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

Table des matières

I Introduction **3**

II λ-calcul non typé **5**
II.1 Syntaxe 5
II.2 Sémantique 6
II.2.1 Différentes stratégies de réduction 6

III λ-calcul simplement typé. **9**
III.1 Sûreté 10
III.2 Enrichir le calcul avec des types de bases 12

IV λ-calcul simplement typé avec type récursif **13**
IV.1 Sureté 13

V λ-calcul avec sous-typage et enregistrements. **15**
V.1 Sureté 15

VI System F **17**
VI.1 Sureté de System F 17

VII System $F_{<:}$ **19**
VII.1 Sureté de System $F_{<:}$ 19

VIII Enregistrement avec type chemin dépendant **21**
VIII.1 Sureté 21
VIII.2 Implémentation 21

Chapitre I

Introduction

La programmation modulaire est un principe de développement visant 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 chaines 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 valeurs prenant un module en paramètres et en retournant un autre. De plus, il n'est pas possible de définir des valeurs prenant un module et un type de base et retournant un module (ou un type de base).

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 ressemble en quelques points aux modules.

Ce mémoire vise à fournir un calcul typé, construit à partir du λ-calcul simplement typé, dans lequel le langage de modules est confondu avec le langage de base grâce aux enregistrements. Une preuve de la sûreté de ce calcul ainsi qu'un interpréteur avec un algorithme de sous-typage et d'inférence de type est fourni.

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

Dans le chapitre 1, nous présentons *le λ-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 λ-calcul, ce qui nous donnera *le λ-calcul simplement typé*. Nous discuterons de la notion de *sureté du typage* à travers *les théorèmes de préservation et de progression* que nous démontrons pour ce calcul typé.

Dans le chapitre 3, nous parlerons de la notion de type récursif et nous étudierons *le λ-calcul simplement typé avec type récursif*.

Dans les chapitres 4, 5 et 6, nous enrichissons le λ-calcul simplement typé avec la notion de polymorphisme qui permet d'attribuer plusieurs types à un terme. Le chapitre 4 se concentre sur *le polymorphisme avec sous-typage*, illustré avec les enregistrements. Dans le chapitre 5, nous parlons de *polymorphisme paramétré* qui, combiné au λ-calcul simplement typé, forme le calcul appelé *System F*. Le chapitre 6 se charge de combiner ces deux notions de polymorphismes dans un calcul appelé *System $F_{<:}$*.

Pour finir, dans le chapitre 7, nous complétons les enregistrements définis dans le chapitre 4 avec les *types chemins dépendants* qui offre la possibilité d'ajouter des types dans les enregistrements. Nous montrerons comment ce dernier calcul permet de résoudre le problème posé.

Chaque calcul possède une implémentation écrite en OCaml dont les choix et les difficultés d'implémentation ainsi qu'un lien vers celle-ci sont donnés dans une section dans le chapitre correspondant au calcul.

Chapitre II

λ-calcul non typé

II.1 Syntaxe

Définition II.1 (Syntaxe du λ-calcul). *Soit V un ensemble infini dénombrable. On note Λ, appelés **l'ensemble des λ-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 Λ est appelé un **λ-terme**. Un λ-terme de la forme uv est appelé **application** car l'interprétation donnée est une fonction u évaluée en v. Un λ-terme de la forme $\lambda\ x.u$ est appelé **abstraction**. Il est souvent representé comme la fonction qui envoie x sur u.

Des exemples de λ-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 : $\lambda y.(\lambda x.y)$.
- l'application identité appliquée à la fonction identité : $(\lambda x.x)(\lambda y.y)$

Comme dans une expression mathématique, il est important de différentier les variables libres et les variables liées d'un λ-terme. Par exemple, dans le λ-terme $\lambda x.x$ la variable x est liée par un λ tandis que dans l'expression $\lambda x.y$ la variable y est libre.

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

- $FV(x) = \{x\}$

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

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

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

Cela nous amène à la définition suivante :

Définition II.4 (Relation d'α-renommage)**.**

Les éléments $\lambda x \lambda y x y$ et $\lambda y \lambda x y x$ appartiennent à la même classe d'équivalence, ce que nous souhaitons.

On se concentre maintenant uniquement sur les classes d'équivalence.

II.2 Sémantique

A toute syntaxe, nous associons une *sémantique*, c'est-à-dire une interprétation des termes.

Pour le λ-calcul non typé, la sémantique que nous allons définir permet de réduire un λ-terme vers un autre λ-terme. Nous parlons de *β-réduction*, ou encore de *réécriture*. La β-réduction peut se voir comme une relation binaire sur Λ.

Sémantique opérationnelle.

II.2.1 Différentes stratégies de réduction

Parler des différentes stratégies d'évaluations.

Call by value

Call by name

Parler des méthodes de preuves sur le λ-calcul : preuves termes par termes, par la taille du terme.

Normalisation

Peut être ne pas donner des preuves, mais en parler pour dire que c'est très important.

On se pose des questions sur la finitude des évaluations.

Regarder du coté du cours de l'ENS Lyon, chap 3 pour le lambda-calcul simplement typé.

Définition II.5. *On dit qu'un λ-terme est*

— ***fortement normalisable*** *si toute chaine de β-réduction est finie.*

— ***faiblement normalisable*** *s'il existe une chaine de β-réduction finie.*

Chapitre III

λ-calcul simplement typé.

Dans le chapitre 1, nous avons défini la syntaxe et la sémantique d'un calcul appelé le λ-calcul non typé. Nous allons maintenant ajouter une notion de types à chaque terme de notre calcul.

Motivation du typage.

Définition III.1 (Relation de typage)**.** *Soit τ un ensemble, appelé* ***ensemble des types****, dont les éléments sont notés T_i. Soit Λ l'ensemble des λ termes.*

On définit une relation binaire R entre les λ termes et les éléments de τ.

On dit que ***le terme*** *$t \in \Lambda$* ***a le type*** *$T \in \tau$ si $(t, T) \in R$ (noté plus souvent $t : T$).*

Le typage est donc un moyen de spécifier l'appartenance de certains λ-termes à un ensemble précis de type et ainsi réduire les opérations possibles sur ces λ-termes.

Dans le chapitre 1 sur le λ-calcul non typé, nous avons défini une relation d'évaluation, noté $\to$. Nous pouvons nous demander comment la relation de typage est compatible avec la relation $\to$.

Dans une règle de typage, il se peut que certains termes possèdent des variables libres (comme λxxy). Lorsque nous β-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 III.2 (Contexte de typage)**.** *Un* ***contexte de typage****, noté Γ, 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 Γ avec le couple (x, T) est noté $\Gamma, x : T$ (à la place de $\Gamma \cup \{(x, T)\}$).

Définition III.3 (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 (Γ, t, T), noté $\Gamma \vdash t : T$ qu'on lit t ***est de type*** T ***dans le contexte*** Γ*. 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. $\Gamma \vdash u : T_1 \to T_2 \land \Gamma \vdash u : T_1 \Rightarrow \Gamma \vdash uv : T_2$

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

Le contexte est utilisé pour faire des hypothèses sur chaque variable libre dans le λ-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 λ-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 :

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

(T-VAR)

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

(T-ABS)

$$\frac{\Gamma \vdash u : T_1 \to T_2 \qquad \Gamma \vdash v : T_2}{\Gamma \vdash uv : T_2}$$

(T-APP)

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

1. Λ est l'ensemble des λ-termes.
2. $\to$ est la relation de β-réduction.
3. τ l'ensemble des types.
4. Γ est le jugement de typage (défini récursivement).

III.1 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 III.4 (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 λ. 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 III.5. *Tout* λ*-terme bien typé possède un type unique.*

Démonstration. □

Théorème III.6 (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 III.7 (d'affaiblissement)**.** *Soit* $\Gamma \vdash t : T$ *et* $x \notin dom(\Gamma)$*.*

Alors $\Gamma, x : S \vdash t : T$*.*

Démonstration. □

Lemme III.8 (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. □

Normalisation

Peut être ne pas donner des preuves, mais en parler pour dire que c'est très important.

On se pose des questions sur la finitude des évaluations.

Regarder du coté du cours de l'ENS Lyon, chap 3 pour le lambda-calcul simplement typé.

Définition III.9. *On dit qu'un λ-terme est*

- ***fortement normalisable*** *si toute chaine de β-réduction est finie.*
- ***faiblement normalisable*** *s'il existe une chaine de β-réduction finie.*

III.2 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 IV

λ-calcul simplement typé avec type récursif

IV.1 Sureté

Chapitre V

λ-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

V.1 Sureté

Chapitre VI

System F

VI.1 Sureté de System F

Chapitre VII

System $F_{<:}$

VII.1 Sureté de System $F_{<:}$

Chapitre VIII

Enregistrement avec type chemin dépendant

Faire le lien avec le sujet initial, c'est-à-dire que cela résoud le problème

VIII.1 Sureté

VIII.2 Implémentation

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

- Utilisation de AlphaLib pour les variables libres et liées ainsi que l'exploration de l'AST. Décrire les avantages d'AlphaLib.

- 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).

- 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.

- Préciser que tout est écrit en OCaml, avec ocamllex et menhir comme lexeur et parseur.

- Expliquer le système d'actions pour DSubML.
- Parler des sucres syntaxiques.

Conclusion