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 [1], 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 [2], 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. Malheureusement pas abordées dans ce sujet.
2. 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 donner, dans un premier temps et après avoir défini les notions nécessaires, un calcul typé, DOT[15], 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 classe, 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, ce travail apporte 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 DOT.

Les chapitres sont organisés afin de comprendre la construction de DOT à partir du plus simple des calculs, le λ-calcul.

Dans le chapitre 1, nous présenterons *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 introduirons 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 enrichirons 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. Dans le chapitre 4, nous parlerons de *polymorphisme paramétré* qui, combiné au λ-calcul simplement typé, forme le calcul appelé *System F*. Le chapitre 5 se chargera 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.

Ensuite, dans le chapitre 6, nous étudierons le calcul DOT en complétant 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

```
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.

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 enregistrements, le polymorphisme par sous-typage et le polymorphisme paramétré ainsi que les types récursifs.

Pour finir, dans le chapitre 7, nous discuterons de l'implémentation du langage RML[17] qui comprend un algorithme de typage et de sous-typage ainsi que d'un langage de surface pour le calcul DOT. Nous verrons que passer des règles de sous-typages à un algorithme n'est pas évident pour plusieurs raisons.

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 : $\lambda y.\, \lambda x.\, y$.
— l'application identité appliquée à la fonction identité : $(\lambda x.\, x)(\lambda y.\, y)$

Comme le montrent le dernier exemple, 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 la structure des termes 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 la structure des termes 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.

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

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 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.

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

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.

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.

```
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.

Cette stratégie appliquée à l'exemple donne :

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

Formellement, la stratégie call-by-value est définie par les *règles d'évaluation* définies ci-dessous, le terme v étant utilisé 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 typage et de sous-typage.

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 [2]. 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 [7]. Voici l'exemple des booléens, utilisé dans RML ([17]) :

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

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,

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

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 [1].

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*.

1. Plus d'informations peuvent être trouvées dans [8]

Définition II.2. *Soit B un ensemble de type appelé de* ***types de bases****. L'ensemble 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 $\emptyset$.

Le domaine de Γ, noté $dom(\Gamma)$, est l'ensemble des x_i.

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 [2]. Si Γ est vide, nous omettons $\emptyset$ et le jugement devient $\vdash t : T$.*

2. ou encore dans le contexte Γ

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.

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 [3] 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.

3. Un programme est synonyme de terme.

2. Préservation (du typage) : si un terme t de type T s'évalue en un terme t', alors t' est de type T.

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 [4]. 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. □

4. 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.

Progression

Théorème II.8 (de progression de $\lambda_{\to}$)**.** *Soit t un terme bien typé sans variable 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, par exemple x, n'est pas possible car par hypothèse le terme t est clos. Or, x est libre dans x.
- 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 $\vdash u\,v : T$ 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 l'arbre de dérivation de typage.

- 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 l'arbre de dérivation de typage.

- 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 $x \notin dom(\Gamma)$. Si $y \neq x$, $(y, T) \in \Gamma$ par le lemme d'inversion et par conséquent, $(y, T) \in \Gamma, x : S$ 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 du jugement $\Gamma, x : S \vdash t : T$ et ce en fonction de la forme de 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 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, 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 aucune 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 $\{l_i = t_i\}^{1 \leqslant i \leqslant n}$ 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 par les termes t_x et t_y pour 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.

Il nous faut également typer les enregistrements. Pour cela, nous ajoutons une nouvelle syntaxe dans les types, noté $\{l_i : T_i\}^{1 \leqslant i \leqslant n}$ 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 $\mathbb{R}$ pour les réels, notre point serait de type $\{x : \mathbb{R}; y : \mathbb{R}\}$. 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.

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*.

III.1 λ-calcul simplement typé avec enregistrements

Syntaxe

Formellement, la syntaxe des termes et la syntaxe des types sont définies par les grammaires suivantes :

$t ::=$		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 suivante :

$v ::=$		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 enregistrement sont évalués. Quant à (E-PROJ-RCD), elle nous dit que nous pouvons 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 suivantes :

$$\frac{\forall i \in \{1, ..., n\}, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\}^{1 \leqslant i \leqslant n} : \{l_i : T_i\}^{1 \leqslant i \leqslant n}} \quad \text{(T-RCD)}$$

$$\frac{\Gamma \vdash t_1 : \{l_i : T_i\}^{1 \leqslant i \leqslant n}}{\Gamma \vdash t_1.l_j : T_j} \quad \text{(T-PROJ)}$$

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

III.2 Sous-typage

Maintenant que nous avons défini 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.

Le sous-typage est une forme de polymorphisme, c'est-à-dire une possibilité d'attribuer plusieurs types à un terme. Le polymorphisme est très courant dans les langages orientés objets et permet, par exemple, d'élargir le champ d'application des fonctions. De manière plus concrêtes, supposons que nous avons défini un type $\mathbb{R}$ pour l'ensemble des réels et définissons le point $(5, 5)$ du plan $\mathbb{R}^2$ par $\{x = 5; y = 5\}$ [1]. Nous pouvons définir la fonction de projection sur l'axe des abscisses avec

$$\lambda p : \{x : \mathbb{R}; y : \mathbb{R}\} \,.\, p.x.$$

Maintenant, nous souhaitons faire de même pour le point $(5, 5, 5)$ de $\mathbb{R}^3$, représenté par $\{x = 5; y = 5; z = 5\}$. Nous ne pouvons pas utiliser la fonction de projection définie pour $\mathbb{R}^2$ parce que le paramètre de la fonction est un enregistrement ne contenant que les champs x et y. Nous devons donc créer une nouvelle fonction, par exemple

$$\lambda p : \{x : \mathbb{R}; y : \mathbb{R}; z : \mathbb{R}\} \,.\, p.x.$$

Et si nous voulions continuer pour $\mathbb{R}^4$, $\mathbb{R}^5$, etc, nous devrions à chaque fois redéfinir une nouvelle fonction. Cependant, nous remarquons que le corps de la fonction est toujours le même et qu'il n'a besoin que d'un enregistrement avec au moins un champ x, les autres champs étant inutiles.

C'est dans ce cas que le sous-typage intervient : nous allons définir une relation entre les types qui permet d'affiner les règles de typage. Nous dirons que S **est un sous-type de** T ou encore T **est un supertype de** S, noté $S <: T$.

Pour lier la relation de typage à la relation de sous-typage, nous ajoutons une nouvelle règle de typage

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

Cette dernière permet d'affirmer que si dans un contexte donné, un terme t a le type S et que le type S est un sous-type de T, alors dans le même contexte, t a le type T.

Règles de sous-typage

Passons à la définition de la relation de sous-typage. La toute première chose est que nous voulons que cette relation soit réflexive et transitive comme c'est le cas dans la plupart des langages :

1. En supposant que 5 fasse partie de notre syntaxe et que son type soit $\mathbb{R}$.

$$S <: S \quad \text{(S-REFL)} \qquad \frac{S <: T \qquad T <: U}{S <: U} \quad \text{(S-TRANS)}$$

Ensuite, nous aimerions que la relation résolve notre problème, c'est-à-dire que nous devons pouvoir affirmer qu'un type ayant les champs x, y, z est un sous-type du type ayant uniquement le champ x ou uniquement le champ y. Nous résumons cela par les deux règles suivantes :

$$\{l_i : T_i\}^{1 \leqslant i \leqslant n+k} <: \{l_i : T_i\}^{1 \leqslant i \leqslant n} \quad \text{(S-RCD-WIDTH)}$$

$$\frac{\{k_j : s_j\}^{1 \leqslant j \leqslant n} \text{ permutation de } \{l_i : t_i\}^{1 \leqslant i \leqslant n}}{\{k_j : s_j\}^{1 \leqslant j \leqslant n} <: \{l_i : t_i\}^{1 \leqslant i \leqslant n}} \quad \text{(S-RCD-PERM)}$$

(S-RCD-WIDTH) nous permet de « laisser de coté »certain champ tandis que (S-RCD-PERM) nous dit que l'ordre des champs dans un enregistrement n'a pas d'importance.

Nous souhaitons également prendre en compte les types dans les enregistrements et leur relation de sous-typages entre eux. Par exemple, nous aimerions qu'un point du plan $\mathbb{N}^2$ soit également un point du plan $\mathbb{R}^2$.

$$\frac{\forall i \in \{1, ..., n\}, S_i <: T_i}{\{l_i : S_i\}^{1 \leqslant i \leqslant n} <: \{l_i : T_i\}^{1 \leqslant i \leqslant n}} \quad \text{(S-RCD-DEPTH)}$$

(S-RCD-DEPTH) nous dit que, étant donnés deux types enregistrements S et T avec les mêmes labels, si les types des champs de S sont tous sous-types des types des champs de T, alors S est sous-type de T.

Pour finir, nous aimerions dire que si nous avons une fonction f qui attend un enregistrement avec un champ x, nous pouvons également passer en paramètre à f un enregistrement avec deux champs x et y. Nous ajoutons en plus que le type de retour peut être un super-type.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-ARROW)}$$

Nous avons maintenant toutes les propriétés nécessaires pour résoudre notre problème. En effet, voici un arbre de dérivation qui permet de montrer $(\lambda p : \{x : R\}\,.\,p.x)\,\{x = 5; y = 5\} : R$.

$$\text{(T-APP)}\ \frac{\Gamma \vdash \lambda p : \{x : R\}\,.\,p.x : \{x : R\} \to R \qquad \text{(T-SUB)}\ \dfrac{\Gamma \vdash \{x = 5; y = 5\} : \{x : R; y : R\} \qquad \{x : R; y : R\} <: \{x : R\}}{\Gamma \vdash \{x = 5; y = 5\} : \{x : R\}}}{\Gamma \vdash (\lambda p : \{x : R\}\,.\,p.x)\,\{x = 5; y = 5\} : R}$$

L'affirmation $\{x : R; y : R\} <: \{x : R\}$ est inférée grâce à (S-RCD-WIDTH).

III.3 Sureté

Nous allons montrer que les théorèmes de préservation et de progression démontrés pour le λ-calcul simplement typé restent vrais en présence d'enregistrements et du sous-typage.

Les techniques de preuves utilisées et leur structure ne sont pas très différentes de celles employées dans le chapitre précédent. Pour ces raisons, les preuves seront moins détaillées.

Préservation

Lemme III.1 (Inversion de la règle de sous-typage)**.** *1. Si $S <: T_1 \to T_2$, alors S est de la forme $S_1 \to S_2$ avec $T_1 <: S_1$ et $S_2 \to T_2$.*

2. Si $S <: \{l_i : T_i\}^{1 \leqslant i \leqslant n}$, alors S est de la forme $\{k_i : S_i\}^{1 \leqslant i \leqslant m}$ tel que $(k_i)_{1 \leqslant i \leqslant m} \subseteq (l_i)_{1 \leqslant i \leqslant n}$ et $S_i <: T_i$ pour chaque i tel que $l_i = k_i$.

Démonstration. La technique de preuve reste identique : pour chaque affirmation, nous cherchons quelle règle peut y avoir mené et nous montrons cas par cas, en utilisant le principe d'induction sur l'arbre de dérivation. Seul le premier point est démontré, le second étant identique. Pour le premier point, seules les règles (S-REFL), (S-ARROW) et (S-TRANS) sont possibles, les autres étant pour les enregistrements.

— (S-REFL). Le résultat est direct car $S = T_1 \to T_2$.

— (S-ARROW). Le résultat est également direct.

— (S-TRANS). Nous avons donc l'arbre de dérivation suivant :

$$\frac{S <: T \qquad T <: T_1 \to T_2}{S <: T_1 \to T_2}$$

Nous appliquons d'abord l'hypothèse de récurrence sur l'affirmation $T <: T_1 \to T_2$ et ensuite sur l'affirmation $S <: T$.

□

Nous montrons également un lemme d'inversion des règles de typage comme il a été démontré pour le λ-calcul simplementé typé.

Lemme III.2 (d'inversion des règles de typage)**.** *1. Si $\Gamma \vdash \lambda x : S_1 . s_2 : T_1 \to T_2$, alors $T_1 <: S_1$ et $\Gamma, x : S_1 \vdash s_2 : T_2$.*

2. Si $\Gamma \vdash \{k_i = s_i\}^{1 \leqslant i \leqslant n} : \{l_i : T_i\}^{1 \leqslant i \leqslant m}$, alors $(l_i)^{1 \leqslant i \leqslant m} \subseteq (k_i)^{1 \leqslant i \leqslant n}$ et $\Gamma \vdash s_i : T_i$ pour $k_i = l_i$.

Démonstration. Par induction sur l'arbre de typage de $\Gamma \vdash t : T$. Encore une fois, nous ne montrons que pour le premier cas, les arguments étant sensiblement les mêmes pour le second.

Les seuls règles possibles sont T-SUB ou T-ABS.

1. T-SUB. Nous avons donc l'arbre de dérivation suivant :

$$\text{(T-SUB)}\ \frac{\Gamma \vdash \lambda x : S_1 . s_2 : T \qquad T <: T_1 \to T_2}{\Gamma \vdash \lambda x : S_1 . s_2 : T_1 \to T_2}$$

En appliquant le lemme III.1 sur l'affirmation $T <: T_1 \to T_2$, nous obtenons $T = \tilde{S_1} \to \tilde{S_2}$ avec $T_1 <: \tilde{S_1}$ et $\tilde{S_2} <: T_2$. L'affirmation de gauche devient donc

$$\Gamma \vdash \lambda x : S_1 . s_2 : \tilde{S_1} \to \tilde{S_2}$$

En appliquant l'hypothèse de récurrence sur ce jugement de typage, nous obtenons $\tilde{S_1} <: S_1$ et $\Gamma, x : S_1 \vdash s_2 : \tilde{S_2}$.
En utilisant S-TRANS avec $\tilde{S_1} <: S_1$ et $T_1 <: \tilde{S_1}$, nous concluons avec $T_1 <: S_1$. Pour finir, en utilisant T-SUB avec $\Gamma, x : S_1 \vdash s_2 : \tilde{S_2}$ et $\tilde{S_2} <: T_2$, nous obtenons $\Gamma, x : S_1 \vdash s_2 : T_2$.

2. T-ABS. Le résultat est direct et $T_1 = S_1$.

□

Nous pouvons maintenant démontrer le même lemme de substitution défini dans le chapitre précédent.

Lemme III.3 (de préservation de typage par substitution)**.** *Soit* $\Gamma, x : S \vdash t : T$ *et* $\Gamma \vdash s : S$.
Alors $\Gamma \vdash [x \to s]t : T$

Démonstration. Même technique de preuve que dans le chapitre précédent. Il suffit d'ajouter des cas pour les enregistrements et les projections en utilisant respectivement T-RCD et T-PROJ et de manière générale pour T-SUB (hypothèse de récurrence sur le jugement de gauche et utilisation de T-SUB pour conclure). □

Et nous concluons avec le théorème de progression.

Théorème III.4 (de préservation du typage)**.** *Si t est un terme bien typé sans variable libre, alors soit t est une valeur, soit il existe t' tel que $t \to t'$.*

Démonstration. Nous procédons de la même manière que pour le cas du λ-calcul simplement typé, c'est-à-dire sur le jugement $\Gamma \vdash t : T$.

1. T-VAR. Nous avons alors $t = x$: pas possible car il n'y a pas de réduction pour les variables.
2. T-ABS. $t = \lambda x : T_1 . t'$: déjà une valeur.
3. T-APP. $t = t_1\, t_2$. Nous avons $\Gamma \vdash t_1 : T_1 \to T$, $\Gamma \vdash t_2 : T_1$. Les possibles règles d'évaluation sont E-APP1, E-APP2 et E-APPABS. Pour E-APP1 et E-APP2, même raisonnement que pour le λ-calcul simplement typé.
   Quant à E-APPABS, nous obtenons $t_1 = \lambda x : S_1 . t'$ et $t_2 = v$. En utilisant III.3, nous déduisons $T_1 <: S_1$ et $\Gamma, x : S \vdash t' : T$. Nous concluons avec le lemme de substitution qui donne $\Gamma, x : S \vdash [x \to v]t' : T$.
4. T-RCD. La seule règle d'évaluation est E-RCD qui réduit un des termes de l'enregistrement. Il suffit d'appliquer l'hypothèse de récurrence sur ce terme et d'utiliser T-RCD pour conclure.
5. T-PROJ. Nous obtenons $t = t_1.l_j$, $\Gamma \vdash t_1 : \{l_i : T_i\}^{1 \leqslant i \leqslant n}$ et $T = T_j$. Deux règles d'évaluation sont possibles : E-PROJ ou E-PROJ-RCD. Si c'est E-PROJ, nous appliquons l'hypothèse de récurrence. Sinon, E-PROJ-RCD nous dit que $t_1 = \{k_i = v_i\}^{1 \leqslant i \leqslant m}$ et par le lemme III.1, nous avons $(l_i)_{1 \leqslant i \leqslant n} \subseteq (k_i)_{1 \leqslant i \leqslant m}$ et $\Gamma \vdash v_i : T_i$ pour $k_i = l_i$. Nous concluons que $\Gamma \vdash v_j : T_j$.

6. T-SUB. Par hypothèse de récurrence et en utilisant T-SUB.

□

Progression

Nous démontrons avant un lemme appelé *lemme des formes canoniques.*

Lemme III.5 (des formes canoniques)**.** *Soit v une valeur sans variable libre.*

1. *Si v est de type $T_1 \to T_2$, alors v est de la forme $\lambda x : S_1 . t$.*
2. *Si v est de type $\{l_i : T_i\}^{1 \leqslant i \leqslant n}$, alors v est de la forme $\{k_i = v_i\}^{1 \leqslant i \leqslant m}$ où $(l_i)_{1 \leqslant i \leqslant n} \subseteq (k_i)_{1 \leqslant i \leqslant m}$.*

Démonstration. □

Théorème III.6 (de progression)**.** *Soit t un terme bien typé sans variable libre. Alors soit t est une valeur, soit t se réduit en t'*

Démonstration. Par induction sur l'arbre de dérivation de typage de t. □

III.4 Type Top et type Bottom

Finalement, il est courant d'ajouter dans les types un type qui est super-type de tous les autres types, souvent appelé `Top`[2] ainsi qu'un type qui est sous-type de tous les autres types, souvent appelé `Bottom`.

La syntaxe des types est donc finalement

$T ::=$		type
	b	type de base
	$t \to t$	fonction
	$\{l_i : t_i\}^{1 \leqslant i \leqslant n}$	enreg
	Top	Top
	$Bottom$	Bottom

et nous ajoutons les règles de typage suivantes.

$$\text{(S-TOP)}\ T <: Top \qquad\qquad Bottom <: T\ \text{(S-BOTTOM)}$$

2. En Java, le type Top est représenté par la classe Object.

Chapitre IV

System F

Dans ce chapitre, nous allons introduire une autre de notion de polymorphisme appelée le *polymorphisme paramétré*[1]

Nous avons vu dans le chapitre précédent que le polymorphisme par sous-typage nous permet de rendre notre relation de typage plus flexible en donnant la possibilité d'attribuer plusieurs types à un terme grâce à la règle T-SUB. Cette méthode nous permet alors d'éviter d'écrire plusieurs fonctions qui ont le même corps mais qui différent uniquement par le type du paramètre.

Supposons que nous avons $\mathbb{N}$ et $\mathbb{R}$ comme types dans notre langage et prenons l'exemple de la fonction identité sur les réels

$$id_1 = \lambda x : \mathbb{R}.\, x$$

Le polymorphisme avec sous-typage nous permet de passer en paramètre un naturel car $\mathbb{N} <: \mathbb{R}$. Cependant, il n'est pas autorisé de passer un enregistrement ayant un champ x de type $\mathbb{N}$ car $\mathbb{R}$ n'est pas un sous-type de $\{x : \mathbb{N}\}$. Nous devons alors définir une nouvelle fonction pour l'identité :

$$id_2 = \lambda x : \{x : \mathbb{N}\}\, .\, x$$

Et nous pouvons continuer de la sorte avec les enregistrements qui ont uniquement le champ y ou le champ z et ainsi de suite.

Le polymorphisme paramétré résout ce problème en définissant de nouveaux termes et de nouvelles règles de typage et d'évaluation. Le calcul qui en résulte est appelé *System F*.

IV.1 Syntaxe

La syntaxe de *System F* est très proche de celle du λ-calcul simplement typé : nous ajoutons un terme qui permet de créer une application prenant un type et retournant un terme (l'équivalent de l'abstraction sur les termes) ainsi qu'un terme qui permet d'appliquer un type à un terme.

Du coté des types, nous ajoutons des variables de type qui nous servent dans les abstractions de type. Nous ajoutons également un type pour les abstractions de type, appelé *type universel*.

1. Plus d'informations peuvent être trouvées dans [4]

Arbitrairement, nous ajoutons aussi les abstractions de type comme valeurs.

$t ::=$	terme
x	var
$t\,t$	app
$\lambda x : T.\, t$	abs
$\Lambda X.\, t$	type abs
$t[T]$	type app

$T ::=$	type
X	type var
$T \to T$	fonction
$\forall X.\, T$	universel

$v ::=$	valeur
$\lambda x : T.\, t$	abs
$\Lambda X.\, t$	type abs

Par exemple, nous pouvons définir une fonction qui prend un type en paramètre et qui renvoie la fonction identité pour ce type :

$$id_{poly} = \Lambda X.\, (\lambda x : X.\, x)$$

Nous étendons également de manière naturelle la notion de substitution aux nouveaux termes $t[T]$ et $\Lambda X.\, t$.

$$[x \to s]\,(t[T]) = ([x \to s]t)[T]$$
$$[x \to s]\,(\Lambda X.\, t) = \Lambda X.\, ([x \to s]t)$$

IV.2 Sémantique

Nous ajoutons également des règles d'évaluation pour les nouveaux termes définis. Celles-ci sont très semblables à (E-APPABS) et (E-APP).

$$\text{(E-T-APP)}\ \frac{t \to t'}{t[T] \to t'[T]} \qquad\qquad (\Lambda X.\, t)[T] \to [X \to T]t\ \text{(E-T-ABS)}$$

(E-T-APP) est l'équivalent de (E-APP1) pour les applications de types tandis que (E-T-ABS) nous dit comment les abstractions de types sont évaluées. La notation $[X \to T]t$ signifie que nous remplaçons la variable de type X par le type T dans le terme t [2].

Par exemple, nous pouvons utiliser id_{poly} et les types $\mathbb{R}$ et $\mathbb{N}$ pour obtenir la fonction identité sur les réels et les naturels.

$$id_{poly}[\mathbb{N}] \to \lambda x : \mathbb{N}.\, x \qquad\qquad id_{poly}[\mathbb{R}] \to \lambda x : \mathbb{R}.\, x$$

2. Un travail devrait être fait pour redéfinir les notions de variables de type libres, de variables de type liées ainsi que la substitution. Cependant, le travail est xactement le même pour les variables.

IV.3 Contexte de typage

Avant de donner les règles de typage, il est important de remarquer que nous devons également changer notre définition du contexte. En effet, le but initial du contexte est de contenir les variables libres d'un terme. Cependant, des variables de type sont également définies et peuvent être également libres dans des termes comme dans $\lambda z : \{x : X; y : X\}\,.\,z$. Cela nous amène à ajouter les variables dans le contexte afin de garder une trace de leur définition. Le contexte de typage est donc défini comme une liste contenant des couples $(x : T)$ et des variables de type X.

Nous supposons que si une variable X est dans le contexte, alors elle ne peut pas apparaître à la gauche de sa définition. Par exemple, le contexte $x : X, X$ n'est pas valable, mais $X, x : X$ l'est.

IV.4 Règles de typage

Enfin, nous ajoutons des règles de typage pour les nouveaux termes $\Lambda X.\,T$ et $t[T]$.

$$\text{(T-T-ABS)}\ \frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.\,t : \forall X.\,T} \qquad \frac{\Gamma \vdash t_1 : \forall X.\,T}{\Gamma \vdash t_1[T'] : [X \to T']T}\ \text{(T-T-APP)}$$

Ces règles nous permettent de typer nos termes id_{poly}, $id_{poly}[\mathbb{R}]$ et $id_{poly}[\mathbb{N}]$ qui sont respectivement de types $\forall X.\,X \to X$, $\mathbb{R} \to \mathbb{R}$ et $\mathbb{N} \to \mathbb{N}$.

IV.5 Sureté

Les théorèmes de préservation et de progression peuvent également être démontrés pour System F. Comme nous avons pu le remarquer, System F est une extension relativement simple du λ-calcul simplement typé. Les nouvelles règles de typage et d'évaluation ont de l'influence uniquement sur les nouveaux termes et les nouveaux types. Les preuves des théorèmes sont donc très semblables. Comme pour le chapitre précédent, les preuves seront moins détaillées.

Progression

Nous commençons par ajouter de nouveaux cas dans le lemme d'inversion des règles de typage II.6, les cas du λ-calcul simplement typé restant vrais.

Lemme IV.1 (d'inversion des règles de typage)**.** *1. Si* $\Gamma \vdash \Lambda X.\,t : \forall X.\,T$*, alors* $\Gamma, X \vdash t : T$*.*

2. Si $\Gamma \vdash t[T_1] : T_2$*, alors* $\Gamma \vdash t : \forall X.\,T$ *où* $T_2 = [X \to T_1]T$*.*

Démonstration. Même argument que II.2. □

Nous montrons également un lemme des formes canoniques comme nous l'avons fait pour le λ-calcul avec sous-typage.

Lemme IV.2 (des formes canoniques)**.** *1. Si* $\Gamma \vdash v : \forall X.\,T$*, alors* $v = \Lambda X.\,t$ *et* $\Gamma, X \vdash t : T$*.*

2. *Si* $\Gamma \vdash v : T_1 \to T_2$, *alors* $v = \lambda x : T_1 . t$ *et* $\Gamma, x : T_1 \vdash t : T_2$.

Démonstration. Direct vu les règles de typage. □

Théorème IV.3 (de progression)**.** *Soit t un terme bien typé sans variable libre. Alors, soit t est une valeur, soit il existe* t' *tel que* $t \to t'$.

Démonstration. Les seuls nouveaux cas sont $t = \Lambda X . t_1$ et $t = t_1[T_2]$.

Dans le premier cas, t est une valeur donc le résultat est direct.

Dans le second cas, par le lemme d'inversion, nous avons $\vdash t_1 : \forall X\, T_1$. Nous appliquons l'hypothèse de récurrence sur t_1 et utilisons (E-T-APP) ou (E-T-ABS) en fonction du résultat de l'appel récursif. □

Préservation

Nous montrons d'abord un lemme de substitution équivalent à **??**.

Lemme IV.4 (de substitution de variable de type)**.** . *Soit S un type.*
Si $\Gamma, X, \Delta \vdash t : T$, *alors*

$$\Gamma, [X \to S]\Delta \vdash [X \to S]t : [X \to S]T \tag{IV.1}$$

où $[X \to S]\Delta$ *signifie que nous remplaçons toutes les occurences de X par S dans le contexte* Δ.

Démonstration. La preuve se réalise par récurrence sur l'arbre de dérivation en analysant cas par cas le terme et le type. Le contexte Δ est nécéssaire pour l'appel récursif pour le cas $T - ABS$. □

Théorème IV.5 (de préservation du typage)**.** . *Soit* $\Gamma \vdash t : T$ *et* $t \to t'$. *Alors* $\Gamma \vdash t' : T$.

Démonstration. . Identique à la preuve du λ-calcul simplement typé. Le cas $t = \lambda x : T_1 . t_1$ n'est pas possible car il n'existe pas de règle d'évaluation.

Pour le cas $t = t_1[T_1]$, nous utilisons le lemme d'inversion pour obtenir $\Gamma \vdash t_1 : \forall X . T_{12}$ et $T = [X \to T_1]T_{12}$. Nous appliquons ensuite l'hypothèse de récurrence sur t_1 et en fonction du résultat, nous conclusion avec (E-T-ABS) ou (E-T-APP) en utilisant IV.4. □

Chapitre V

System $F_{<:}$

Dans les chapitres III et IV, nous avons défini deux notions de polymorphisme : le polymorphisme par sous-typage à travers les enregistrements et le polymorphisme paramétré.

Les deux calculs permettent de résoudre deux problèmes différents :

1. le sous-typage permet d'affiner notre relation de typage à travers la relation $<:$. Par exemple, il nous est permi de définir la fonction identité sur les réels ($\lambda x : \mathbb{R}.\, x$) et de l'appliquer à un entier car $\mathbb{N}$ est un sous-type de $\mathbb{R}$.

2. le polymorphisme paramétré permet de quantifier sur les types et de créer des fonctions indépendamment du types grâce aux abstractions de types pour ensuite les appliquer aux types souhaités. Par exemple, nous pouvons définir la fonction identité polymorphe $\Lambda X.\, \lambda x : X.\, x$ et les appliquer aux types $\mathbb{R}$ et $\mathbb{N}$, mais également à n'importe quel autre type comme $\mathbb{N} \to \mathbb{R}$ ou encore $\mathbb{R} \to (\mathbb{N} \to \mathbb{R})$.

Dans ce chapitre, nous allons unifier ces deux notions en un langage appelé System $F_{<:}$ [1]. L'idée principale de System $F_{<:}$ est d'élargir notre relation de sous-typage sur les variables de types pour les borner et ainsi restreindre les types qui peuvent être appliqués aux abstractions de type. De cette façon, une même abstraction de type dépendante d'une variable X, bornée supérieurement par $\mathbb{R}$, pourra accepter les types $\mathbb{N}$ et $\mathbb{R}$, mais pas $\mathbb{N} \to \mathbb{R}$ ou encore $\mathbb{R} \to \mathbb{R}$. Nous ne considérons donc plus seules les variables de types, mais liées à une borne supérieure. Nous notons $X <: T$ pour dire que la variable de type X est borné supérieurement par T.

Ajouter une borne supérieure permet d'affiner le type de la variable X en obligeant le type appliqué à avoir certaines caractéristiques. Par exemple, la fonction $\Lambda X <: \{z : \mathbb{R}\}\, .\, \lambda x : X.\, x.z$ oblige, lors d'une application de type, de donner un type enregistrement avec au moins le champ z qui est de type réel.

1. prononcé « system F sub »

V.1 Syntaxe

V.2 Sémantique

V.3 Contexte de typage

V.4 Règles de typage

V.5 Sureté

Les théorèmes de préservation et de progression restent vrais pour System $F_{<:}$. Cependant, nous ne les démontrerons pas car ils nécessitent des lemmes techniques et ce n'est pas le sujet principal de ce document. La structure et les techniques restent les mêmes : lemme d'inversion des relations de typage, lemme des formes canoniques, lemme d'inversion de la relation de sous-typage, lemme de substitution des types, preuve par induction structurelle, etc. Des preuves des théorèmes peuvent être trouvées dans [5].

Chapitre VI

Enregistrement avec type chemin dépendant

Faire le lien avec le sujet initial.

VI.1 Syntaxe

VI.2 Sémantique

La sémantique est laissée de coté actuellement.

VI.3 Règles de typage

VI.4 Règles de sous-typage

Pouvoir définir des record récursifs.

Problème d'échappement

VI.5 Encodage de System $F_{<:}$

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

VI.6 Notion de bonne formation

VI.7 Sureté

Ne pas tout démontrer, voir les théorèmes dans WF.

Chapitre VII

RML : implémentation

Dans ce chapitre, nous allons discuter de l'apport principal de ce travail qui est l'implémentation d'un algorithme de typage, d'un algorithme de sous-typage ainsi que d'un langage de surface basé sur le calcul théorique du chapitre précédent, DOT. Nous parlerons également des difficultés rencontrées lors de l'implémentation d'un calcul théorique et nous remarquerons que DOT nécessite des règles et des termes supplémentaires pour résoudre certains problèmes d'implémentation.

Le code de RML peut être trouvé sur Github[17]. RML est développé entièrement en OCaml en utilisant plusieurs dépendances comme `ocamllex` comme lexeur, `menhir`[11] pour la génération du parseur, `AlphaLib`[10] pour la gestion des variables, `PPrint`[12] pour pouvoir afficher plus clairement les termes et les types et enfin `ANSITerminal`[16] pour ajouter des couleurs lors de l'affichage.

Sur la page principale se trouve, en anglais, une description complète de la syntaxe de RML, du but du langage ainsi que des exemples. Cette documentation est plus orientée pour les développeurs ayant une base en OCaml et les personnes souhaitant utiliser le langage pour écrire des programmes ; les concepts théoriques sont donc omis. La structure du projet, la méthode de compilation et l'exécution des algorithmes sont clairement expliquées sur la page du projet. Pour ces raisons, ces détails ne seront pas dupliqués dans ce document. Il est fortement conseillé au lecteur de lire la page principale pour avoir une idée générale avant de continuer.

VII.1 Langage de surface

Par langage de surface, nous désignons une syntaxe plus simple et plus pratique à utiliser pour écrire des programmes d'un calcul. Pour RML, la syntaxe d'OCaml est utilisée. En particulier, la syntaxe des modules OCaml est utilisée pour les termes récursifs $\nu(x : T)d$ ou encore la syntaxe des types récursifs $\mu(x : T)$ est remplacée par la syntaxe des signatures des modules en OCaml.

Le langage de surface est entièrement décrit sur la page du projet[17]. Dans la suite, nous utiliserons aussi bien la syntaxe du langage de surface que la syntaxe de DOT selon le besoin et la facilité d'écriture.

VII.2 Implémentation des grammaires

Les grammaires des termes, des types et des déclarations, définies dans le fichier `grammar/grammar.cppo.ml`, sont implémentées par des types sommes appelés respectivement `term`, `typ` et `decl`. Chaque type est paramétré par deux types `'bn` et `'fn` qui représentent respectivement le type des variables liées et le type des variables libres.

Gestion des variables

La gestion des variables liées et des variables libres est l'une des tâches les plus fastidieuses lors de l'implémentation d'un langage. En effet, il est nécessaire de gérer l'unicité des variables, le renommage ou encore l'environnement pour se souvenir des variables déjà utilisées, ce dernier grandissant quand nous rentrons dans un lambda et se réduisant quand nous en sortons. De plus, cette tâche n'est pas très intéressante au niveau algorithmique, implique très souvent des erreurs d'inattention et il existe diverses méthodes pour représenter et gérer les variables.

AlphaLib[10] est une librairie basée sur visitors[13] qui fournit des macros [1] générant des fonctions pour gérer les variables. Ces macros permettent par exemple :

1. de définir plusieurs représentations des variables. Dans RML, nous utilisons deux représentations : *brute* (types `raw_term`, `raw_typ` et `raw_decl`) et *nominative* (types `nominal_term`, `nominal_typ` et `nominal_decl`). Dans la représentation dite brute, les variables sont représentées par des chaines de caractères tandis que dans la représentation nominative, les variables sont des atomes représentés par un type `AlphaLib.Atom.t`, fourni par AlphaLib. Cette dernière attribue à chaque nouvelle variable un entier unique pour obtenir une représentation unique.
2. de passer d'une représentation à une autre. La représentation brute est utilisée par le parseur et la conversion vers la représentation nominative est réalisée directement après la lecture du parseur.
3. de récupérer toutes les variables libres ou liées d'un terme.
4. dans le cas de la représentation nominative, de générer des nouveaux atomes.
5. d'utiliser des types polymorphes prédéfinis comme `abs` qui représentent de manière générale le comportement d'une abstraction. Lorsqu'un type `abs` est rencontré, la variable de l'abstraction est automatiquement ajoutée dans l'environnement. `abs` est utilisé dans RML pour les types récursifs, les abstractions et la gestion des fonctions dépendantes.

VII.8 montre l'implémentation concrète des termes.

Plus d'information et d'explications sur AlphaLib et son fonctionnement peuvent être trouvées dans [14].

1. en utilisant cppo, d'où l'extension cppo.ml pour le fichier définissant la grammaire.

```
type ('bn, 'fn) term =
  (* x *)
  | TermVariable of 'fn
  (* lambda(x : S) t --> (S, (x, t)) *)
  | TermAbstraction of
      ('bn, 'fn) typ * ('bn, ('bn, 'fn) term) abs
  (* x y *)
  | TermVarApplication of 'fn * 'fn
  (* let x = t in u --> (t, (x, u))*)
  | TermLet of
      ('bn, 'fn) term * ('bn, ('bn, 'fn) term) abs
  (* nu(x : T) d *)
  | TermRecursiveRecord of
      ('bn, 'fn) typ * ('bn, ('bn, 'fn) decl) abs
  (* x.a *)
  | TermFieldSelection of 'fn * field_label

type raw_term = (string, string) term
type nominal_term = (AlphaLib.Atom.t, AlphaLib.Atom.t) term
```

Code OCaml 5: Implémentation de la grammaire des termes officiels de DOT en utilisant AlphaLib.

VII.3 Contexte de typage

Un contexte est implémenté comme un dictionnaire dont les clefs sont des atômes et les valeurs sont les types nominaux de ces atômes. Cette implémentation se trouve dans le fichier `typing/contextType.ml` et contient également des fonctions pour afficher un contexte donné.

VII.4 Bornes d'un type dépendant

Un algorithme important est celui qui permet de donner la meilleure borne d'un type dépendant. C'est-à-dire, étant donnés une variable x et un contexte, quelle est la borne inférieure ou la borne supérieure du type dépendant $x.t$? La question se pose, par exemple, quand nous devons comparer deux types dépendants $x.t$ et $y.t$.

En d'autres termes, pour la borne supérieure, pour une variable x, de type T et un champ t, quel est le plus petit type U tel que $T <: \{t : L..U\}$. La fonction implémentant cet algorithme est `least_upper_bound_of_recursive_type` (pour la borne inférerieure, la fonction est `greater_lower_bound_of_recursive_type`) dans le fichier `typing/TypeUtils.ml`.

Remarquons d'abord que la question n'a pas toujours de réponse. En effet, si x est de type `Top`[2], le seul super-type de T est `Top`. En OCaml, nous représentons cela par un type `option` où `None` est renvoyé si la question n'a pas de réponse.

2. Bien que la question n'ait pas de sens, nous pouvons la poser car les règles ne l'empêchent pas.

L'algorithme travaille sur la structure de T :

- Si T est `Bottom`, la réponse est `Bottom`.
- Si T est `Top`, il n'y a pas de réponse, l'algorithme renvoie donc `None`.
- Si T est une fonction, il n'y a également pas de réponse car nous ne pouvons comparer un enregistrement avec une fonction. L'algorithme renvoie `None`.
- Si T est un type récursif, nous utilisons la règle (VAR-UNPACK) et appelons récursivement l'algorithme.
- Si T est un type intersection, par exemple $T_1 \wedge T_2$, nous appliquons l'algorithme récursivement sur T_1 et sur T_2. Plusieurs cas sont possibles :
  - Si les deux appels retournent `None`, l'algorithme retourne `None` car cela signifie qu'il n'y a pas de réponse. Cela peut se passer si T_1 et T_2 sont des fonctions.
  - Si l'un des deux retourne une valeur, nous la retournons.
  - Si les deux retournent un type, nous renvoyons la valeur retournée pour le type T_2.
- Si T est une déclaration de type, par exemple $\{t : L..U\}$, la réponse est U.
- Si T est une déclaration de champ, par exemple $\{t : U\}$, la réponse est U
- Le dernier cas est si T est de la forme $y.t'$. Un appel récursif est effectué sur le type de y avec le champ t'. Si la réponse est `None`, `None` est renvoyé car cela signifie qu'il n'y avait déjà pas de réponse pour y (par exemple, si y est de type fonction) : il ne peut donc pas en avoir pour x. Si un type U est renvoyé, alors, nous avons le plus petit U tel que $T_y <: \{t' : L..U\}$ où T_y est le type de y. En supposant que l'algorithme donne bien le plus petit U, nous savons alors que U est la borne supérieure de T, c'est-à-dire le plus petit super-type de T. Nous retournons alors le résultat de l'appel récursif sur U avec le champ t.

L'algorithme actuel est satisfaisant sur différents exemples. Cependant, nous remarquons que le cas où $T = y.t'$ n'est pas très clair. Il serait nécessaire de montrer que la réponse pour T est la même que la réponse pour le U donné par l'appel récursif. [3] Nous pouvons seulement dire que la réponse actuelle est un candidat, mais non le meilleur.

L'algorithme n'est également pas exact dans le cas des intersections. En effet, dans le cas où les deux retournent une valeur, il faudrait retourner le plus petit type. Cependant, cela implique une dépendance de module avec l'algorithme de sous-typage, ce qui n'est pas possible vu l'implémentation actuelle car l'algorithme de sous-typage dépend de `TypeUtils`. Et même si ce problème de dépendances était résolu, vu que la relation de sous-typage n'est pas totale, il se peut que les types ne soient pas comparables.

Un algorithme semblable, `least_upper_bound_of_dependent_function`, est implémenté pour le cas où nous cherchons le plus petit type fonction pour une variable donnée. Le retour de l'algorithme est différent pour le cas des fonctions et des déclarations de type et de champ. Le même argument est réalisé pour la forme $y.t'$ et l'intersection.

3. Plusieurs tentatives de démonstrations ont échoué, et la possibilité que l'assertion soit fausse a été envisagée. Cependant, par manque de temps, nous acceptons l'algorithme actuel.

VII.5 Algorithme de typage

L'algorithme de typage se trouve dans le fichier `typer.ml` du dossier `typing`. L'implémentation est relativement fidèle aux règles de typage. La fonction principale est `typ_of_internal` qui prend en paramètre un contexte et un terme.

Cette fonction contient essentiellement un pattern matching sur la structure des termes et applique la règle de typage appropriée selon la forme. Le problème d'échappement est traité dans le cas des règles (LET) et (ALL-I) à travers la fonction `check_avoidance_problem` du module `CheckUtils`. Les fonctions `least_upper_bound_of_dependent_function` et `least_upper_bound_of_recursive_type`, expliquées précédemment, sont utilisées respectivement dans les règles (ALL-E) et (FLD-E). La règle (SUB) est utilisée dans (ALL-E) pour vérifier que l'argument est un sous-type que la fonction attend.

Typage des termes récursifs

Dans la syntaxe des termes de DOT, un terme récursif est toujours accompagné de son type. Lorsque nous écrivons des programmes, cela implique que lorsque nous définissons des modules, nous devons donner leur signature. Du coté développeur, cela n'est pas très pratique.

Pour éviter cela, un algorithme est implémenté pour typer une liste de declarations. Lorsqu'un type récursif est rencontré, par exemple $\mu(x : T)d$, nous commençons par ajouter dans le contexte x avec le type `Top`. Ensuite, nous allons parcourir la déclaration d et affiner le type de x en fonction de la forme de d.

- Si d est une déclaration de type ou de champ, nous affirmons que le type de x est son type actuel intersecté avec la déclaration.
- Si d est une intersection de déclaration, nous effectuons un premier appel sur le membre de gauche de l'intersection et ensuite de droite.

L'algorithme n'est pas parfait : il échouera par exemple si des champs sont mutuellement récursifs.

Cependant, celui-ci est satisfaisant pour une partie des cas et permet de faciliter l'écrire de programmes DOT.

VII.6 Algorithme de sous-typage

L'algorithme de sous-typage se trouve dans le fichier `subtype.ml` du dossier `typing` et travaille essentiellement sur la structure des deux types donnés grâce à un pattern matching. Le but de cet algorithme est, étant donnés deux types S et T, retourner oui si S est un sous-type de T et non sinon.

L'ordre dans le pattern matching est très important contrairement à l'algorithme de typage. En effet, pour arriver à une conclusion, il est possible d'y arriver par plusieurs chemins. En effet, pour répondre à $S_1 \wedge S_2 <: T_1 \wedge T_2$, nous pouvons utiliser ($<:$ AND), (AND-1-$<:$) ou (AND-2-$<:$).

De plus l'ordre influence la réponse. Par exemple, prenons le cas où $S_1 = T_1 = \{t : Nothing..Any\}$ et $T_1 = T_2 = \{a : Nothing\}$. La question $S_1 \wedge S_2 <: T_1 \wedge T_2$ est alors vraie par réflexivité ou encore en utilisant ($<:$-AND) et ensuite (AND-1-$<:$) sur le membre de gauche et (AND-2-$<:$) sur le membre de

droite. Cependant, en utilisant (AND-1-$<:$) (ou (AND-2-$<:$)) en premier, nous obtenons une réponse fausse.

L'algorithme actuel a été implémenté de façon pragmatique, l'implémentation des règles théoriques n'étant pas simple comme peuvent le montrer [3] et [6].

Voici, dans l'ordre, comment l'algorithme gère les différents cas.

$S <: Top$ **ou** $Bottom <: T$

Nous appliquons (TOP) ou (BOTTOM) en fonction du cas. Ces règles peuvent s'appliquer directement dans l'arbre de dérivation donc nous pouvons les placer en premiers dans le pattern matching.

$\{a : L...U\} <: \{a : L'...U'\}$

Même cas que précedemment. Utilisation de la règle (TYP $<:$ TYP).

$\forall(x : S_1)T_1 <: \forall(x : S_2)T_2$

Même cas que précedemment. Utilisation de la règle (ALL $<:$ ALL).

$x.A <: y.A$

Les premiers cas posant difficultés sont ceux des types dépendants, par exemple $x.A$ et $y.A$. En effet, les règles (REFL), (SEL $<:$) et ($<:$ SEL) peuvent être employées.

L'algorithme procède par cas :

- Dans le cas de la réflexivité, cela signifie que x et y sont les mêmes atômes. Nous utilisons donc les fonctions fournies par `AlphaLib` pour le vérifier. Nous appelons cette règle (UN-REFL-TYP).
- Sinon, nous testons en premier (SEL $<:$). Si nous avons réussi à démontrer en utilisant (SEL $<:$), nous renvoyons cette solution. Sinon, nous testons avec ($<:$ SEL). Si cette dernière échoue, cela signifie qu'il n'existe pas de démonstration pour la question $x.A <: y.A$: nous renvoyons donc non.

Dans le second cas, `best_bound_of_recursive_type` est utilisé pour trouver le type x.

Types récursifs

Il est important de remarquer qu'il n'y a pas de règle de sous-typage pour les types récursifs. En effet, pour comparer deux types récursifs, ou au moins un type récursif, il est nécessaire de d'utiliser (VAR-UNPACK) ou (VAR-PACK).

Du coté de l'implémentation, nous ajoutons, dans l'ordre, les deux cas particuliers suivants.

- Si nous avons deux types récursifs $\nu(x_1 : S')$ et $\nu(x_2 : T')$, nous créons un nouvel atôme x et renommons les variables internes x_1 et x_2 par celui-ci dans S' et T'. Un appel récursif est alors effectué avec S' et T' après avoir étendu le contexte avec $x : S'$. Nous appelons cette règle (UN-REC).

— Si S est de la forme $\nu(x : S')$ et T quelconque (resp. T de la forme $\nu(x : T')$ et S quelconque), nous ajoutons x dans le contexte avec le type S' (resp T') et nous effectuons un appel récursif avec S' et T (resp. S et T'). Etendre le contexte est nécessaire si S' contient des champs mutuellement dépendants. Nous appelons ces règles respectivement (UN-$<:$ REC) et (UN-REC $<:$).

Le premier cas est nécessaire pour pouvoir comparer des types récursifs qui ne se différencient que par leur variable interne, voir VII.6.

```
sig
  type t
  val f : self.t
end
<:
sig(self')
  type t
  val f : self'.t
end
```

Code OCaml 6: Ces deux signatures sont identiques à l'exception de la variable interne. Si nous ne donnons pas le même nom à la variable interne, la question $self.t <: self'.t$ va être posée. Comme ce ne sont pas les mêmes atômes, la question $Top <: Bottom$ sera posée que nous utilisions (SEL $<:$) ou ($<:$ SEL).

$x.T <: T'$ ou $T' <: x.T$

Nous utilisons la version de `best_bound_of_recursive_type` correspondante au cas et nous appelons récursivement l'algorithme.

Intersections

L'ordre est également important pour les intersections et surtout en présence de types récursifs dans l'un des membres.

Premièrement, il est nécessaire de placer ($<:$AND) avant (AND-1-$<:$) et (AND-2-$<:$) pour pouvoir gérer le cas de la réflexivité comme nous l'avons vu ci-dessus.

Ensuite, comme pour (SEL $<:$) et ($<:$ SEL), nous ne pouvons uniquement tester (AND-1-$<:$) ou (AND-2-$<:$) car les règles peuvent être utilisées en même temps. Nous procédons donc de la même manière que pour (SEL $<:$) et ($<:$ SEL).

De plus, dans le cas ($<:$ AND), nous devons gérer les types récursifs à cause de la règle (VAR-PACK). En effet, si nous avons la question $\mu(x : S_1) \wedge \mu(x : S_2) <: \mu(x : S_1 \wedge S_2)$, nous pouvons utiliser successivement (VAR-UNPACK) à droite, puis ($<:$-AND) et (VAR-PACK) pour conclure avec (AND-1-$<:$) et (AND-2-$<:$).

De manière générale, si nous avons $\mu(x : S_1) \wedge \mu(x : S_2) <: T$, une solution est de montrer $\mu(x : S_1 \wedge S_2) <: T$. En effet, en utilisant successivement ($<:$ AND), (VAR-PACK), (VAR-UNPACK), (AND-1-$<:$) et (AND-2-$<:$), nous montrons que $\mu(x : S_1 \wedge S_2) <: T$ implique $\mu(x : S_1) \wedge \mu(x : S_2) <: T$.

La méthode utilisée dans l'algorithme consiste à renommer la variable interne de S_1 et S_2 en utilisant un même atôme unique et d'appeler récursivement l'algorithme.

Une méthode similaire est utilisée pour les cas $\mu(x : S_1) \wedge S_2 <: T$ et $S_1 \wedge \mu(x : S_2) <: T$.

Réflexivité

La régle de réflexivité (REFL) n'est pas implémentée directement. En effet, cette règle peut être dérivée d'autres règles.

- Si S est de la forme $x.A$, (UN-REFL-TYP) est utilisée.
- Si S est de la forme $\forall(x : S')T'$, nous pouvons utiliser (ALL $<:$ ALL).
- Si S est de la forme $\nu(x : S')$, (UN-REC) est utilisée.
- Si S est une intersection, nous pouvons utiliser ($<:$ AND), puis (AND-1-$<:$) sur le membre de gauche et (AND-2-$<:$) sur le membre de droite.
- Si S est de la forme $\{t : L..U\}$, (TYP $<:$ TYP) est utilisée.
- Si S est de la forme $\{a : T\}$, (FLD $<:$ FLD) est utilisée.

Transitivité

Actuellement, la transtivité n'est pas gérée. En effet, pour le gérer, il faudrait trouver un type U tel que $S <: U$ et $T <: U$, ce qui n'est pas possible, ou au moins compliqué.

VII.7 Arbre de dérivation

Les algorithmes de typages et de sous-typages nécessite un paramètre supplémentaire `history` qui permet de se souvenir des règles utilisées et de pouvoir ainsi reconstruire l'arbre de dérvation. Cet arbre de dérivation peut être affiché pour une expression en utilisant l'annotation `[@show_derivation_tree]` à la fin des expressions. Par défaut, l'arbre affiche également le contexte. Comme celui-ci peut être grand pour les longs programmes, l'annotation `[@show_derivation_tree, no_context]` affiche l'arbre de dérivation sans le contexte.

VII.8 Sucres syntaxiques

La syntaxe de base de DOT n'est pas très élégante et très pratique à utiliser. En effet, il n'est par exemple pas possible de définir des fonctions à plusieurs variables, de passer des termes en paramètre d'une fonction ou encore d'utiliser un terme dans une sélection.

Pour ces raisons, des sucres syntaxiques au niveau des parseurs sont implémentés dans le langage de surface de RML.

Currification des fonctions

Dans RML, il est possible de définir des fonctions à plusieurs variables en les séparant par une virgule. Par exemple, `fun(x : Int.t, y : Int.t) -> t`

est équivalent à `fun(x : Int.t) -> fun(y : Int.t)`. Le parseur se charge de créer l'arbre de syntaxe correspondant.

Variable interne par défaut

DOT nécessite une variable interne lors de la définition d'un module afin de pouvoir faire référence aux champs et types. Une variable par défaut, `self`, est utilisée dans le parseur afin d'alléger l'écriture de programme si le nom de la variable interne n'est pas importante.

Enregistrement

Les enregistrements ont la même représentation interne que les modules, `TermRecursiveRecord`. Dans le langage de surface, les enregistrements ainsi que le type enregistrement sont définis de la même manière qu'en OCaml. Afin d'éviter des références internes entre champs, la variable interne utilisée est `'self`. Le nom des variables commençant par des simples guillemets n'étant pas acceptés dans le lexeur, cela implique qu'il n'est pas possible d'avoir des champs mutuellement dépendants.

Termes comme paramètres et fonctions

Une fonctionnalité intéressante et pratique est l'utilisation de termes pour les paramètres ainsi que pour les fonctions. Ceci est géré dans le parseur et ce dernier génère des bindings locaux.

Un point important [4] est qu'il faut éviter de réaliser un binding local d'une variable car cela pourrait provoquer des problèmes d'échappement.

```
let module M = struct
  type t = Int.t
  let x : self.t = 42
end;;

let f = fun(m : sig type t val x : self.t end) -> m.x;;

f M;;
```

Code OCaml 7: Exemple où un binding local d'une variable ne doit pas être généré afin de ne pas provoquer un problème d'échappement. Si des bindings locaux sont réalisées pour chaque variable, le code généré pour fM est $letn = Minfn$ dont le type est $n.t$.

Applications de fonctions à plusieurs paramètres

Une autre fonctionnalité importante est la possibilité d'appliquer plusieurs paramètres à une fonction. C'est aussi le parseur qui s'en occupe en générant des bindings locaux. Voici quelques exemples :

— $f\ x\ y$ est interprété comme $let\ f_x\ =\ (f\ x)\ in\ (f_x\ y)$.

4. Et qui n'est pas mentionné dans les documents sur DOT.

— $f\ x\ y\ z$ est interprété comme $let\ f_x\ =\ (f\ x)\ in\ let\ f_y\ =\ (f_x\ y)\ in\ (f_y\ z)$.

VII.9 Termes ajoutés

Termes unit et entiers

Des types basiques comme $Int.t$ et $Unit.t$ pour les entiers et le terme `unit` sont implémentés. Il est possible d'utiliser des entiers comme en OCaml ou le terme `()` pour le terme `unit`.

Unimplemented

L'implémentation ne se focalisant pas sur l'évaluation, le sémantique des termes peut être laissée de coté. Pour cela, un terme `Unimplemented` est présent et de type `Bottom`.

TermAscription

Comme dans la plupart des langages, RML autorise l'ascription de termes, c'est-à-dire forcer le type d'un terme. En particulier, cela permet de donner le type voulu au terme `Unimplemented`. La syntaxe est $t : T$.

TermRecursiveRecordUntyped

Un terme est ajouté dans la grammaire pour les modules définis sans type, l'algorithme de typage sur les modules décrit précédemment étant utilisé pour typer le terme ;

VII.10 Exemples

TODO

VII.11 Travail futur

Bien que l'implémentation actuelle donne des résultats satisfaisants sur différents cas, diverses améliorations peuvent être effectuées. Une liste non-exhaustive peut être trouvée, en anglais, sur la page du projet[18]. Voici quelques exemples.

— L'algorithme de sous-typage provoque sur certains cas des stack overflow. Ceci n'est pas très surprenant. En effet, à cause du type récursif, un arbre de dérivation n'est pas nécessairement de taille finie car il est possible que l'algorithme doive répondre à la même question dans un sous-arbre.
— Nous nous sommes focalisés essentiellement sur l'implémentation des algorithmes de typage et de sous-typage, et non sur l'évaluation des termes. Un évaluateur pourrait être implémenté en se basant sur [15].
— Un interpréteur intéractif.

— Améliorer l'algorithme d'inférence de type pour les modules. En effet, comme nous l'avons vu, celui-ci est relativement naif et ne permet pas par exemple de gérer un module contenant des champs mutuellement dépendants.
— Améliorer et prouver ensuite que les algorithmes de sous-typage et de typage définissent bien les relations de typages et de sous-typages.
— Pour l'instant, il est nécessaire de donner un type lorsque nous utilisons un match sur une option (voir fichier `stdlib/option_church.rml`), ce qui n'est pas courant en OCaml car le type est inféré. Cette inférence de type passe par la résolution d'équations et nécessite de travailler avec deux arbres différents.

Conclusion

Dans les premiers chapitres, nous avons défini les bases théoriques de la programmation fonctionnelle et des langages typés. Nous sommes partis de différents calculs relativement simples comme le λ-calcul non typé et le λ-calcul simplement typé pour arriver à un calcul plus compliqué et plus récent (2016) appelé DOT qui unifie le comportement des enregistrements et des modules et permet en même temps de considérer les modules comme des citoyens de première classe. Nous avons également montré comment DOT pouvait être interprété comme une extension de System $F_{<:}$.

Dans le dernier chapitre, nous avons présenté une implémentation en OCaml basée sur DOT. Nous avons pu remarquer qu'implémenter un langage à partir de règles théoriques n'est pas évident et cela à cause des différents arbres de dérivations possibles pour une même question ou encore en raison des arbres de tailles infinis. Nous avons également remarqué qu'il était nécessaire de changer certaines règles d'inférence pour écrire un algorithme, comme pour le cas de la réflexivité, l'inclusion de T-SUB dans les règles d'inférence ou encore l'introduction de UN-REC pour pouvoir comparer des types récursifs. De plus, nous avons remarqué qu'introduire des types chemins dépendents dans le langage ne facilite pas l'implémentation.

Nous avons également vu qu'écrire des programmes dans un calcul théorique comme DOT n'est pas très pratique et implique de développer un langage de surface. Dans ce langage de surface, des sucres syntaxiques sont implémentés afin de pouvoir écrire des termes interdits dans DOT, comme l'application de termes à une fonction. Cependant, à travers l'implémentation de ces sucres, nous avons remarqué qu'il manquait certaines règles pour pouvoir écrire certains programmes usuels comme le binding local d'une variable.

DOT n'est pas le seul calcul dans lequel les modules peuvent être considérés comme citoyens de première classe. D'autres calculs ont été explorés comme 1ML[1]. Ce dernier catégorise les types en genres[9] afin d'affiner les règles de typage et de sous-typage sur les types. DOT a été choisi à la place de 1ML car ce dernier possède déjà une implémentation et les règles de typage et de sous-typage sont plus élaborées que DOT.

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 hauteur de l'arbre de typage (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.

Bibliographie

[1] Andreas, R. 1ML. https://people.mpi-sws.org/~rossberg/1ml/, 2015-2016.

[2] Goubault-Larrecq, J. Cours intitulé λ-calcul et langages fonctionnels. http://www.lsv.fr/~goubault/Lambda/lambda.pdf.

[3] Pierce, B. C. *TAPL - Chapter 16 - Metatheory of subtyping.* The MIT Press, 2002.

[4] Pierce, B. C. *TAPL - Chapter 23 - System F.* The MIT Press, 2002.

[5] Pierce, B. C. *TAPL - Chapter 26 - Bounded quantification.* The MIT Press, 2002.

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

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

[8] Pierce, B. C. *TAPL - Chapter 9 - The Simply Typed Lambda-Calculus.* The MIT Press, 2002.

[9] Pierce, B. C. *TAPL - Partie 4 - Higher Order Systems.* The MIT Press, 2002.

[10] Pottier, F. AlphaLib. https://gitlab.inria.fr/fpottier/alphaLib.

[11] Pottier, F. Menhir. http://gallium.inria.fr/~fpottier/menhir/.

[12] Pottier, F. PPrint. http://opam.ocaml.org/packages/pprint/.

[13] Pottier, F. Visitors. https://gitlab.inria.fr/fpottier/visitors.

[14] Pottier, F. Visitors Unchained. http://gallium.inria.fr/~fpottier/publis/fpottier-visitors-unchained.pdf, 2017.

[15] Rompf, T., Amin, N., Grütter, S., Odersky, M., and Stucki, S. The Essence of Dependent Object Types. *WF* (2016).

[16] Troestler, C. ANSITerminal. https://github.com/Chris00/ANSITerminal.

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

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