donc implémenté le lambda calcul non typé. La suite du projet consistera donc comme le sujet l'indique en une implémentation d'un système de type dépendant en Ocaml.