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

I λ-calcul non typé **9**
I.1 Syntaxe . . . 9
I.2 Sémantique . . . 12
I.2.1 Stratégies de réduction . . . 13
I.3 Codage de termes usuels . . . 14

II λ-calcul simplement typé. **17**
II.1 Typage, contexte de typage et règle d'inférence . . . 17
II.2 Sûreté du typage . . . 19

III λ-calcul avec sous-typage et enregistrements. **23**
III.1 λ-calcul simplement typé avec enregistrements . . . 23
III.2 Sous-typage . . . 25
III.3 Sureté . . . 25
III.4 Top et Bottom . . . 25

IV System F **27**
IV.1 Syntaxe . . . 27
IV.2 Sémantique et règle de typage . . . 27
IV.3 Sureté . . . 27

V System $F_{<:}$ **29**
V.1 Syntaxe . . . 29
V.2 Sémantique et règle de typage . . . 29
V.3 Sureté . . . 29

VI λ-calcul simplement typé avec type récursif **31**
VI.1 Syntaxe . . . 31
VI.2 Sémantique et règle de typage . . . 31
VI.3 Sureté . . . 31

VII Enregistrement avec type chemin dépendant **33**
VII.1 Syntaxe . . . 33
VII.2 Sémantique et règle de typage . . . 33
VII.3 Encodage de System $F_{<:}$ . . . 33
VII.4 Sureté . . . 33

VIII RML : implémentation **35**
VIII.1 Gestion des variables 35
VIII.2 Autres 35
VIII.3 Complexité des algorithmes 36
VIII.4 Algorithme de typage 36
VIII.5 Algorithme de sous-typage 36
VIII.6 Système d'« actions » 36
VIII.7 Sucres syntaxiques 36
VIII.8 Types de bases 36
VIII.9 Ce qui n'est pas fait 37

A Preuve par récurrence sur les termes et les types **41**

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 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 démontrerons pour ce calcul typé.

Dans les chapitres 3, 4 et 5, nous enrichissons le λ-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 λ-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

λ-calcul non typé

Dans ce chapitre, nous allons introduire les bases théoriques de la programmation fonctionnelle en parlant du λ-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 β-réduction.

I.1 Syntaxe

Définition I.1 (Syntaxe du λ-calcul)**.** *Soit V un ensemble infini dénombrable dont les éléments sont appelés **variables**. On note Λ, appelé **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**, ou tout simplement un **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**, 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

$t ::=$		terme
	x	var
	$t\, t$	app
	$\lambda x.\, t$	abs

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 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 : $\lambda y.\, \lambda xy.$.
— 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 λ-terme. Par exemple, dans le λ-terme $\lambda x.\, x$ la variable x est liée par un λ [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 Λ 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 Λ par :*

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

Un terme qui ne comporte pas de variable libre est dit *clos*.

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 Λ. 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

1. on dit aussi qu'elle est « sous »un λ.

n'a pas le même sens que $\lambda x.y$. Cet exemple nous montre que nous devons faire 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' α-renommage, sur les abstractions qui capture notre volonté d'égalité à renommage de variables près.

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

Nous notons $=_\alpha$ la plus petite relation comprenant α 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 α.*
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 α-renommage près.

I.2 Sémantique

Maintenant que nous avons introduit la syntaxe du λ-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 β-réduction. Comme pour la relation α, nous commençons par définir une relation β, et nous l'étendons au contexte.

Définition I.7 (Relation de β-réduction)**.** *Soit β la relation sur Λ tel que $(\lambda x.u)v \ \ \beta \ \ u[x := v]$.*[3] *La relation de β-réduction, noté $\to_\beta$, ou simplement $\to$, est la plus petite relation contenant β 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 β met en relation, la β-réduction consiste donc à réécrire les redex.

Nous définissons aussi les *valeurs* qui sont les termes finaux possibles d'une β-réduction. Dans le cas du λ-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 α-renommage près.

Le dernier exemple montre qu'il existe des termes dont aucune réduction se termine. Celui-ci nous montre que la β-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 λz.

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 :

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

```
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.

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

$$\frac{t_1 \to t_1'}{t_1\, t \to t_1'\, t} \quad \text{(E-APP1)} \qquad \frac{t \to t'}{v\, t \to v\, t'} \quad \text{(E-APP2)}$$

$$(\lambda x.\, t)v \to [x := v]t \quad \text{(E-APPABS)}$$

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-APPABS, 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-APPABS), elle signifie qu'un redex se réduit toujours en utilisant la fonction de substitution (définition de la relation β).

La relation de β-réduction pour la stratégie call-by-value est alors définie comme le plus petit ensemble généré par les règles I.2.1. Quand nous ajouterons à notre langage des autres termes comme les enregistrements, nous mentionnerons uniquement les règles d'évaluation, la relation de β-réduction étant implicitement définie de la même manière.

Par la suite, nous considérerons toujours cette dernière stratégie car c'est la plus utilisée.

La réécriture des termes est un large sujet, plus d'informations sur ce sujet sont disponibles dans [1]. Dans ce cours sont traités les sujets de normalisation (forme normale, finitude de la β-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 λ-calcul.

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

Chapitre II

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, ce qui nous mènera au λ-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 $T_1 \rightarrow T_2$, lu « T_1 flèche 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 Λ un ensemble de termes. Soit τ 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 τ.*

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 λ-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'en-*

semble des ***types simples*** *est défini par la grammaire suivante :*

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

L'ensemble de base B est assez naturel : il existe souvent dans les langages des types dit de bases ou primitifs.

Contexte et jugement 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 :

$t ::=$		terme
	x	var
	$t\, t$	app
	$\lambda x : T.\, t$	abs

Avant de discuter des règles de typages, il convient de remarquer qu'il est nécéssaire de connaitre certaines informations quand nous souhaitons typer des termes. En effet, si nous prenons le terme $\lambda x : T.\, y$ et que nous souhaitons le typer, il est nécessaire de connaître le type de y. Cela nous amène à la notion de *contexte de typage.*

Définition II.3 (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. Chaque* x_1 *est différent.*

L'union d'un contexte de typage Γ *avec un couple* (x, T) *est noté* $\Gamma, x : T$*. Le contexte vide est noté* $\emptyset$*.*

La relation de typage devient alors une relation à trois composantes : le contexte, le terme et le type. Nous parlons alors de *jugement de typage.*

Définition II.4 (Jugement de typage)**.** ***Un jugement de typage*** *est un triplet* (Γ, t, T) *où* Γ *est un contexte de typage,* t *un terme et* T *un type. Nous le notons le plus souvent* $\Gamma \vdash t : T$ *et nous disons « t à le type T sous les hypothèses* Γ [1]*. Si* Γ *est vide, nous omettons* $\emptyset$ *et le jugement devient* $\vdash t : T$*.*

Règle de typage et arbre de dérivation

Maintenant, nous avons les outils pour définir nos règles de typages, c'est-à-dire comment nous assignons les types aux termes.

1. ou encore dans le contexte Γ

Définition II.5 (Règles de typage)**.** *Les règles de typage pour le λ-calcul simplement typé sont*

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma, x : T_1 \vdash \lambda(x : T_1)t : T_1 \to T_2} \text{ (T-ABS)}$$

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

La règle $(T - VAR)$ est évidente : si (x, T) est dans le contexte, alors x est de type T sous le contexte Γ. Quant à $(T - ABS)$, elle affirme que si le terme t de l'abstraction $\lambda x : T_1 . t$ est de type T_2, alors l'abstraction est de type $T_1 \to T_2$. Pour finir, $(T - APP)$ type les applications : dans le terme $u\, v$, u doit être une fonction de type $T_1 \to T_2$, et v doit être du même type que u attend, c'est-à-dire T_1, l'application ayant le type T_2.

Le typage d'un terme produit des *arbres de dérivation de typage* (ou tout simplement *une dérivation de typage*). Un arbre de dérivation de typage est un arbre dont les noeuds sont des jugements de typages, construits à partir des règles de typages et dont la racine est le jugement de typage du terme à typer. La racine de l'arbre est également appelée *conclusion*. La racine de l'arbre de dérivation est le jugement de typage le plus en bas. Les branches sont annotées par le nom des règles qui permet de déduire le type.

Par exemple, $\lambda x : T_1 \to T_2 . \lambda y : T_1 . x\, y$ est de type $(T_1 \to T_2) \to T_1 \to T_2$. Un arbre de dérivation possible est

$$\dfrac{\dfrac{\dfrac{\dfrac{x : T_1 \to T_2 \in \Gamma}{\Gamma \vdash x : T_1 \to T_2}\text{ (T-VAR)} \qquad \dfrac{y : T_1 \in \Gamma}{\Gamma \vdash y : T_1}\text{ (T-VAR)}}{\Gamma, x : T_1 \to T_2, y : T_1 \vdash x\, y : T_2}\text{ (T-APP)}}{\Gamma, x : T_1 \to T_2 \vdash \lambda y : T_1 . x\, y : T_1 \to T_2}\text{ (T-ABS)}}{\Gamma \vdash \lambda x : T_1 \to T_2 . \lambda y : T_1 . x\, y : (T_1 \to T_2) \to T_1 \to T_2}\text{ (T-ABS)}$$

II.2 Sûreté du typage

Dans cette partie, nous allons aborder deux théorèmes importantes : les théorèmes de progression et de préservation du typage. En assemblant ces deux théorèmes, nous en déduisons le principe « les programmes [2] bien typés ne bloquent pas ». Ne pas bloquer signifie que si le programme se termine (un programme bien typé peut contenir une boucle infinie), alors il s'évaluera en une valeur du type du programme.

Ces deux théorèmes unifient les deux relations précédemment définies : la relation de typage et la relation de β-réduction.

1. Progression : si un terme est bien typé, alors soit il s'évalue en une valeur, soit il s'évalue en un terme.
2. Préservation (du typage) : si un terme t de type T s'évalue en un terme t', alors t' est de type T.

2. Un programme est synonyme de terme.

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

Lemme II.6 (Inversion des règles de typage)**.**
1. *Si* $\Gamma \vdash x : T$*, alors* $(x : T) \in \Gamma$
2. *Si* $\Gamma \vdash \lambda x : T_1 . t_2 : T$ *alors* $T = T_1 \to T_2$ *pour un* T_2 *tel que* $t : T_2$*.*
3. *Si* $\Gamma \vdash t_1 t_2 : T$*, alors il existe* T_1 *tel que* $t_1 : T_1 \to T$ *et* $t_2 : T_1$*.*

Démonstration. Ces propositions découlent des règles de typage. En effet, pour la deuxième par exemple, la seule règle qui permet d'affirmer que $\Gamma \vdash \lambda x : T_1 . t_2 : T$ est $T - ABS$. □

Le lemme d'inversion des règles de typage dit également quelque chose de fondamentale sur les arbres de typage et qui a une énorme importance lorsque nous souhaitons implémenter un algorithme de typage [3]. En effet, les 3 points du lemmes nous donnent quels sont les possibles fils de la conclusion. Par exemple, si nous devons montrer que $\lambda x : T . t : T'$, nous sommes convaincus, par le lemme d'inversion, que le noeud précédent provient de la règle $(T - ABS)$. Cela implique que pour un jugement de typage donné, il n'y a au plus qu'un seul arbre de dérivation.

Une autre remarque importante, et qui découle du fait que les arbres de dérivations ont uniques, est l'unicité de type. Nous verrons que cette proposition n'est pas vraie dans tous les calculs.

Théorème II.7 (Unicité du typage)**.** *Soit* t *un* λ*-terme. Si* t *est bien typé, alors son type est unique. De plus, il existe au plus un arbre de dérivation qui permet de montrer que* t *a ce type.*

Démonstration. Supposons que t possède deux types, par exemple S et T. Nous avons donc les jugements de typage : $\Gamma \vdash t : S$ et $\Gamma \vdash t : T$. Nous procédons par induction sur la structure des termes.

- t est une variable x. Alors nous avons les jugements de typage $\Gamma \vdash x : S$ et $\Gamma \vdash x : T$. Par le lemme d'inversion, nous en déduisons que $(x, S) \in \Gamma$ et $(x, T) \in \Gamma$. Comme une variable ne peut apparaître qu'une fois dans un contexte, nous en déduisons $S = T$.
- t est de la forme $\lambda x : T_1 . t'$. Par le lemme d'inversion, $S = T_1 \to R_1$ et $T = T_1 \to R_2$ avec $t' : R_1$ et $t' : R_2$. Par induction sur t', nous déduisons que $R_1 = R_2$. Donc $S = T$.
- t est de la forme $u\, v$. Par le lemme d'inversion, il existe T_1 tel que u est de type $T_1 \to S$ et de type $T_1 \to T$ avec v de type T_1. Par induction sur u, le type de u est unique donc $S = T$

L'unicité de l'arbre de dérivation découle immédiatement du lemme d'inversion et de la remarque ci-dessus. □

Théorème II.8 (de progression de $\lambda_\to$)**.** *Soit* t *un terme bien typé sans variables libre. Alors, soit* t *est une valeur, soit il existe* t' *tel que* $t \to t'$*.*

Démonstration. Nous procédons par induction sur la structure des termes.

- Le cas d'une variable n'est pas possible. En effet, aucune règle d'évaluation n'a comme prémisse une variable.

3. Nous verrons par la suite que ce n'est pas tout le temps évident de passer des règles de typage à un algorithme de typage.

- t est une abstraction. Le résultat est direct car t est une valeur.
- t est de la forme $u\,v$. t étant bien typé, nous avons le jugement de typage $\vdash t : T$ où Γ est vide car t ne possède pas de variables libres. Par le lemme d'inversion, $u : T_1 \to T$ et $v : T_1$. Par induction, comme u (resp. v) est bien typé, u (resp. v) est soit une valeur, soit s'évalue en un u' (resp. v').
  - Si u s'évalue en u', alors $(E - APP1)$ s'applique et $u\,v$ s'évalue en $u'\,v$.
  - Si u est une valeur et v s'évalue en v', alors $(E - APP2)$ s'applique et $u\,v$ s'évalue en $u\,v'$.
  - Si u et v sont des valeurs, $(E - APPABS)$ s'applique.

□

Préservation

Lemme II.9 (de permutation)**.** *Soit* $\Gamma \vdash t : T$ *et soit* Δ *une permutation de* Γ*. Alors* $\Delta \vdash t : T$*.*

Démonstration. Par induction sur la structure des termes.

- t est une variable x. Par hypothèse, $\Gamma \vdash x : T$. Par le lemme d'inversion, $(x, T) \in \Gamma$ d'où $(x, T) \in \Delta$. Par $(T - VAR)$, $\Delta \vdash x : T$.
- $t = \lambda x : T_1.\, t'$. Par hypothèse et le lemme d'inversion, $\Gamma \vdash \lambda x : T_1.\, t' : T_1 \to T_2$ et $\Gamma, x : T_1 \vdash t' : T_2$. Par hypothèse de récurrence, $\Delta, x : T_1 \vdash t' : T_2$. Par $(T - ABS)$, $\Delta \vdash \lambda x : T_1.\, t' : T_1 \to T_2$.
- $t = u\,v$. Nous savons que $\Gamma \vdash u\,v : T$. Par le lemme d'inversion, nous obtenons $\Gamma \vdash u : T_1 \to T$ et $\Gamma \vdash v : T_1$. Par hypothèse de récurrence, $\Delta \vdash u : T_1 \to T$ et $\Delta \vdash v : T_1$. Par $(T - APP)$, $\Delta \vdash u\,v : T$.

□

Lemme II.10 (d'affaiblissement)**.** *Soit* $\Gamma \vdash t : T$ *et* $x \notin dom(\Gamma)$*.*
Alors $\Gamma, x : S \vdash t : T$*.*

Démonstration. Par induction sur la structure des termes.

- t est une variable y. Le cas où $y = x$ est impossible car par le lemme d'inversion, nous avons $(x, T) \in \Gamma$ et cela contredit l'hypothèse que $x \notin \Gamma$. Si $y \neq x$, $(y, T) \in \Gamma$ par le lemme d'inversion et par conséquence, $(y, T) \in \Gamma, x : S$ et nous concluons en utilisant $(T - VAR)$.
- $t = \lambda y : T_1.\, t'$ tel que $\Gamma \vdash t : T$. Par le lemme d'inversion, nous avons $T = T_1 \to T_2$ et $\Gamma, y : T_1 \vdash t' : T_2$. Par hypothèse de récurrence, on a $\Gamma, y : T_1, x : S \vdash t' : T_2$. Par le lemme de permutation, nous avons $\Gamma, x : S, y : T_1 \vdash t' : T_2$ et par (T-ABS), nous déduisons $\Gamma, x : S \vdash \lambda y : T_1.\, t' : T_1 \to T_2$.
- $t = u\,v$ tel que $\Gamma \vdash u\,v : T$. Par le lemme d'inversion, nous avons $\Gamma \vdash u : T_1 \to T$ et $\Gamma \vdash v : T_1$. Par hypothèse de récurrence, $\Gamma, x : S \vdash u : T_1 \to T$ et $\Gamma, x : S \vdash v : T_1$. Nous concluons que $\Gamma, x : S \vdash u\,v : T$ par T-APP.

□

Lemme II.11 (de préservation du typage pour la substitution)**.** *Soit* $\Gamma, x : S \vdash t : T$ *et* $\Gamma \vdash s : S$*.*
Alors $\Gamma \vdash [x \to s]t : T$

Démonstration. Nous procédons par une induction sur l'arbre de dérivation $\Gamma, x : S \vdash t : T$.

— $t = z$. Alors, par le lemme d'inversion, $(z, T) \in \Gamma, x : S$. Deux cas sont possibles. Si $z = x$, alors $[x \to s]z = s$ ainsi que $S = T$ et nous obtenons le résultat souhaité. Si $z \neq x$, alors $[x \to s]z = z$ et il n'y a rien à montrer car $\Gamma \vdash z : T$.
— $t = \lambda y : T_1 . t'$. Sans perte de généralité, nous supposons $y \notin FV(s)$ et $x \neq y$. Rappelons que par définition de la β-réduction,

$$[x \to s](\lambda y : T_1 . t') = (\lambda y : T_1 .)([x \to s]t')$$

Alors $T = T_1 \to R$ avec $\Gamma, x : S, y : T_1 \vdash t' : R$. Par le lemme de permutation, nous avons également $\Gamma, y : T_1, x : S \vdash t' : R$. En utilisant le lemme d'affaiblissement avec $\Gamma \vdash s : S$, comme $y \notin \Gamma$, nous obtenons $\Gamma, y : T_1 \vdash s : S$. Nous appliquons alors l'hypothèse de récurrence avec t' et nous obtenons $\Gamma, y : T_1 \vdash [x \to s]t' : R$. Par T-ABS, nous avons $\Gamma \vdash [x \to s](\lambda y : T_1 . ([x \to s]t')) : R$.
— $t = u\, v$. Rappelons que par définition de la β-réduction,

$$[x \to s](u\, v) = ([x \to s]u)\, ([x \to s]v)$$

Par le lemme d'inversion, nous avons $\Gamma, x : S \vdash u : T_1 \to T$ et $\Gamma, x : S \vdash v : T$. Par hypothèse d'induction sur u et v, nous avons $\Gamma \vdash [x \to s]u : T_1 \to T$ et $\Gamma \vdash [x \to s]v : T_1$. Par T-APP et le rappel ci-dessus, nous pouvons conclure $\Gamma \vdash [x \to s](u\, v) : T$.

□

Théorème II.12 (de préservation du typage)**.** *Soit $\Gamma \vdash t : T$ et $t \to t'$. Alors $\Gamma \vdash t' : T$.*

Démonstration. Par induction sur l'arbre de dérivation de $\Gamma \vdash t : T$.
— $\Gamma \vdash x : T$. Ce cas n'est pas possible car aucun règle de réduction existe pour les variables.
— $\Gamma \vdash \lambda x : T_1 . t' : T$. Même chose que pour le cas des variables.
— $\Gamma \vdash uv : T$. Par le lemme d'inversion, nous avons $\Gamma \vdash u : T_1 \to T$ et $\Gamma \vdash v : T_1$. Plusieurs cas possibles :
  — u s'évalue en u'. Alors, $t' = u'\, v$ par E-APP1. Par hypothèse de récurrence sur l'arbre de dérivation $\Gamma \vdash u\, v : T$, nous obtenons $\Gamma \vdash u' : T_1 \to T$. Par T-APP, $\Gamma \vdash u'\, v : T$.
  — v s'évalue en v' et u est une valeur. Alors, $t' = u\, v'$ par E-APP2. Nous appliquons alors le même argument que pour le cas précedent.
  — u et v sont des valeurs. Posons $u = \lambda x : T_1 . t_1$. alors $t' = [x \to v]t_1$. Par le lemme d'inversion, nous obtenons $\Gamma, x : T_1 \vdash t_1 : T$. Par le lemme de substitution, nous concluons $\Gamma \vdash [x \to v]t_1 : T$.

□

Chapitre III

λ-calcul avec sous-typage et enregistrements.

La syntaxe des termes du λ-calcul simplement typé est pauvre : nous ne pouvons définir que des variables, des fonctions et appliquer des termes entre eux. La plupart des langages de programmation fournissent diverses structures de données comme les paires, les tuples, ou encore les enregistrements.

Un enregistrement est ensemble fini de couples (l_i, t_i), noté $\{l_i = t_i\}$ où l_i est un label pour le terme t_i, les couples étant séparés par des points-virgules. Par exemple, nous pouvons représenter un point d'un plan par ses coordonnées cartésiennes, nommées x et y et les termes t_x et t_y étant la valeur des coordonnées. Nous notons ce terme $\{x = t_x; y = t_y\}$. Il est également intéressant de pouvoir récupérer une des coordonnées d'un point. Nous ajoutons pour cela le terme $t.l$ qui permet de récupérer le label l du terme t.

Dans ce chapitre, nous allons étendre notre ensemble de termes avec les enregistrements ainsi que définir les règles d'évaluation et de typage pour ces nouveaux termes pour enfin introduire dans un second temps la notion de sous-typage qui définit le principe du *polymorphisme par sous-typage*.

Il nous faut également typer les enregistrements. Pour cela, nous ajoutons une nouvelle syntaxe dans les types, noté $\{l_i : T_i\}$ où l_i est un label de l'enregistrement et T_i le type du terme référencé par le label l_i. Pour l'exemple du point dans le plan, si nous supposons avoir un type R pour les réels, notre point serait de type $\{x : R; y : R\}$. Pour le typage des projections, il est naturel de dire que le type de $t.l_i$ soit le type du terme t_i de l'enregistrement t.

III.1 λ-calcul simplement typé avec enregistrements

Syntaxe

Formellement, la syntaxe des termes et la syntaxe des types sont définies par les grammaires III.1 et III.1.

$t ::=$		terme			
	x	var	$T ::=$		type
	$t\,t$	app		b	type de base
	$\lambda x.\,t$	abs		$t \to t$	fonction
	$\{l_i = t_i\}^{1 \leqslant i \leqslant n}$	enreg		$\{l_i : t_i\}^{1 \leqslant i \leqslant n}$	enreg
	$t.l$	proj			

Nous allons également ajouter les enregistrements dont tous les termes sont des valeurs comme valeurs de notre langage. Nous obtenons alors la grammaire III.1.

$v ::=$		valeur
	$\lambda x.\,t$	abs
	$\{l_i = v_i\}^{1 \leqslant i \leqslant n}$	enreg

Sémantique

Il nous faut également définir comment nous réduisons nos enregistrements. Nous ajoutons les règles d'évaluation suivantes aux règles d'évaluation définies dans les chapitres précédents.

$$\frac{t_j \to t'_j}{\begin{array}{c}\{l_1 = v_1; ...; l_j = t_j; ...; l_n = v_n\} \to \\ \{l_1 = v_1; ...; l_j = t'_j; ...; l_n = v_n\}\end{array}} \quad \text{(E-RCD)} \qquad \frac{t_1 \to t'_1}{t_1.l \to t'_1.l} \quad \text{(E-PROJ)}$$

$$\{l_i = v_i\}^{1 \leqslant i \leqslant n}.l_j \to v_j \quad \text{(E-PROJ-RCD)}$$

La règle (E-RCD) nous dit comment les termes à l'intérieur d'un record sont évalués. Quant à (E-PROJ-RCD), elle nous dit que nous pouvons effectuer une évaluer une projection uniquement si les termes de l'enregistrement ont tous été réduits à des valeurs. Pour finir, (E-PROJ) nous dit comment nous simplifions le terme dans une projection.

Règles de typage

En plus des règles de typages du λ-calcul simplement typé, la relation de typage comprend les règles définies par III.1.

$$\frac{\forall i \in \{1, ...n\}, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\} : \{l_i : T_i\}} \quad \text{(T-RCD)} \qquad \frac{\Gamma \vdash t_1 : \{l_i : T_i\}}{\Gamma \vdash t_1.l_j : T_j} \quad \text{(T-PROJ)}$$

La règle (T-RCD) nous dit comment introduire un type enregistrement tandis que (T-PROJ) nous dit comme typer une projection.

III.2 Sous-typage

Maintenant que nous avons définis la syntaxe, la sémantique et les règles de typages de notre langage comprenant les enregistrements, nous allons définir la notion de sous-typage ainsi que les règles pour ce langage.

III.3 Sureté

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.4 Top et Bottom

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

λ-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].

Un arbre de dérivation peut être infini.

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 α-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

Annexe A

Preuve par récurrence sur les termes et les types

Les termes ainsi que les types d'un langage sont définis de manière récursive. Par exemple, pour le λ-calcul simplement typé, la grammaire des termes est définie par

$t ::=$		terme
	x	var
	$t\,t$	app
	$\lambda x : T.\, t$	abs

et la grammaire des types, en supposant que nous avons uniquement `Bool` (pour les booléens) comme type de base, est définie par

$T ::=$		types
	$Bool$	type des booléens
	$T \to T$	type des fonctions

Ces définitions récursives sur les termes et les types nous permettent de définir récursivement des fonctions agissant sur les termes et les types. Par exemple, nous pouvons définir de manière inductive une fonction `size` sur les termes et les types de la manière suivante.

$$\begin{aligned} size(x) &= 1 \\ size(t_1 t_2) &= size(t_1) + size(t_2) \\ size(\lambda x : T.\, t) &= size(t) + 1 \end{aligned}$$

$$\begin{aligned} size(Bool) &= 1 \\ size(T_1 \to T_2) &= size(T_1) + size(T_2) \end{aligned}$$

Si nous nous représentons les termes et les types en forme d'arbre, la définition se résume à la définition du nombre de noeuds de l'arbre. Cette représentation et la définition de fonctions comme `size` nous permettent de raisonner par induction sur le nombre de noeuds de l'arbre en utilisant l'induction sur les naturels, comme le montre la preuve d'unicité de type pour le λ-calcul simplement typé. Une telle induction est appelée *induction structurelle*.

Nous supposons pour la plupart des grammaires que nous disposons d'une telle fonction qui permette de raisonner inductivement sur la structure des programmes ou des types, la fonction `size` étant souvent facile à définir.

Certaines preuves nécessitent une induction sur deux paramètres naturels comme celles du lemme d'affaiblissement et du lemme de permutation pour le λ-calcul simplement typé.

En effet, pour le lemme de permutation, pour le cas des abstractions, l'argument complet est :

« Par hypothèse et le lemme d'inversion, $\Gamma \vdash \lambda x : T_1.\, t' : T_1 \to T_2$ et $\Gamma, x : T_1 \vdash t' : T_2$. Par hypothèse de récurrence, $\Delta, x : T_1 \vdash t' : T_2$. Par $(T - ABS)$, $\Delta \vdash \lambda x : T_1.\, t' : T_1 \to T_2$. »

Cependant, l'hypothèse de récurrence est « $\Gamma \vdash t : T$ implique $\Delta \vdash t : T$ », et non « $\Gamma, x : T_1 \vdash t : T$ implique $\Delta, x : T_1 \vdash t : T$ » : le contexte n'est pas le même.

L'argument reste pourtant vrai : le principe de récurrence utilisé est celui sur $\mathbb{N}^2$ muni de l'ordre lexicographique en utilisant comme premier paramètre la taille du terme (qui diminue strictement dans le cas donné) et comme second la taille du contexte (qui augmente).

Pour rappel, le principe de récurrence sur $\mathbb{N}^2$ est le suivant :

Proposition A.1. *Soit P une proposition sur $\mathbb{N}^2$.*

Si, pour tout $(m, n) \in \mathbb{N}^2$, $P(m', n')$ est vrai pour tout $(m', n') \leqslant (m, n)$, alors, $P(m, n)$ est vrai pour tout $(m, n) \in \mathbb{N}^2$.

et se démontre en plusieurs lemmes :

Lemme A.2 (Principe d'induction sur un ensemble bien ordonné)**.** *Soit $(X, \leqslant)$ un ensemble bien ordonné, alors le principe d'induction est vrai sur X, c'est-à-dire :*

Soit P une proposition sur X. Si pour tout $x \in X$, quelque soit $y \in X$ tel que $y \leqslant x$, $P(y)$ est vrai, alors $P(x)$ est vrai pour tout $x \in X$.

Démonstration. Même principe que la preuve sur $\mathbb{N}$. □

Lemme A.3. *Soit $(X, \leqslant)$ un ensemble bien ordonné. Alors $(X^2, \leqslant_l)$ où $\leqslant_l$ est l'ordre lexicographique est bien ordonné.*

Démonstration. Soit $S \subseteq X^2$.

Posons $S_1 = \{x \in X \mid \exists y \in X \text{ tel que } (x, y) \in S\}$. Comme $S_1 \subseteq X$ et X est bien ordonné, S_1 possède un minimum. Notons le $\min S_1$.

Posons $T = \{y \in X \mid (\min S_1, y) \in S\}$. Comme $T \subseteq X$, T possède un minimum. Notons le $\min T$.

Alors, $s = (\min S_1, \min T)$ est le minimum de S. En effet, si $(x, y) \in S$, on a $x \in S_1$ car $(x, y) \in S$, donc $\min S_1 \leqslant x$ et si $\min S_1 = x$, alors, comme $y \in T$, $\min T \leqslant y$. Par construction, $s \in S$. □

De manière générale, le dernier lemme peut se démontrer, en utilisant les mêmes arguments, pour X^n où n est un naturel quelconque.

Nous gardons le terme *induction structurelle* qu'importe l'ensemble sur lequel nous utilisons le principe induction.

Bibliographie

[1] Goubault-Larrecq, J. Cours intitulé λ-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.