Faculté
des Sciences
Département des Sciences Mathématiques

UMONS
Université de Mons

# Vers un langage typé pour la programmation modulaire

Mémoire réalisé par Danny WILLEMS
pour l'obtention du diplôme de Master en sciences mathématiques

Année académique 2016–2017

**Directeur:** François Pottier
**Co-directeurs:** Christophe Troestler
**Service:** Service d'Analyse Numérique
**Rapporteurs:** Christophe Troestler

Faculté des Sciences • Université de Mons • Place du Parc 20 •
B-7000 Mons

# Remerciements

En premier lieu, je remercie François Pottier pour avoir accepté de me suivre pour ce mémoire et de m'avoir permis d'intégrer l'INRIA Paris pendant toute la durée de celui-ci. Sans nos discussions, ses disponibilités, ses conseils et ses remarques, ce travail n'aurait pas pu être réalisé.

Je remercie également Christophe Troestler pour m'avoir aidé à choisir mon sujet de mémoire ainsi que les conseils quant à la rédaction de ce document.

Je remercie chaque membre de l'équipe Gallium de l'INRIA avec qui j'ai discuté et qui m'ont permis de découvrir de nouveaux domaines dans la recherche informatique, plus ou moins éloigné du sujet de mon mémoire.

Je remercie également Vincent Balat qui m'a permis de découvrir lors de mon stage différents chercheurs dans le domaine de la recherche dans les langages de programmation. Sans ses conseils et son aide, je n'aurais eu l'idée de contacter les membres de l'équipe Gallium afin d'obtenir un sujet.

Ensuite, je tiens à remercier Paul-André Melliès pour, dans un premier temps, m'avoir invité à suivre son cours de lambda-calcul et catégories à l'ENS Ulm qui m'a donné l'envie d'explorer plus en profondeur le lien entre l'informatique théorique et les catégories, et, dans un second temps, pour sa disponibilité et ses conseils lors de la recherche de mon sujet de mémoire.

Entre autres, je remercie chaque personne ayant porté ou portant de l'intérêt à mon travail, ce qui me pousse à continuer d'explorer ce sujet par la suite.

Je remercie aussi les chercheurs et développeurs travaillant sur DOT [1], travail de recherche sur lequel mon travail est basé, pour leurs disponibilités et leurs réponses à mes questions. En particulier, je remercie Nada Amin, dont la thèse est consacrée à DOT, pour ses réponses à mes emails.

Pour finir, je remercie chaque professeur m'ayant suivi pendant ces années d'études.

1. Dependent Object Type

# Table des matières

# Introduction

La programmation modulaire est un principe de développement consistant à séparer une application en composants plus petits appelés *modules*. Le langage de programmation OCaml contient un langage de modules qui permet aux développeurs d'utiliser la programmation modulaire. Dans ce langage de module, un module est un ensemble de types et de valeurs, les types des valeurs pouvant dépendre des types définis dans le même module. OCaml étant un langage fortement typé, les modules possèdent également un type, appelé dans ce cas *signature*.

Bien que les modules soient bien intégrés dans OCaml, une distinction est faite entre le langage de base, contenant les types dits « de bases »comme les entiers, les chaînes de caractères ou les fonctions, et le langage de module. En particulier, le terme *foncteur* est employé à la place de *fonction* pour parler des fonctions prenant un module en paramètres et en retournant un autre. De plus, il n'est pas possible de définir des fonctions prenant un module et un type de base et retournant un module (ou un type de base).

```
module Point2D = struct
  type t = { x : int ; y : int }
  let add = fun p1 -> fun p2 ->
    let x' = p1.x + p2.x in
    let y' = p1.y + p2.y in
    { x = x' ; y = y'}
end;;
(* Signature (type) de Point2D
module Point2D : sig
  type t = { x : int; y : int; }
  val add : t -> t -> t
end
*)
```

Code OCaml 1: Exemple d'un module nommé Point2D contenant un type t pour réprésenter un point par ses coordonnées cartésiennes dans un enregistrement et d'une fonction add retournant un point dont les coordonnées sont la somme de deux points donnés en paramètres.

D'un autre côté, dans les types de bases d'OCaml se trouvent les *enregistrements*. Ces derniers sont des ensembles de couples *(label, valeur)*, et ressemblent aux modules. Cependant, la différence majeure entre eux se situent dans la possibilité de définir des types dans un module.

```
module MakePoint2D
    (T : sig type t val add : t -> t -> t end) =
  struct
    type t = { x : T.t ; y : T.t }
    let add = fun p1 -> fun p2 ->
      let x' = T.add p1.x p2.x in
      let y' = T.add p1.y p2.y in
      { x = x' ; y = y'}
  end;;
(* Signature de MakePoint2D
module MakePoint2D :
  functor (T : sig type t val add : t -> t -> t end) ->
    sig type t = { x : T.t; y : T.t; } val add : t -> t -> t end
*)
```

Code OCaml 2: MakePoint2D est un foncteur qui permet de rendre polymorphique notre module Point2D.

Ce mémoire vise à définir, dans un premier temps, un calcul typé dans lequel le langage de modules est confondu avec le langage de base grâce aux enregistrements. Cette unification implique que les modules (et in fine les foncteurs) sont des citoyens de première classes, c'est-à-dire que nous pouvons les manipuler comme tout autre terme, ce qui n'est pas le cas actuellement en OCaml. Dans un second temps, de fournir une implémentation en OCaml des algorithmes de typage et de sous-typage et d'un langage de surface qui nous permet d'écrire des programmes dans ce langage.

Les chapitres sont organisés afin de comprendre la construction d'un tel calcul à partir du plus simple des calculs, le $\lambda$-calcul.

Dans le chapitre 1, nous présentons *le $\lambda$-calcul non typé*, un calcul minimal qui contient des termes pour les variables, pour les abstractions (afin de représenter des fonctions) et des applications (afin de représenter l'application d'une fonction à un paramètre). Nous discuterons également de la sémantique que nous attribuons à ce calcul.

Dans le chapitre 2, nous introduisons la notion de type et nous l'appliquons au $\lambda$-calcul, ce qui nous donnera *le $\lambda$-calcul simplement typé*. Nous discuterons de la notion de *sureté du typage* à travers *les théorèmes de préservation et de progression* que nous démontrerons pour ce calcul typé.

Dans les chapitres 3, 4 et 5, nous enrichissons le $\lambda$-calcul simplement typé avec la notion de polymorphisme qui permet d'attribuer plusieurs types à un terme. Le chapitre 3 se concentre sur *le polymorphisme avec sous-typage*, illustré avec les enregistrements. Nous parlerons aussi de l'implémentation d'un algorithme de sous-typage pour ce calcul et nous montrerons qu'un travail est nécessaire pour passer de la théorie à l'implémentation. Dans le chapitre 4, nous parlons de *polymorphisme paramétré* qui, combiné au $\lambda$-calcul simplement typé, forme le calcul appelé *System F*. Le chapitre 5 se charge de combiner ces deux notions de polymorphismes dans un calcul appelé *System* $F_{<:}$. Une preuve des théorèmes de préservation et de progression seront donnés pour les calculs définis dans les chapitres 3 et 4.

Dans le chapitre 6, nous parlerons de la notion de type récursif et nous

```
module Point2DInt = MakePoint2D (Int64);;

(* Signature de Point2DInt
module Point2DInt :
  sig
    type t = MakePoint2D(Int64).t = { x : Int64.t; y : Int64.t; }
    val add : t -> t -> t
  end
*)
Point2DInt.add
  { x = Int64.of_int 5 ; y = Int64.of_int 5 }
  { x = Int64.of_int 5 ; y = Int64.of_int 5 };;
(* Type
  Point2DInt.t = {Point2DInt.x = 10L; y = 10L}
*)
```

Code OCaml 3: Application de notre foncteur au module des entiers.

étudierons *le λ-calcul simplement typé avec type récursif.*

Ensuite, dans le chapitre 7, nous compléterons les enregistrements définis dans le chapitre 3 avec les *types chemins dépendants* qui offre la possibilité d'ajouter des types dans les enregistrements, la syntaxe manquante pour une convergence entre enregistrements et modules. Ce dernier chapitre comportera en plus des types chemins dépendants, chaque notion étudiée précédemment, c'est-à-dire le λ-calcul simplement typé, les types récursifs, les enregistrements, le polymorphisme par sous-typage et le polymorphisme paramétré.

Pour finir, dans le chapitre 8, nous discuterons de l'implémentation du langage RML[5] qui comprend un algorithme de typage et de sous-typage ainsi que d'un langage de surface pour le calcul défini dans le chapitre précédent. Nous donnerons des comparaisons entre OCaml et RML pour

La principale difficulté de ce travail se trouve dans l'étude des types chemins dépendants, sujet de recherche récent et moins bien compris que les calculs comme *System F* ou *System* $F_{<:}$, ainsi que la gestion de ceux-ci dans les algorithmes.

# Chapitre I

# $\lambda$-calcul non typé

Dans ce chapitre, nous allons introduire les bases théoriques de la programmation fonctionnelle en parlant du $\lambda$-calcul non typé. Nous discutons de la syntaxe de ce langage (les termes) pour ensuite discuter de la réduction de ceux-ci à travers la $\beta$-réduction.

## I.1 Syntaxe

**Définition I.1** (Syntaxe du $\lambda$-calcul)**.** *Soit $V$ un ensemble infini dénombrable. On note $\Lambda$, appelé **l'ensemble des $\lambda$-termes**, le plus petit ensemble tel que :*

1. $V \subseteq \Lambda$
2. $\forall u, v \in \Lambda, uv \in \Lambda$
3. $\forall x \in V, \forall u \in \Lambda, \lambda x.\, u \in \Lambda$

Un élément de $\Lambda$ est appelé un **$\lambda$-terme**, ou tout simplement un **terme**. Un $\lambda$-terme de la forme $uv$ est appelé **application** car l'interprétation donnée est une fonction $u$ évaluée en $v$. Un $\lambda$-terme de la forme $\lambda x.\, u$ est appelé **abstraction**, le terme $u$ étant appelé le **corps**, et est interprété comme la fonction qui envoie $x$ sur $u$.

La plupart des ensembles que nous définirons seront définis de manière récursive comme ci-dessus. Pour des raisons de facilité d'écriture, la syntaxe

$$\Lambda ::= V \mid \Lambda\Lambda \mid V\Lambda$$

ou encore

$$\begin{array}{lr} t ::= & \text{term} \\ \quad x & \text{var} \\ \quad t\,t & \text{app} \\ \quad \lambda x.\, t & \text{abs} \end{array}$$

où $x$ parcourt l'ensemble des variables $V$ et $t$ l'ensemble des termes, sont utilisées pour définir ces ensembles. La dernière syntaxe sera celle que nous utiliserons tout le long de ce document car elle permet une visualisation simple de la syntaxe des termes et permet de nommer chaque forme facilement.

Des exemples de $\lambda$-termes sont

— la fonction identité : $\lambda x.\, x$
— la fonction constante en $y$ : $\lambda x.\, y$
— la fonction qui renvoie la fonction constante pour n'importe quelle variable : $\lambda y.\, \lambda xy.$.
— l'application identité appliquée à la fonction identité : $(\lambda x.\, x)(\lambda y.\, y)$

Comme le montrent les deux derniers exemples, des parenthèses sont utilisées pour délimiter les termes.

Il est également possible de définir des fonctions à plusieurs paramètres à travers la curryfication : une fonction prenant 2 paramètres sera représentée par une fonction qui renvoie une fonction. Par exemple, $\lambda x.\, \lambda y.\, xy$ est une fonction qui attend un paramètre $x$ retournant une fonction qui attend un paramètre $y$, mais elle peut aussi être interprétée comme une fonction à deux paramètres $x$, $y$.

Comme dans une formule mathématique, il est important de différencier les variables libres et les variables liées d'un $\lambda$-terme. Par exemple, dans le $\lambda$-terme $\lambda x.\, x$ la variable $x$ est liée par un $\lambda$ [1] tandis que dans l'expression $\lambda x.\, y$ la variable $y$ est libre. Nous définissons récursivement l'ensemble des variables libres et l'ensemble des variables liées à partir des variables, des abstractions et des applications.

**Définition I.2** (Ensemble de variables libres)**.** *L'ensemble des variables* ***libres*** *d'un terme $t$, noté $FV(t)$ est défini récursivement sur les générateurs de $\Lambda$ par :*

- $FV(x) = \{x\}$
- $FV(\lambda x.\, t) = FV(t) \setminus \{x\}$
- $FV(uv) = FV(u) \cup FV(v)$

**Définition I.3** (Ensemble de variables liées)**.** *L'ensemble des variables* ***liées*** *d'un terme $t$, noté $BV(t)$ est défini récursivement sur les générateurs de $\Lambda$ par :*

- $BV(x) = \emptyset$
- $BV(\lambda x.\, t) = BV(t) \cup \{x\}$
- $BV(uv) = BV(u) \cup BV(v)$

Il existe également des termes qui sont syntaxiquement différents, mais que nous voudrions naturellement qu'ils soient les mêmes. Par exemple, nous voudrions que la fonction identité $\lambda x.\, x$ ne dépende pas de la variable liée $x$, c'est-à-dire que les termes $\lambda x.\, x$ et $\lambda y.\, y$ soient un seul et unique terme : la fonction identité. Cette égalité se résume à une substitution de la variable $x$ par la variable $y$, ou plus généralement par un terme $u$.

Avant de donner une définition exacte, il est important de remarquer que la substitution n'est pas une action triviale si nous ne voulons pas changer le sens des termes. Si nous effectuons une substitution purement syntaxique, nous pouvons alors obtenir des termes qui ne sont plus dans la syntaxe des éléments de $\Lambda$. Par exemple, si nous substituons toutes les occurences de $x$ par un terme $u$ dans la fonction constante $\lambda x.y$, nous aurions $\lambda u.y$, qui n'a pas de sens car $u$ n'est pas obligatoirement une variable.

La définition doit aussi prendre en compte les notions de variables liées et libres. En effet, si nous prenons la fonction constante $\lambda x.y$ et que nous substituons $y$ par $x$ uniquement dans le corps de la fonction, nous obtenons $\lambda x.x$, qui n'a pas le même sens que $\lambda x.y$. Cet exemple nous montre que nous devons faire

1. on dit aussi qu'elle est « sous »un $\lambda$.

attention lorsque la variable à substituer, dans ce cas $x$, est liée dans le terme où se passe la substitution (ici $\lambda x.y$).

Un autre exemple où la substitution n'est pas évidente est la substition de la variable $z$ du terme $\lambda x.z$ (la fonction constante en $z$) par le terme $\lambda y.x$ (la fonction constante en $x$). Après subtitution, nous nous retrouvons avec le terme $\lambda x.\lambda y.x$, c'est-à-dire la fonction qui renvoie la fonction constante pour le paramètre donné. Ce dernier exemple nous montre que nous devons également faire attention aux variables libres du terme substituant.

**Définition I.4** (Substitution de variable par un terme)**.** *Soit $x \in V$ et soient $u, v \in \Lambda$. On dit que la variable $x$ est **substituable par** $v$ **dans** $u$ si et seulement si $x \notin BV(u)$ et $FV(v) \cap BV(u) = \emptyset$.*

Nous définissons alors la fonction de substitution d'une variable $x$ par un terme $v$ dans un terme $u$.

**Définition I.5** (fonction de subtitution)**.** *Soient $x$ une variable et $u, v \in \Lambda$ tel que $x$ est substituable par $v$ dans $u$. On définit récursivement la fonction de substitution, notée $u[x := v]$, par :*

- $x[x := v] = v$
- $y[x := v] = y$ *(si $y \neq x$)*
- $(u_1 u_2)[x := v] = (u_1[x := v])(u_2[x := v])$
- $(\lambda y.u)[x := v] = \lambda y.(u[x := v])$

*$u[x := v]$ se lit $x$ est substitué par $v$ dans $u$.*

Nous définissons maintenant une relation, appelée relation d' $\alpha$-renommage, sur les abstractions qui capture notre volonté d'égalité à renommage de variables près.

**Définition I.6** (relation d'$\alpha$-renommage)**.** *Soient $x, y \in V$ et $u \in \Lambda$. La relation d'$\alpha$-renommage, notée $\alpha$, est définie par*

*$\lambda x.u \ \ \alpha \ \ \lambda y.(u[x := y])$*

*si $x = y$ ou si $x$ est substituable par $y$ dans $u$ et $y$ n'est pas libre dans $u$.*

Nous allons étendre cette relation à tous les termes, c'est-à-dire sur tout l'ensemble $\Lambda$.

Nous notons $=_\alpha$ la plus petite relation comprenant $\alpha$ et tel que

- $=_\alpha$ est réflexive, symétrique et transitive
- $=_\alpha$ passe au contexte : si $u_1 =_\alpha v_1$ et $u_2 =_\alpha v_2$ alors $u_1 u_2 =_\alpha v_1 v_2$ et $\lambda x.u_1 =_\alpha \lambda x.v_1$.

**Exemple.**

1. *Il est clair que $\lambda x.x =_\alpha \lambda y.y$ par définition de la relation $\alpha$.*
2. *De même, $\lambda x.\lambda y.xy =_\alpha \lambda y.\lambda x.yx$. En effet, on montre que $\lambda x.\lambda y.xy =_\alpha \lambda z.\lambda w.zw$ et $\lambda y.\lambda x.yx =_\alpha \lambda z.\lambda w.zw$ en appliquant deux fois la subtitution (par $z$ et par $w$). Par symétrie et transitivité de $=_\alpha$, on obtient l'égalité.*

Par définition, la relation $=_\alpha$ est une relation d'équivalence. Nous construisons alors le quotient $\Lambda \backslash =_\alpha$. Dans ce quotient, les termes égaux à renommage de variable près se retrouvent dans la même classe d'équivalence. A partir de maintenant, nous considérons $\Lambda \backslash =_\alpha$, c'est-à-dire que nous parlons des termes à $\alpha$-renommage près.

## I.2 Sémantique

Maintenant que nous avons introduit la syntaxe du $\lambda$-calcul, nous allons discuter de la sémantique que nous lui associons, c'est-à-dire comment nous effectuons des calculs avec ce langage. Les calculs se définissent par des *réductions*[2] de termes, et en particulier des applications. Par exemple, nous voudrions dire que $(\lambda x.x)y$, i.e. $y$ appliqué à la fonction identité, se réduit en $y$ ou encore que $(\lambda x.(\lambda y.xy))z$, i.e. $z$ appliqué à la fonction qui retourne la fonction constante pour toute variable, se réduit en $\lambda y.zy$, i.e. la fonction constante en $z$. Nous parlons également *d'étape de calcul*, une étape de calcul correspondant à une réduction effectuée.

La définition de réduction des termes passe par une relation entre les termes appelée relation de $\beta$-réduction. Comme pour la relation $\alpha$, nous commençons par définir une relation $\beta$, et nous l'étendons au contexte.

**Définition I.7** (Relation de $\beta$-réduction)**.** *Soit $\beta$ la relation sur $\Lambda$ tel que $(\lambda x.u)v \ \ \beta \ \ u[x := v]$.*[3] *La relation de $\beta$-réduction, noté $\to_\beta$, ou simplement $\to$, est la plus petite relation contenant $\beta$ qui passe au contexte. Nous notons $\to_\beta^*$ sa fermeture réfléxive transitive et $\to_\beta^+$ sa fermeture transitive.*

Voici quelques exemples de réductions :

**Exemple.**
1. $(\lambda x.x)y \to y$.
2. $(\lambda y.(\lambda x.yx))z \to \lambda x.zx$.
3. $(\lambda w.(\lambda x.x)v)z \to (\lambda w.v)z$ *(on réduit à l'intérieur, c'est-à-dire $(\lambda x.x)v$).*
4. $(\lambda w.(\lambda x.x)v)z \to (\lambda x.x)v$ *(on réduit à l'extérieur, c'est-à-dire $(\lambda w.t)z$ où $t = (\lambda x.x)v$).*
5. $(\lambda w.(\lambda x.x)v)z \to_\beta^* v$
6. $(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$.

Un élément de la forme $(\lambda x.u)v$ est appelé *redex*. En analysant les termes que $\beta$ met en relation, la $\beta$-réduction consiste donc à réécrire les redex.

Nous définissons aussi les *valeurs* qui sont les termes finaux possibles d'une $\beta$-réduction. Dans le cas du $\lambda$-calcul, les valeurs sont les abstractions.

Un terme $t$ qui peut être réduit, c'est-à-dire qu'il existe $u$ tel que $t \to u$, est dit *réductible*. Sinon, il est dit *irréductible* ou on dit également que c'est une *forme normale*. S'il est possible de trouver une forme normale $u$ tel que $t$ se réduit en $u$, on dit que $t$ *possède une forme normale* et que $u$ est une *forme normale de $t$*.

Certains termes peuvent être réduits en des formes normales comme dans les deux premiers exemples. Dans le premier exemple, le terme irréductible n'est pas une valeur tandis que dans le second, nous obtenons une valeur.

Lorsque toute réduction commençant par $t$ possède une forme normale, on dit que le terme $t$ est *fortement normalisant*, ou tout simplement *normalisant*. Lorsqu'il existe au moins une stratégie de réduction qui permet d'obtenir un terme irréductible, on dit que le terme est *faiblement normalisant*.

Le troisième et quatrième exemples montrent qu'il existe plusieurs manières, appelées aussi *stratégie de réduction*, de réduire un terme.

2. On parle aussi de *réécriture*.
3. Ne pas oublier que nous travaillons à $\alpha$-renommage près.

Le dernier exemple montre qu'il existe des termes dont aucune réduction se termine. Celui-ci nous montre que la $\beta$-réduction ne se termine pas toujours. Ce fait n'est pas si étrange que ça : dans la plupart des langages de programmation, il est possible d'écrire des programmes qui bouclent à l'infini, c'est-à-dire que la réduction ne se termine pas.

### I.2.1 Stratégies de réduction

Nous présentons les stratégies les plus utilisées. Le terme $id(id(\lambda z.idz))$ où $id = \lambda x.x$ sera utilisé pour interpréter chaque stratégie de réduction.

#### Ordre normale

Cette méthode réduit d'abord les redex à l'extérieur, les plus à gauche. La chaine de réduction de notre exemple est alors :

$$\begin{aligned}
&\underline{id\,(id\,(\lambda z.(id\,z)))} \to_\beta \\
&\underline{id\,(\lambda z.(id\,z))} \to_\beta \\
&\lambda z.\underline{(id\,z)} \to_\beta \\
&\lambda z.z
\end{aligned}$$

#### Call by name

La stratégie appelée *call-by-name* consiste à réduire les redex les plus à gauche en premier, comme l'ordre normale. La différence est que le call-by-name ne permet pas de réduire les redex qui sont dans le corps d'un lambda.

$$\begin{aligned}
&\underline{id\,(id\,(\lambda z.(id\,z)))} \to_\beta \\
&\underline{id\,(\lambda z.(id\,z))} \to_\beta \\
&\lambda z.(id\,z)
\end{aligned}$$

La dernière étape de réduction de l'ordre normal n'est pas effectuée car celle-ci est sous le lambda $\lambda z$.

#### Call by value

La réduction dite *call-by-value* consiste à réduire en premier les redexes les plus à l'extérieur et réduire les arguments jusqu'à obtenir une valeur, et ensuite le corps de la fonction. Cette méthode de réduction est la plus courante dans les langages de programmation.

Cette stratégie appliquée à l'exemple donne :

$$\begin{aligned}
&id\underline{(id(\lambda z.(id\,z)))} \to_\beta \\
&id\underline{(\lambda z.(id\,z))} \to_\beta \\
&\lambda z.(id\,z)
\end{aligned}$$

Formellement, la stratégie call-by-value est définie par les *règles d'évaluation* définies ci-dessous, le terme $v$ étant utilisée pour une valeur.

```
let a = ref 0;;
(* Affiche 0 et 1 *)
(fun () -> Printf.printf "%d\n" (!a); a := 2)
  (Printf.printf "%d\n" (!a); a := 1);;
(* Affiche 2 *)
Printf.printf "%d\n" (!a);;
```

Code OCaml 4: Exemple qui montre que la stratégie de réduction utilisée par défaut dans OCaml est le call-by-value.

$$\frac{t_1 \to t_1'}{t_1\, t \to t_1'\, t} \qquad \text{(E-APP1)}$$

$$\frac{t \to t'}{v\, t \to v\, t'} \qquad \text{(E-APP2)}$$

$$(\lambda x.\, t)u \to t[x := u] \qquad \text{(E-ABS)}$$

La notation $\dfrac{t \to t'}{v\, t \to v\, t'}$ est l'équivalent d'une implication où la prémisse (ici $t \to t'$) se trouve au dessus et la conclusion en dessous (ici $v\, t \to v\, t'$). La règle E-APP1 se lit donc « si $t_1$ se réduit en $t_1'$, alors $t_1 t$ se réduit en $t_1' t$ ». Lorsque qu'une règle ne comporte pas de conclusion comme E-ABS, cela signifie que c'est un axiome. Cette notation sera utilisée tout au long de ce document, en particulier pour les règles de typages et de sous-typages.

Les règles (E-APP1) et (E-APP2) nous disent que nous devons, lors d'une application, réduire la fonction avant les paramètres, et ce jusqu'à obtenir une valeur. Quant à la règle (E-ABS), elle signifie qu'un redex se réduit toujours en utilisant la fonction de substitution (définition de la relation $\beta$).

Par la suite, nous considérerons toujours le call-by-value car celle-ci est la plus utilisée. La sémantique et la $\beta$-réduction seront utiles lorsque nous discuterons des théorèmes de préservation et de sureté du typage.

Plus d'informations sur la réduction sont disponibles dans [1].

Dans ce cours sont traités les sujets de normalisation (forme normale, finitude de la $\beta$-réduction), de confluence (est-ce que tout terme se réduit en une unique forme normale) et d'une différente sémantique appelée *sémantique dénotationnelle*.

## I.3 Codage de termes usuels

Le λ-calcul est assez riche pour définir des termes usuels des langages de programmations comme les booléens (et en même temps les conditions), les paires ou encore les entiers. Ces codages peuvent être trouvées dans [4]. Voici l'exemple des booléens, utilisé dans RML ([5]) :

- $true = \lambda t.\, \lambda f.\, t$
- $false = \lambda t.\, \lambda f.\, f$
- $test = \lambda b.\, \lambda t'.\, \lambda f'.\, b\, t'\, f'$

Avec les définitions de $true$ et $false$, la fonction $test$ simule le fonctionnement d'une condition : si le premier paramètre ($b$) est $true$, il renvoie $t'$, si c'est $false$, il renvoie $f'$. En effet,

$$
\begin{aligned}
&test\, true\, v\, w \to_\beta \\
&(\lambda b.\, \lambda t'.\, \lambda f'.\, b\, t'\, f')\, true\, v\, w \to_\beta \\
&(\lambda t'.\, \lambda f'.\, true\, t'\, f')\, v\, w \to_\beta \\
&(\lambda f'.\, true\, v\, f')\, w \to_\beta \\
&true\, v\, w \to_\beta \\
&v
\end{aligned}
$$

Un même raisonnement se fait pour $test\, false\, v\, w$, qui donne $w$.
Les fonctions $and$ et $or$ peuvent aussi être codées en $\lambda$-calcul.

— $and = \lambda b.\, \lambda b'.\, b\, b'\, false$
— $or = \lambda b.\, \lambda b'.\, b\, true\, b'$

# Chapitre II

# $\lambda$-calcul simplement typé.

Dans le chapitre 1, nous avons défini la syntaxe et la sémantique d'un calcul appelé le $\lambda$-calcul non typé. Nous allons maintenant ajouter une notion de types à chaque terme de notre calcul, ce qui nous mènera au $\lambda$-calcul simplement typé.

## II.1 Typage, contexte de typage et règle d'inférence

Le typage consiste à classer les termes en fonction de leur nature. Par exemple, une abstraction est interprétée comme une fonction prenant un paramètre et renvoyant un terme. Nous représentons cela par le type $\rightarrow$, appelé couramment *type flèche*. Un type flèche dépend naturellement de deux autres types : le type du terme qu'il prend en paramètre (disons $T_1$) et le type du terme qu'il retourne (disons $T_2$). Dans ce cas, l'abstraction est dite de type $T_1 \rightarrow T_2$. Un autre exemple est l'application. Une application $u\,v$ représentant une application de $v$ à la fonction $u$, il serait naturel de dire que $u$ est un type flèche dont le type de son paramètre est le type de $v$.

**Définition II.1** (Relation de typage)**.** *Soit $\Lambda$ un ensemble de termes. Soit $\tau$ un ensemble, appelé* ***ensemble des types****, dont les éléments sont notés $T$.*

*On définit une relation binaire $R$, appelée* ***relation de typage****, entre les termes et les éléments de $\tau$.*

*On dit que* ***le terme*** *$t \in \Lambda$* ***a le type*** *$T \in \tau$ si $(t, T) \in R$, noté le plus souvent $t : T$. Si un terme $t$ est en relation avec au moins un type $T$, on dit que $t$ est* ***bien typé****.*

Cette définition de la relation de typage est générale car il suffit de se donner un ensemble de termes et un ensemble de types. Dans ce chapitre, nous allons nous focaliser sur les termes du $\lambda$-calcul non typé. Dans les prochains chapitres, nous ajouterons des autres termes comme les enregistrements et nous devrons en conséquence donner un type à ces nouveaux termes.

Dans ce chapitre, nous allons travailler avec l'ensemble des types dit *simples*.

**Définition II.2.** *Soit $B$ un ensemble de type appelé de* ***types de bases****. L'ensemble des* ***types simples****, élément représentée par la lettre $T$, est défini par la grammaire suivante :*

| | | |
|---|---|---|
| $T ::=$ | | *types* |
| | $B$ | *base* |
| | $T \to T$ | *type des fonctions* |

L'ensemble $B$ pourrait être vide, et nous nous retrouvons alors avec un langage où seuls le type des fonctions existent. Cependant, il existe souvent dans les langages des types dit de bases ou primitifs, c'est-à-dire l'ensemble $B$.

## Contexte de typage

Nous avons déjà mentionné que, naturellement, les abstraction $\lambda x.\, t$ ont le type flèche, par exemple $T_1 \to T_2$. Cependant, comment pouvons nous connaître le type des arguments, c'est-à-dire le type du paramètre que la fonction attend ? Deux solutions sont couramment utilisées : soit ajouter le type dans le terme de l'abstraction soit étudier le type de corps de la fonction et en déduire le type que le paramètre devrait avoir. Dans la suite, nous utiliserons la première solution. Le terme de l'abstraction se voit alors ajouter un type à son argument et devient $\lambda x : T.\, t$. La syntaxe des termes devient alors :

| | | |
|---|---|---|
| $t ::=$ | | term |
| | $x$ | var |
| | $t\, t$ | app |
| | $\lambda x : T.\, t$ | abs |

Dans une règle de typage, il se peut que certains termes possèdent des variables libres (comme $\lambda xxy$). Lorsque nous $\beta$-réduisons un terme, il nous faut connaître chaque type de chaque variable libre (dans notre cas, $y$). Nous introduisons pour cela **le contexte de typage** qui fondamentalement est une suite finie de couple $(x_i, T_i)$ où $x_i$ est une variable et $T_i$ est un type.

**Définition II.3** (Contexte de typage)**.** *Un* ***contexte de typage****, noté* $\Gamma$*, est un ensemble fini de couple* $(x_i, T_i)$ *où* $x_i$ *est une variable et* $T_i$ *est un type.*

*L'union d'un contexte de typage* $\Gamma$ *avec le couple* $(x, T)$ *est noté* $\Gamma, x : T$ *(à la place de* $\Gamma \cup \{(x, T)\}$*).*

## Règle et jugement de typage

**Définition II.4** (Règle de typage / jugement de typage)**.** *Voir la définition récursive de wikipedia : https://fr.wikipedia.org/wiki/Lambda-calcul#cite_ref-8 Cela permet de voir un jugement de typage comme un triplet* $(\Gamma, t, T)$*, noté* $\Gamma \vdash t : T$ *qu'on lit* $t$ ***est de type*** $T$ ***dans le contexte*** $\Gamma$*. On définit récursivement un jugement de typage*

1. $(x : T) \in \Gamma \Rightarrow \Gamma \vdash t : T$
2. $\Gamma, x : T \vdash t : T_1 \Rightarrow \Gamma, x : T \vdash \lambda(x : T)t : T \to T_1$
3. $\Gamma \vdash u : T_1 \to T_2 \wedge \Gamma \vdash u : T_1 \Rightarrow \Gamma \vdash uv : T_2$

*Si* $(t, T) \in \Gamma$*, on dit alors que* $t$ *est bien typé dans* $\Gamma$*.*

Le contexte est utilisé pour faire des hypothèses sur chaque variable libre dans le $\lambda$-terme $t$. Donc, on ne peut avoir un jugement de typage de type :

$\vdash (\lambda(x : T_1)x)y$ car toutes les variables libres (en l'occurrence ici $y$) du $\lambda$-terme ne sont pas typées dans le contexte  : nous ne connaissons pas le type de $y$.

Comme pour les règles d'évaluation, on écrit la définition d'un jugement de typage comme des règles d'inférence. La définition d'un jugement de typage devient donc :

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$

$$\frac{\Gamma, x : T \vdash t : T_1}{\Gamma, x : T \vdash \lambda(x : T)t : T_1} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash u : T_1 \to T_2 \qquad \Gamma \vdash v : T_2}{\Gamma \vdash uv : T_2} \text{ (T-APP)}$$

Le $\lambda$-calcul simplement typé est donc un tuple $(\Lambda, \to, \tau, \Gamma_\tau)$ où

1. $\Lambda$ est l'ensemble des $\lambda$-termes.
2. $\to$ est la relation de $\beta$-réduction.
3. $\tau$ l'ensemble des types.
4. $\Gamma$ est le jugement de typage (défini récursivement).

## II.2 Syntaxe

## II.3 Sémantique et règle de typage

## II.4 Sûreté

Expliquer la préservation et la progression. Regarder dans le cours de l'ENS à la place ? ?

Avant de montrer la préservation et la progression, il est nécessaire de remarquer certains faits qui découlent immédiatement des règles de typages.

### Progression

**Lemme II.5** (Inversion des règles de typage)**.** *1. Si* $\Gamma \vdash x : T$*, alors* $(x : T) \in \Gamma$

*2.*

*3.*

*4.*

*5.*

*6.*

*Démonstration.* Evident d'après les règles de typages données. □

Une autre remarque importante sur le calcul $\lambda_{\rightarrow}$ est l'unicité de type pour les $\lambda$. Cette proposition est tellement fondamentale que le terme théorème est utilisé. Cependant, cette propriété n'est pas vraie dans tous les calculs, comme nous le montrerons quand nous introduirons le sous-typage.

**Théorème II.6.** *Tout λ-terme bien typé possède un type unique.*

*Démonstration.* □

**Théorème II.7** (de progression de $\lambda_{\rightarrow}$)**.** *Soit $t$ un terme bien typé sans variable libre. Alors, soit $t$ est une valeur, soit il existe $t'$ tel que $t \rightarrow t'$.*

*Démonstration.* □

### Préservation

**Lemme II.8** (d'affaiblissement)**.** *Soit $\Gamma \vdash t : T$ et $x \notin dom(\Gamma)$.*
*Alors $\Gamma, x : S \vdash t : T$.*

*Démonstration.* □

**Lemme II.9** (de préservation du typage pour la substitution)**.** *Soit $\Gamma, x : S \vdash t : T$ et $\Gamma \vdash s : S$.*
*Alors $\Gamma \vdash [x \rightarrow s]t : T$*

*Démonstration.* □

## II.5 Enrichir le calcul avec des types de bases

Remarquer qu'on peut ajouter d'autres types comme les listes, les records, etc avec des règles de typages et des règles d'évaluations propres sans que cela ne change la propriété de soundness.

Nous utiliserons dans la suite la syntaxe let x = t in u qui est un alias pour une forme de lambda (la donner). Cette syntaxe nous permet de n'étudier que les applications entre variables. En effet, si nous avons une expression de la forme (t u), nous pouvons réduire l'expression sous la forme (x y) avec let x = t in let y = u in x y

Cependant, il faut vérifier que la sémantique reste la même. Je ne pense pas...

# Chapitre III

# $\lambda$-calcul avec sous-typage et enregistrements.

Remarquons que de l'information sur le type du retour est perdue dans certains cas. Par exemple, prenons l'expression

let f = lambda(x : Any) x in let g = lambda(y : Nothing) y in f g

## III.1 Syntaxe

## III.2 Sémantique et règle de typage

## III.3 Sureté

# Chapitre IV

# System F

## IV.1 Syntaxe

## IV.2 Sémantique et règle de typage

## IV.3 Sureté

# Chapitre V

# System $F_{<:}$

## V.1 Syntaxe

## V.2 Sémantique et règle de typage

## V.3 Sureté

Qu'en est-il de la sureté de ce calcul ? Les théorèmes de progression et de préservation restent-ils vrais ?

La réponse est *oui*. Cependant, la preuve est technique et assez longue.

TODO : donner quelques raisons pourquoi la preuve est technique.

Une preuve complète et détaillée peut être trouvée dans [3].

# Chapitre VI

# $\lambda$-calcul simplement typé avec type récursif

## VI.1 Syntaxe

## VI.2 Sémantique et règle de typage

## VI.3 Sureté

Qu'en est-il de la sureté de ce calcul ? Les théorèmes de progression et de préservation restent-ils vrais ?

La réponse est *oui*. Cependant, la preuve est technique et assez longue.

TODO : donner quelques raisons pourquoi la preuve est technique.

Une preuve complète et détaillée peut être trouvée dans [2].

# Chapitre VII

# Enregistrement avec type chemin dépendant

Faire le lien avec le sujet initial.

## VII.1 Syntaxe

## VII.2 Sémantique et règle de typage

Pouvoir définir des record récursifs.

## VII.3 Encodage de System $F_{<:}$

Montrer l'inclusion comme dans WF, comment les variables de types sont gérées.

## VII.4 Sureté

Ne pas tout démontrer, voir les théorèmes dans WF.

# Chapitre VIII

# RML : implémentation

- Lien vers GitHub.

- Préciser que tout est écrit en OCaml, avec ocamllex et menhir comme lexeur et parseur. - Donner les mêmes exemples qu'au début, mais cette fois-ci avec RML. - Montrer ce qui est possible en RML et ce qui ne l'est pas en OCaml.

## VIII.1 Gestion des variables

La gestion de variables liées et des variables libres est la tâche la plus fastidieuse lors de l'implémentation d'un compilateur/interpréteur. Pour ne pas perdre de temps, utilisation d'alphaLib.

- Utilisation de AlphaLib pour les variables libres et liées ainsi que l'exploration de l'AST. Décrire les avantages d'AlphaLib (rename variable, transformation raw -> nominal, fresh name facile, égalité de terme à $\alpha$-renommage près, etc).

- Discuter de la représentation des variables en termes d'atomes. Parler de la différence entre un type raw et un type nominal. Le parseur crée un type raw et lorsque nous obtenons un type raw du parseur, nous le transformons en type nominal.

## VIII.2 Autres

- Avoidance problem = le problème d'échappement. Donner les exemples qui sont dans `dsubml/test/typing/simple_wrong.dsubml`.

- Structure de données pour le contexte : map d'atomes vers un type nominal. - Gestion d'un environnement non vide, avec une librairie standard. On regarde alors maintenant les termes top level qui sont composés soit d'un let top level (sans in), soit d'un terme. Les termes top level sont là pour étendre l'environnement tandis que les termes usuels non. Dans l'implémentation, cela se traduit par un type somme Grammar.TopLevelTerm. Lorsque nous rencontrons un terme top level qui est un let, nous appelons la fonction `$read_top_level_term` qui se charge de...

- Possibilité de voir l'arbre de dérivation de typage.

- Utilisation du terme Unimplemented.

## VIII.3 Complexité des algorithmes

- Donner un détail sur la complexité des algorithmes de sous-typages et de typages. Expliquer pourquoi on a supprimé la règle REFL pour la remplacer par REFL-TYP et donner la preuve d'équivalence (qui est directe, en quelques mots).

## VIII.4 Algorithme de typage

- L'ordre dans le pattern matching n'a pas énormément d'importance. - Algorithme d'inférence de type pour les modules. Expliquer les points positifs et les points négatifs. Discuter des améliorations possibles. - Montrer un exemple d'arbre de dérivation.

## VIII.5 Algorithme de sous-typage

- L'ordre dans le pattern matching a tout son importance. En effet, pour arriver à une conclusion, il est possible d'y arriver par plusieurs chemins.
- Réunion de REFL en une seule.
- Montrer un exemple d'arbre de dérivation.

## VIII.6 Système d'« actions »

- Expliquer le système d'actions pour DSubML.
- Problème algorithmique pour les types chemins dépendants. Quel est vraiment le type de `x.A` ? Parler des types bien formés.
- Implémentation de DSubml et de RML. Insister sur le fait que l'extension n'est pas si simple pour l'implémentation.
- Algorithme d'inférence (partiel) de types pour les modules.

## VIII.7 Sucres syntaxiques

- fun(x : int.T) -> fun(y : int.T) <==> fun(x : int.T, y : int.T) - sucres syntaxiques pour unit. - Ascription et check de sous-typage. - Choix type unimplemented. - Choix pour le typage d'un entier dans l'algo (bottom, pour ascription). - Séparation dans le parser des termes qui doivent avoir des parenthèses quand on les utilise comme paramètres de fonctions et ceux qui n'en ont pas besoin. - application de fonctions à plusieurs paramètres. - Remarquer qu'il y a une erreur pour les bindings locaux. On ne peut binder localement une variable sans être sûr que l'on ait pas l'avoidance problem.

## VIII.8 Types de bases

- Implémentation des types de bases. - Entiers dans le lexeur. - Enregistrement : RecursiveType mais avec un ID `'self` pour éviter de p

## VIII.9 Ce qui n'est pas fait

- Quand une question a déjà été posée, stack overflow. - Evaluation. - top level.

# Conclusion

# Bibliographie


[1] Goubault-Larrecq, J. Cours intitulé $\lambda$-calculs et langages fonctionnels. http://www.lsv.fr/~goubault/Lambda/lambda.pdf.

[2] Pierce, B. C. *TAPL - Chapter 21 - Metatheory of recursive types.* The MIT Press, 2002.

[3] Pierce, B. C. *TAPL - Chapter 28 - Metatheory of bounded quantification.* The MIT Press, 2002.

[4] Pierce, B. C. *TAPL - Chapter 5 - The Untyped Lambda-Calculus.* The MIT Press, 2002.

[5] Willems, D. RML - ML modules and functors as first-class citizens. https://github.com/dannywillems/RML.