

# 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-directeur:** Christophe Troestler

**Service:** Equipe Gallium (INRIA Paris)



# 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<sup>1</sup>, 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 Eduardo Dom, Guillaume Huysmans et Aline Goeminne pour leurs divers commentaires et diverses corrections.

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

---

1. Malheureusement pas abordées dans ce sujet.



# Table des matières

<b>Introduction</b>	<b>5</b>
<b>I <math>\lambda</math>-calcul non typé</b>	<b>9</b>
I.1 Syntaxe	9
I.2 Sémantique	12
I.2.1 Stratégies de réduction	13
I.3 Codage de termes usuels	15
<b>II <math>\lambda</math>-calcul simplement typé</b>	<b>17</b>
II.1 Typage, contexte de typage et règle d'inférence	17
II.2 Sûreté du typage	19
<b>III <math>\lambda</math>-calcul avec sous-typage et enregistrements.</b>	<b>25</b>
III.1 $\lambda$ -calcul simplement typé avec enregistrements	25
III.2 Sous-typage	27
III.3 Sûreté	29
III.4 Type Top et type Bottom	32
<b>IV Système F</b>	<b>33</b>
IV.1 Syntaxe	33
IV.2 Sémantique	35
IV.3 Contexte de typage	35
IV.4 Règles de typage	35
IV.5 Sûreté	36
<b>V Système <math>F_{&lt;}</math></b>	<b>39</b>
V.1 Syntaxe	40
V.2 Sémantique	40
V.3 Contexte de typage	40
V.4 Règles de typage et de sous-typage	41
V.5 Sûreté	42
V.6 Indécidabilité du sous-typage	42
<b>VI DOT</b>	<b>43</b>
VI.1 Syntaxe	43
VI.2 Sémantique	45
VI.3 Règles de typage	45
VI.4 Règles de sous-typage	47
VI.5 Problème de mauvaises bornes	48

VI.6 Encodage de Système $F_{<}$	48
VI.7 Sûreté	49
<b>VII RML</b>	<b>51</b>
VII.1 Langage de surface	51
VII.2 Implémentation des ASTs	52
VII.3 Contexte de typage	53
VII.4 Meilleures bornes d'un type dépendant	54
VII.5 Algorithme de typage	56
VII.6 Algorithme de sous-typage	57
VII.7 Arbre de dérivation	61
VII.8 Sucres syntaxiques	61
VII.9 Termes ajoutés	63
VII.10 Exemples	63
VII.11 Travail futur	65
<b>Conclusion</b>	<b>69</b>

# 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 modules, 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 « primitifs », comme les entiers, les chaînes de caractères ou les fonctions, et le langage de modules. En particulier, le terme *foncteur* est employé à la place de *fonction* pour parler des fonctions prenant un module en paramètre et en renvoyant un autre.

```
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 représenter 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 base d'OCaml se trouvent les *enregistrements*. Ces derniers peuvent être interprétés comme des listes de couples (*label*, *valeur*), et ressemblent aux modules. Cependant, les deux différences majeures sont la possibilité de définir des types dans un module ainsi que d'avoir des champs mutuellement dépendants.

Ce mémoire vise à donner, dans un premier temps et après avoir défini les

```

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 polymorphe notre module Point2D.

notions nécessaires, un calcul typé, DOT[18], 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 propose une implémentation en OCaml des algorithmes de typage et de sous-typage ainsi qu'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  $\lambda$ -calcul.

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

Dans le chapitre 2, nous introduirons la notion de type et nous l'appliquons au  $\lambda$ -calcul, ce qui nous donnera *le  $\lambda$ -calcul simplement typé*. Nous discuterons de la notion de *sûreté 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  $\lambda$ -calcul simplement typé avec la notion de polymorphisme qui permet d'attribuer plusieurs types à un terme. Le chapitre 3 se concentre sur *le polymorphisme avec sous-typage*, illustré avec les enregistrements. Dans le chapitre 4, nous parlerons de *polymorphisme paramétré* qui, combiné au  $\lambda$ -calcul simplement typé, forme le calcul appelé *Système F*. Le chapitre 5 se chargera de combiner ces deux notions de polymorphismes dans un calcul appelé *Système F<sub><</sub>*. Une preuve des théorèmes de préservation et de progression sera donnée 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 offrent 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  $\lambda$ -calcul simplement typé, les enregistrements, le polymorphisme par sous-typage et le polymorphisme paramétré ainsi que des types récurifs.

Pour finir, dans le chapitre 7, nous discuterons de l'implémentation du langage RML[19] 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-typage à un algorithme n'est pas évident pour plusieurs raisons.

La principale difficulté de ce travail se trouve dans l'étude des types chemin dépendants, sujet de recherche récent et moins bien compris que les calculs comme *Système F* ou *Système F*<sub><</sub>, ainsi que la gestion de ceux-ci dans les algorithmes.



# Chapitre I

## $\lambda$ -calcul non typé

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

### I.1 Syntaxe

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

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

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

La plupart des ensembles que nous définirons seront définis de manière inductive 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
$tt$	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  $\lambda$ -termes sont :

- la fonction identité :  $\lambda x. x$
- la fonction constante en  $y$  :  $\lambda x. y$
- la fonction qui renvoie la fonction constante pour n'importe quelle variable :  $\lambda y. \lambda x. y$ .
- 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  $\lambda$ -terme. Par exemple, dans le  $\lambda$ -terme  $\lambda x. x$  la variable  $x$  est liée par un  $\lambda$ <sup>1</sup> 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  $\Lambda$  par :*

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

**Définition I.3** (Ensemble de variables liées). *L'ensemble des variables **liées** d'un terme  $t$ , noté  $BV(t)$  est défini récursivement sur la structure des termes de  $\Lambda$  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 dont nous voudrions naturellement qu'ils soient les mêmes. Par exemple, nous voudrions que la fonction identité  $\lambda x. x$  ne dépende pas de la variable liée  $x$ , c'est-à-dire que les termes  $\lambda x. x$  et  $\lambda y. y$  soient un seul et unique terme : la fonction identité. Cette égalité se résume à une substitution de la variable  $x$  par la variable  $y$ , ou plus généralement par un terme  $u$ .

Avant de donner une définition exacte, il est important de remarquer que la substitution n'est pas une action triviale si nous ne voulons pas changer le sens des termes. Si nous effectuons une substitution purement syntaxique, nous pouvons alors obtenir des termes qui ne sont plus dans la syntaxe des éléments de  $\Lambda$ . Par exemple, si nous substituons toutes les occurrences 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  $\lambda$ .

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 substitution 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 substitution, 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 substitution). *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  $[x := v]u$ , par :*

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

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

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

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

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

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

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

Nous notons  $=_\alpha$  la plus petite relation comprenant  $\alpha$  telle que :

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

**Exemple.** 1. Il est clair que  $\lambda x.x =_\alpha \lambda y.y$  par définition de la relation  $\alpha$ .

2. De même,  $\lambda x.\lambda y.xy =_\alpha \lambda y.\lambda x.yx$ . En effet, on montre que  $\lambda x.\lambda y.xy =_\alpha \lambda z.\lambda w.zw$  et  $\lambda y.\lambda x.yx =_\alpha \lambda z.\lambda w.zw$  en appliquant deux fois la substitution (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  $\Lambda \setminus =_\alpha$ . Dans ce quotient, les termes égaux à renommage

de variables près se retrouvent dans la même classe d'équivalence. À partir de maintenant, nous considérons  $\Lambda \setminus =_\alpha$ , c'est-à-dire que nous parlons des termes à  $\alpha$ -renommage près.

## I.2 Sémantique

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

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

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

Voici quelques exemples de réductions :

**Exemple.** 1.  $(\lambda x.x)y \rightarrow y$ .

2.  $(\lambda y.(\lambda x.yx))z \rightarrow \lambda x.zx$ .

3.  $(\lambda w.(\lambda x.x)v)z \rightarrow (\lambda w.v)z$  (on réduit à l'intérieur, c'est-à-dire  $(\lambda x.x)v$ ).

4.  $(\lambda w.(\lambda x.x)v)z \rightarrow (\lambda x.x)v$  (on réduit à l'extérieur, c'est-à-dire  $(\lambda w.t)z$  où  $t = (\lambda x.x)v$ ).

5.  $(\lambda w.(\lambda x.x)v)z \rightarrow_\beta^* v$

6.  $(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx)$ .

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

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

Un terme  $t$  qui peut être réduit, c'est-à-dire qu'il existe  $u$  tel que  $t \rightarrow u$ , 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. N'oublions pas que nous travaillons à  $\alpha$ -renommage près.

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

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

### I.2.1 Stratégies de réduction

Nous présentons les stratégies les plus utilisées. Le terme  $id(id(\lambda z.idz))$

où  $id = \lambda x.x$  sera utilisé pour interpréter chaque stratégie de réduction. Les redexes de ce terme sont :

- $id(id(\lambda z.idz))$
- $id(\lambda z.idz)$ .
- $idz$

Dans les exemples ci-dessous, le redex qui est réduit est souligné.

#### Ordre normal

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

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

#### Call-by-name

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

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

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

### Call-by-value

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

```
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 :

$$\begin{aligned} id(id(\lambda z.(id\ z))) &\rightarrow_{\beta} \\ id(\lambda z.(id\ z)) &\rightarrow_{\beta} \\ \lambda z.(id\ z) \end{aligned}$$

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

$$\begin{array}{ccc} \frac{t_1 \rightarrow t'_1}{t_1 t \rightarrow t'_1 t} & \text{(E-APP1)} & \frac{t \rightarrow t'}{v t \rightarrow v t'} \quad \text{(E-APP2)} \\ & & (\lambda x.t)v \rightarrow [x := v]t \quad \text{(E-APPABS)} \end{array}$$

La notation  $\frac{t \rightarrow t'}{v t \rightarrow v t'}$  est l'équivalent d'une implication où la prémisse (ici  $t \rightarrow t'$ ) se trouve au dessus et la conclusion en dessous (ici  $v t \rightarrow v t'$ ). 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  $\beta$ ).

La relation de  $\beta$ -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  $\beta$ -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 [4]. Dans ce cours sont traités les sujets de normalisation (forme normale, finitude de la  $\beta$ -réduction), de confluence (est-ce que tout terme se réduit en une unique forme normale) et d'une différente sémantique appelée *sémantique dénotationnelle*.

### I.3 Codage de termes usuels

Le  $\lambda$ -calcul est assez riche pour définir des termes usuels des langages de programmation comme les booléens (et en même temps les conditions), les paires ou encore les entiers. Ces codages peuvent être trouvés dans [9]. Voici l'exemple des booléens, utilisé dans RML ([19]) :

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

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  $\lambda$ -calcul :

- $and = \lambda b. \lambda b'. b\ b'\ false$
- $or = \lambda b. \lambda b'. b\ true\ b'$



## Chapitre II

# $\lambda$ -calcul simplement typé

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

### 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  $\Lambda$  un ensemble de termes. Soit  $\tau$  un ensemble, appelé **ensemble des types**, dont les éléments sont notés  $T$ .*

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

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

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

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

---

1. Plus d'informations peuvent être trouvées dans [10]

**Définition II.2.** Soit  $B$  un ensemble de types appelés **types de base**. L'ensemble des **types simples** est défini par la grammaire suivante :

$$\begin{array}{ll} T ::= & \text{types} \\ & B \quad \text{base} \\ & T \rightarrow T \quad \text{type des fonctions} \end{array}$$

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

## Contexte et jugement de typage

Nous avons déjà mentionné que, naturellement, les abstractions  $\lambda x. t$  ont le type flèche, par exemple  $T_1 \rightarrow T_2$ . 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 annoter la variable avec un type, soit étudier le type du 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 :

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

Avant de discuter des règles de typage, il convient de remarquer qu'il est nécessaire de connaître 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  $\Gamma$ , est une suite finie de couples  $(x_i, T_i)$  où  $x_i$  est une variable et  $T_i$  est un type. Chaque  $x_i$  est différent.

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

Le domaine de  $\Gamma$ , noté  $\text{dom}(\Gamma)$ , 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  $(\Gamma, t, T)$  où  $\Gamma$  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  $\Gamma$ <sup>2</sup> ». Si  $\Gamma$  est vide, nous omettons  $\emptyset$  et le jugement devient  $\vdash t : T$ .

---

2. ou encore dans le contexte  $\Gamma$

## Règle de typage et arbre de dérivation

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

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

$$\begin{array}{c} \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR}) \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma, x : T_1 \vdash \lambda x : T_1. t : T_1 \rightarrow T_2} \quad (\text{T-ABS}) \\[10pt] \frac{\Gamma \vdash u : T_1 \rightarrow T_2 \quad \Gamma \vdash v : T_2}{\Gamma \vdash uv : T_2} \quad (\text{T-APP}) \end{array}$$

La règle (T-VAR) est évidente : si  $(x, T)$  est dans le contexte, alors  $x$  est de type  $T$  sous le contexte  $\Gamma$ . Quant à (T-ABS), elle affirme que si le corps  $t$  de l'abstraction  $\lambda x : T_1. t$  est de type  $T_2$ , alors l'abstraction est de type  $T_1 \rightarrow T_2$ . Pour finir, (T-APP) type les applications : dans le terme  $uv$ ,  $u$  doit être une fonction de type  $T_1 \rightarrow T_2$ , et  $v$  doit être du même type que celui 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 typage, construits à partir des règles de typage 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 permettent de déduire le type.

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

$$\begin{array}{c} \frac{\frac{(\text{T-VAR}) \quad \frac{x : T_1 \rightarrow T_2 \in \Gamma}{\Gamma \vdash x : T_1 \rightarrow T_2} \quad \frac{y : T_1 \in \Gamma}{\Gamma \vdash y : T_1} (\text{T-VAR})}{\Gamma, x : T_1 \rightarrow T_2, y : T_1 \vdash xy : T_2} (\text{T-APP})}{\Gamma, x : T_1 \rightarrow T_2 \vdash \lambda y : T_1. xy : T_1 \rightarrow T_2} (\text{T-ABS}) \\[10pt] \frac{\Gamma, x : T_1 \rightarrow T_2 \vdash \lambda y : T_1. xy : T_1 \rightarrow T_2}{\Gamma \vdash \lambda x : T_1 \rightarrow T_2. \lambda y : T_1. xy : (T_1 \rightarrow T_2) \rightarrow T_1 \rightarrow T_2} (\text{T-ABS}) \end{array}$$

## II.2 Sûreté du typage

Dans cette partie, nous allons aborder deux théorèmes importants : les théorèmes de progression et de préservation du typage. En assemblant ces deux théorèmes, nous en déduisons le principe « les programmes<sup>3</sup> 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 se réduira en une valeur du type du programme.

Ces deux théorèmes relient les deux relations précédemment définies : la relation de typage et la relation de  $\beta$ -réduction.

1. Progression : si un terme est bien typé, alors soit c'est une valeur, soit il se réduit en un terme.

---

3. Un programme est synonyme de terme.

2. Préservation (du typage) : si un terme  $t$  de type  $T$  se réduit 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 typage.

**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  $T = T_1 \rightarrow T_2$  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  $t_1 : T_1 \rightarrow T$  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).  $\square$

Le lemme d'inversion des règles de typage dit également quelque chose de fondamental sur les arbres de typage et qui a une énorme importance lorsque nous souhaitons implémenter un algorithme de typage<sup>4</sup>. En effet, les 3 points du lemme nous donnent quels sont les possibles fils de la conclusion. Par exemple, si nous devons montrer que  $\Gamma \vdash \lambda x : T. t : T'$ , 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 sont 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  $\lambda$ -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,  $S = T_1 \rightarrow R_1$  et  $T = T_1 \rightarrow R_2$  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  $T_1$  et  $T_2$  tel que  $u$  est de type  $T_1 \rightarrow S$  et de type  $T_2 \rightarrow T$  avec  $v$  de type  $T_1$  et de type  $T_2$ . Par induction sur  $u$  et sur  $v$ , le type de  $v$  est unique ( $T_1 = T_2$ ) et celui de  $u$  également. Nous avons donc  $S = T$ .

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

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

*Démonstration.* 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  $uv$ .  $t$  étant bien typé, nous avons le jugement de typage  $\vdash uv : T$ . Par le lemme d'inversion,  $u : T_1 \rightarrow T$  et  $v : T_1$ . Par induction, comme  $u$  (resp.  $v$ ) est bien typé,  $u$  (resp.  $v$ ) est soit une valeur, soit se réduit en un  $u'$  (resp.  $v'$ ).
  - Si  $u$  se réduit en  $u'$ , alors (E-APP1) s'applique et  $uv$  se réduit en  $u'v$ .
  - Si  $u$  est une valeur et  $v$  se réduit en  $v'$ , alors (E-APP2) s'applique et  $uv$  se réduit en  $uv'$ .
  - Si  $u$  et  $v$  sont des valeurs, (E-APPABS) s'applique.

□

## Préservation

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

*Démonstration.* Par induction sur l'arbre de dérivation de typage.

- (T-VAR).  $t$  est une variable  $x$ . Par hypothèse,  $\Gamma \vdash x : T$  et  $(x, T) \in \Gamma$ . D'où  $(x, T) \in \Delta$ . Par (T-VAR),  $\Delta \vdash x : T$ .
- (T-ABS). Nous avons  $t = \lambda x : T_1. t'$  et

$$(T-ABS) \frac{\Gamma, x : T_1 \vdash t' : T_2}{\Gamma \vdash \lambda x : T_1. t' : T_1 \rightarrow T_2}$$

Par hypothèse de récurrence,  $\Delta, x : T_1 \vdash t' : T_2$ . Par (T-ABS),  $\Delta \vdash \lambda x : T_1. t' : T_1 \rightarrow T_2$ .

- (T-APP). Nous avons donc  $t = uv$  et

$$(T-APP) \frac{\Gamma \vdash u : T_1 \rightarrow T \quad \Gamma \vdash v : T_1}{\Gamma \vdash uv : T}$$

Par hypothèse de récurrence,  $\Delta \vdash u : T_1 \rightarrow T$  et  $\Delta \vdash v : T_1$ . Par (T-APP),  $\Delta \vdash uv : T$ .

□

**Lemme II.10** (d'affaiblissement). *Soit  $\Gamma \vdash t : T$  et  $x \notin \text{dom}(\Gamma)$ .*

*Alors  $\Gamma, x : S \vdash t : T$ .*<sup>5</sup>

*Démonstration.* Par induction sur l'arbre de dérivation de typage.

- (T-VAR).  $t = y$ . Le cas où  $y = x$  est impossible car nous avons  $(x, T) \in \Gamma$  et cela contredit l'hypothèse que  $x \notin \text{dom}(\Gamma)$ . Si  $y \neq x$ , nous avons  $(y, T) \in \Gamma$  et par conséquent,  $(y, T) \in \Gamma, x : S$  et nous concluons en utilisant (T-VAR).

---

<sup>5</sup>. Nous pouvons généraliser le lemme à un contexte  $\Gamma'$  dont le domaine est distinct de celui de  $\Gamma$ .

- (T-ABS).  $t = \lambda y : T_1. t'$ . Nous avons  $T = T_1 \rightarrow T_2$  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  $\Gamma, x : S \vdash \lambda y : T_1. t' : T_1 \rightarrow T_2$ .
- (T-APP).  $t = uv$ . Nous avons  $\Gamma \vdash u : T_1 \rightarrow T$  et  $\Gamma \vdash v : T_1$ . Par hypothèse de récurrence,  $\Gamma, x : S \vdash u : T_1 \rightarrow T$  et  $\Gamma, x : S \vdash v : T_1$ . Nous concluons que  $\Gamma, x : S \vdash uv : T$  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  $\Gamma \vdash [x \rightarrow s]t : T$*

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

- (T-VAR).  $t = z$ . Alors, par le lemme d'inversion,  $(z, T) \in \Gamma, x : S$ . Deux cas sont possibles. Si  $z = x$ , alors  $[x \rightarrow s]z = s$  ainsi que  $S = T$  et nous obtenons le résultat souhaité. Si  $z \neq x$ , alors  $[x \rightarrow s]z = z$  et il n'y a rien à montrer car  $\Gamma \vdash z : T$ .
- (T-ABS).  $t = \lambda y : T_1. t'$ . Nous avons  $T = T_1 \rightarrow R$  avec  $\Gamma, x : S, y : T_1 \vdash t' : R$ .

Sans perte de généralité, nous supposons  $y \notin \text{dom}(\Gamma)$ . Rappelons que par définition de la  $\beta$ -réduction,

$$[x \rightarrow s]\lambda y : T_1. t' = \lambda y : T_1. [x \rightarrow s]t'$$

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  $y \notin \text{dom}(\Gamma)$ , nous obtenons  $\Gamma, y : T_1 \vdash s : S$ . Nous appliquons alors l'hypothèse de récurrence et nous obtenons  $\Gamma, y : T_1 \vdash [x \rightarrow s]t' : R$ . Par (T-ABS), nous avons  $\Gamma \vdash [x \rightarrow s]\lambda y : T_1. t' : R$ .

- $t = uv$ . Rappelons que par définition de la  $\beta$ -réduction,

$$[x \rightarrow s](uv) = ([x \rightarrow s]u)([x \rightarrow s]v)$$

Nous avons  $\Gamma, x : S \vdash u : T_1 \rightarrow T$  et  $\Gamma, x : S \vdash v : T$ . Par hypothèse de récurrence, nous avons  $\Gamma \vdash [x \rightarrow s]u : T_1 \rightarrow T$  et  $\Gamma \vdash [x \rightarrow s]v : T$ . Par (T-APP), nous avons  $\Gamma \vdash [x \rightarrow s](uv) : T$ .

□

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

*Démonstration.* Par induction sur l'arbre de dérivation de  $\Gamma \vdash t : T$ .

- (T-VAR).  $t = x$ . Ce cas n'est pas possible car aucune règle de réduction existe pour les variables.
- (T-ABS).  $t = \lambda x : T_1. t_2 : T$ . Même chose que pour le cas des variables.
- (T-APP).  $t = uv$ . Nous avons  $\Gamma \vdash u : T_1 \rightarrow T$  et  $\Gamma \vdash v : T_1$ . Plusieurs cas possibles :
  - $u$  se réduit en  $u'$ . Alors,  $t' = u'v$  par (E-APP1). Par hypothèse de récurrence, nous obtenons  $\Gamma \vdash u' : T_1 \rightarrow T$ . Par (T-APP),  $\Gamma \vdash u'v : T$ .



- $v$  se réduit en  $v'$  et  $u$  est une valeur. Alors,  $t' = uv'$  par (E-APP2). Nous appliquons alors le même argument que pour le cas précédent.
- $u$  et  $v$  sont des valeurs. Posons  $u = \lambda x : T_1. t_2$ . Alors  $t' = [x \rightarrow v]t_2$  et nous avons  $\Gamma \vdash v : T_1$ . Nous avons  $\Gamma, x : T_1 \vdash t_2 : T$ . Par le lemme de substitution, nous concluons  $\Gamma \vdash [x \rightarrow v]t_2 : T$ .

□



## Chapitre III

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

La syntaxe des termes du  $\lambda$ -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  $\{l_i = t_i\}^{1 \leq i \leq n}$  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. Pour cette raison, nous ajoutons le terme  $t.l$  qui permet d'accéder au label  $l$  du terme  $t$ .

Il nous faut également typer les enregistrements. Pour cela, nous ajoutons une nouvelle syntaxe dans les types, noté  $\{l_i : T_i\}^{1 \leq i \leq n}$  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 réel, 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 $\lambda$ -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	$T \rightarrow T$	fonction
$\lambda x : T. t$	abs	$\{l_i : T_i\}^{1 \leq i \leq n}$	enreg
$\{l_i = t_i\}^{1 \leq i \leq n}$	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
$\lambda x : T. t$	abs
$\{l_i = v_i\}^{1 \leq i \leq n}$	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.

$$\begin{array}{c}
\frac{t_j \rightarrow t'_j}{\{l_1 = v_1; \dots; l_j = t_j; \dots; l_n = t_n\} \rightarrow \{l_1 = v_1; \dots; l_j = t'_j; \dots; l_n = t_n\}} \quad (\text{E-RCD}) \qquad \frac{t_1 \rightarrow t'_1}{t_1.l \rightarrow t'_1.l} \quad (\text{E-PROJ}) \\
\{l_i = v_i\}^{1 \leq i \leq n}.l_j \rightarrow v_j \quad (\text{E-PROJ-RCD})
\end{array}$$

La règle (E-RCD) nous dit comment les termes à l'intérieur d'un enregistrement sont évalués. Quant à (E-PROJ-RCD)<sup>1</sup>, 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 simplifier le terme dans une projection.

## Règles de typage

En plus des règles de typage du  $\lambda$ -calcul simplement typé, la relation de typage comprend les règles suivantes :

---

1. Nous supposons que  $j \in \{1, \dots, n\}$ .

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

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

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 typage 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 concrète, supposons que nous ayons défini un type  $\mathbb{R}$  pour l'ensemble des réels et définissons le point  $(5, 5)$  du plan  $\mathbb{R}^2$  par  $\{x = 5; y = 5\}$ <sup>2</sup>. 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 :

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

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  $T$ .

---

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

## 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 :

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

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 :

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

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

(S-RCD-WIDTH) nous permet de « laisser de côté » 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-typage entre eux. Par exemple, nous aimerions qu'un point du plan  $\mathbb{N}^2$  soit également un point du plan  $\mathbb{R}^2$ .

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

(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 ajoutons une règle pour comparer deux types flèches. Nous disons que le type flèche est *contravariant* pour les paramètres et *covariant* pour le type de retour.

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

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

$$(\text{T-APP}) \frac{\Gamma \vdash \lambda p : \{x : \mathbb{R}\} . p.x : \{x : \mathbb{R}\} \rightarrow \mathbb{R} \quad \frac{\Gamma \vdash \{x = 5; y = 5\} : \{x : \mathbb{R}; y : \mathbb{R}\} \quad \frac{\{x : \mathbb{R}; y : \mathbb{R}\} <: \{x : \mathbb{R}\}}{\Gamma \vdash \{x = 5; y = 5\} : \{x : \mathbb{R}\}} \quad (\text{T-SUB})}{\Gamma \vdash (\lambda p : \{x : \mathbb{R}\} . p.x) \{x = 5; y = 5\} : \mathbb{R}}$$

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

### III.3 Sûreté

Nous allons montrer que les théorèmes de préservation et de progression démontrés pour le  $\lambda$ -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  $S <: T_1 \rightarrow T_2$ , alors  $S$  est de la forme  $S_1 \rightarrow S_2$  avec  $T_1 <: S_1$  et  $S_2 <: T_2$ .*  
*2. Si  $S <: \{l_i : T_i\}^{1 \leq i \leq n}$ , alors  $S$  est de la forme  $\{k_i : S_i\}^{1 \leq i \leq m}$  tel que  $(k_i)_{1 \leq i \leq m} \subseteq (l_i)_{1 \leq i \leq n}$  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.

- (S-REFL). Le résultat est direct car  $S = T_1 \rightarrow T_2$ .
- (S-ARROW). Le résultat est également direct.
- (S-TRANS). Nous avons donc l'arbre de dérivation suivant :

$$\frac{S <: T \quad T <: T_1 \rightarrow T_2}{S <: T_1 \rightarrow T_2}$$

Nous appliquons d'abord l'hypothèse de récurrence sur l'affirmation  $T <: T_1 \rightarrow T_2$  et ensuite sur l'affirmation  $S <: T$ . Nous concluons en utilisant (S-TRANS).

□

Nous montrons également un lemme d'inversion des règles de typage comme il a été démontré pour le  $\lambda$ -calcul simplement typé.

**Lemme III.2** (d'inversion des règles de typage). *1. Si  $\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2$ , alors  $T_1 <: S_1$  et  $\Gamma, x : S_1 \vdash s_2 : T_2$ .*  
*2. Si  $\Gamma \vdash \{k_i = s_i\}^{1 \leq i \leq n} : \{l_i : T_i\}^{1 \leq i \leq m}$ , alors  $(l_i)_{1 \leq i \leq m} \subseteq (k_i)_{1 \leq i \leq n}$  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 :

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

En appliquant le lemme III.1 sur l'affirmation  $T <: T_1 \rightarrow T_2$ , nous obtenons  $T = \tilde{S}_1 \rightarrow \tilde{S}_2$  avec  $T_1 <: \tilde{S}_1$  et  $\tilde{S}_2 <: T_2$ . L'affirmation de gauche devient donc

$$\Gamma \vdash \lambda x : S_1. s_2 : \tilde{S}_1 \rightarrow \tilde{S}_2$$

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  $T_1 <: \tilde{S}_1$  et  $\tilde{S}_1 <: S_1$ , 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. La fonction de substitution est étendue aux projections et aux enregistrements de manière naturelle.

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

*Alors  $\Gamma \vdash [x \rightarrow s]t : T$*

*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). Pour (T-SUB), nous utilisons l'hypothèse de récurrence sur le jugement de gauche et (T-SUB) pour conclure. □

**Théorème III.4** (de préservation du typage). *Soit  $\Gamma \vdash t : T$  et soit  $t'$  tel que  $t \rightarrow t'$ . Alors,  $\Gamma \vdash t' : T$ .*

*Démonstration.* Nous procédons de la même manière que pour le cas du  $\lambda$ -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).  $t = \lambda x : T_1. t_{12}$  : déjà une valeur.
3. (T-APP).  $t = t_1 t_2$ . Nous avons  $\Gamma \vdash t_1 : T_1 \rightarrow T$ ,  $\Gamma \vdash t_2 : T_1$ . Les possibles règles d'évaluation sont (E-APP1), (E-APP2) et (E-APPABS). Pour (E-APP1) et (E-APP2), même raisonnement que pour le  $\lambda$ -calcul simplement typé.  
Quant à (E-APPABS), posons  $t_1 = \lambda x : S_1. t_{12}$  et  $t_2 = v$ . En utilisant III.3, nous déduisons  $T_1 <: S_1$  et  $\Gamma, x : S \vdash t_{12} : T$ . Nous concluons avec le lemme de substitution qui donne  $\Gamma, x : S \vdash [x \rightarrow v]t_{12} : T$ .
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  $t = t_1.l_j$ ,  $\Gamma \vdash t_1 : \{l_i : T_i\}^{1 \leq i \leq n}$  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  $t_1 = \{k_i = v_i\}^{1 \leq i \leq m}$  et par le lemme III.1, nous avons  $(l_i)_{1 \leq i \leq n} \subseteq (k_i)_{1 \leq i \leq m}$  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

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

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

*Démonstration.* Une valeur ne peut être que de deux formes :  $\lambda x : S_1. t_1$  ou  $\{k_i = v_i\}^{1 \leq i \leq m}$ .

- $v$  est de type  $T_1 \rightarrow T_2$ . Les seules règles permettant de montrer  $\vdash v : T_1 \rightarrow T_2$  sont (T-SUB) et (T-ARROW).
  - (T-ARROW). Le résultat est direct.
  - (T-SUB). Nous avons alors :

$$\frac{\vdash v : S \quad S <: T_1 \rightarrow T_2}{\vdash v : T_1 \rightarrow T_2}$$

En utilisant le lemme d'inversion des règles de sous-typage, nous obtenons  $S = S_1 \rightarrow S_2$  avec  $T_1 <: S_1$  et  $S_2 <: T_2$ .

En utilisant l'hypothèse de récurrence sur  $\vdash v : S_1 \rightarrow S_2$ , nous avons  $v$  qui est une abstraction.

- $v$  est de type  $\{l_i : T_i\}^{1 \leq i \leq n}$ . Les seules règles possibles sont (T-SUB) et (T-RCD). Le résultat est direct pour (T-RCD). Quant à (T-SUB), nous utilisons les mêmes arguments que précédemment.

□

**Théorème III.6** (de progression). *Si  $t$  est un terme bien typé sans variable libre, alors soit  $t$  est une valeur, soit il existe  $t'$  tel que  $t \rightarrow t'$ .*

*Démonstration.* Par induction sur l'arbre de dérivation de typage de  $t$ .  $t$  ne peut pas être une variable car  $t$  est clos et si  $t = \lambda x : S_1. t_1$ , le résultat est direct car les abstractions sont des valeurs.

- (T-APP). Alors  $t = t_1 t_2$ ,  $\vdash t_1 : T_1 \rightarrow T_2$  et  $\vdash t_2 : T_1$ . Plusieurs cas sont possibles.
  - Si  $t_1$  n'est pas une valeur, (E-APP1) peut s'appliquer.
  - Si  $t_1$  est une valeur et  $t_2$  non, (E-APP2) s'applique.
  - Si  $t_1$  et  $t_2$  sont des valeurs, par le lemme des formes canoniques, comme  $\vdash t_1 : T_1 \rightarrow T_2$  et  $t_1$  n'a pas de variable libre,  $t_1 = \lambda x : S_1. t_{12}$  et (E-APPABS) s'applique.
- (T-RCD). Alors  $t = \{l_i = t_i\}^{1 \leq i \leq n}$  et  $\vdash t : \{l_i : T_i\}^{1 \leq i \leq n}$ . Plusieurs cas sont possibles :
  - si chaque  $t_i$  est une valeur, alors le résultat est direct car  $t$  est une valeur.

- s'il existe  $i$  tel que  $t_i$  n'est pas une valeur, alors nous utilisons l'hypothèse de récurrence sur  $t_i$  et (E-RCD) s'applique.
- (T-PROJ). Nous avons alors  $t = t_1.l_j$  et  $\vdash t_1 : T$  où  $T = \{l_i : T_i\}^{1 \leq i \leq n}$ . Par hypothèse de récurrence sur  $\vdash t_1 : T$ ,  $t_1$  est soit une valeur, soit il existe  $t'_1$  tel que  $t_1 \rightarrow t'_1$ . Si  $t_1$  est une valeur, nous utilisons le lemme des formes canoniques et (E-PROJ-RCD) s'applique. Sinon, (E-PROJ) s'applique.
- (T-SUB). Il suffit d'appliquer l'hypothèse de récurrence.

□

### 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** 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
$t \rightarrow t$	fonction
$\{l_i : t_i\}^{1 \leq i \leq n}$	enreg
$Top$	Top
$Bottom$	Bottom

Nous ajoutons les règles de typage suivantes :

$$(S-TOP) \quad T <: Top \qquad Bottom <: T \quad (S-BOTTOM)$$

Nous utiliserons ceux-ci dans DOT.

## Chapitre IV

# Système F

Dans ce chapitre, nous allons introduire une autre notion de polymorphisme appelée le *polymorphisme paramétré*<sup>1</sup>

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 diffèrent uniquement par le type du paramètre.

Supposons que nous ayons  $\mathbb{N}$  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  $\mathbb{N} <: \mathbb{R}$ . Dans tous les cas, la fonction  $id_1$  étant de type  $\mathbb{R} \rightarrow \mathbb{R}$ , le type de la valeur de retour sera  $\mathbb{R}$ , et non  $\mathbb{N}$  si nous passons un naturel à la fonction. Dans le cas des enregistrements, cela implique que nous perdons des champs de l'enregistrement passé en paramètre.

De plus, 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 au  $\lambda$ -calcul simplement typé. Le calcul qui en résulte est appelé *Système F*.

### IV.1 Syntaxe

La syntaxe de *Système F* est très proche de celle du  $\lambda$ -calcul simplement typé : nous ajoutons un terme qui permet de créer une fonction prenant un type

---

1. Plus d'informations peuvent être trouvées dans [6]

et retournant un terme (l'équivalent de l'abstraction sur les termes) ainsi qu'un terme qui permet d'appliquer un terme à un type.

Du côté des types, nous ajoutons des variables de type qui nous servent dans les abstractions de type. Nous supposons, comme pour les variables de terme, qu'il existe un nombre dénombrable de variables de type. Nous ajoutons également un type pour les abstractions de type, appelé *type universel*.

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

$t ::=$	terme	$T ::=$	type
$x$	var	$X$	type var
$tt$	app	$T \rightarrow T$	fonction
$\lambda x : T. t$	abs	$\forall X. T$	universel
$\Lambda X. t$	type abs		
$t[T]$	type app		
$v ::=$	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  $\Lambda X. t$  :

$$\begin{aligned} [x := s] (t[T]) &= ([x := s]t)[T] \\ [x := s] (\Lambda X. t) &= \Lambda X. ([x := s]t) \end{aligned}$$

Nous définissons également une fonction de substitution de variable de type pour un type donné, notée  $[X \rightarrow S]$ . Comme les types peuvent apparaître aussi bien dans les termes que dans les types, cette fonction est définie sur ces deux ensembles.

**Définition IV.1.** Soit  $X$  une variable de type et  $S$  un type. On définit la fonction de substitution de  $X$  par  $S$ , noté  $[X \rightarrow S]$ , sur les termes et types de la manière suivante :

$$\begin{aligned}
[X \rightarrow S]x &= x \\
[X \rightarrow S]\lambda x : T. t &= \lambda x : [X \rightarrow S]T. [X \rightarrow S]t \\
[X \rightarrow S](t_1 \ t_2) &= ([X \rightarrow S]t_1) ([X \rightarrow S]t_2) \\
[X \rightarrow S](\Lambda Y. t) &= \Lambda Y. ([X \rightarrow S]t) \\
[X \rightarrow S](t[T]) &= ([X \rightarrow S]t)[X \rightarrow S]T \\
[X \rightarrow S]X &= S \\
[X \rightarrow S]Y &= Y \\
[X \rightarrow S](T_1 \rightarrow T_2) &= ([X \rightarrow S]T_1) \rightarrow ([X \rightarrow S]T_2) \\
[X \rightarrow S](\forall Y. T) &= \forall Y. [X \rightarrow S]T
\end{aligned}$$

## 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).

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

(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.

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.

$$id_{poly}[\mathbb{N}] \rightarrow \lambda x : \mathbb{N}. x \quad id_{poly}[\mathbb{R}] \rightarrow \lambda x : \mathbb{R}. x$$

## 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  $t[T]$  :

$$\begin{array}{c}
\text{(T-T-ABS)} \quad \frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X. t : \forall X. T} \qquad \frac{\Gamma \vdash t_1 : \forall X. T}{\Gamma \vdash t_1[T'] : [X \rightarrow T']T} \text{(T-T-APP)}
\end{array}$$

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

## IV.5 Sûreté

Les théorèmes de préservation et de progression peuvent également être démontrés pour Système F. Comme nous avons pu le remarquer, Système F est une extension relativement simple du  $\lambda$ -calcul simplement typé. Les nouvelles règles de typage et d'évaluation influencent uniquement 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  $\lambda$ -calcul simplement typé restant vrais.

**Lemme IV.2** (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  $T_2 = [X \rightarrow T_1]T$ .*

*Démonstration.* Direct vu les règles de typage.  $\square$

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

**Lemme IV.3** (des formes canoniques). *1. Si  $\Gamma \vdash v : \forall X. T$ , alors  $v = \Lambda X. t$  et  $\Gamma, X \vdash t : T$ .*  
*2. Si  $\Gamma \vdash v : T_1 \rightarrow T_2$ , alors  $v = \lambda x : T_1. t$  et  $\Gamma, x : T_1 \vdash t : T_2$ .*

*Démonstration.* Direct vu les règles de typage.  $\square$

**Théorème IV.4** (de progression). *Soit  $t$  un terme bien typé sans variable libre. Alors, soit  $t$  est une valeur, soit il existe  $t'$  tel que  $t \rightarrow t'$ .*

*Démonstration.* 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  $\vdash t_1 : \forall X. T_1$ . L'hypothèse de récurrence nous dit que soit  $t_1$  est une valeur, soit il existe  $t'_1$  tel que  $t_1 \rightarrow t'_1$ .

- Si  $t_1$  est une valeur, par le lemme des formes canoniques,  $t_1 = \Lambda X. t$  et (E-T-ABS) s'applique.
- Si  $t_1$  se réduit en  $t'_1$ , alors (E-T-APP) s'applique.

$\square$

## Préservation

**Lemme IV.5** (de substitution de variable de type). *Soit  $S$  un type.*

*Si*

$$\Gamma, X, \Delta \vdash t : T$$

*alors*

$$\Gamma, [X \rightarrow S]\Delta \vdash [X \rightarrow S]t : [X \rightarrow S]T$$

où  $[X \rightarrow S]\Delta$  signifie que nous remplaçons toutes les occurrences de  $X$  par  $S$  dans le contexte  $\Delta$ .

*Démonstration.* La preuve se réalise par récurrence sur l'arbre de dérivation. Nous ne démontrons que le cas (T-ABS) où  $\Delta$  est nécessaire.

(T-ABS). Nous avons

$$\Gamma, X, \Delta \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2$$

Par le lemme d'inversion, nous obtenons

$$\Gamma, X, \Delta, x : T_1 \vdash t_2 : T_2$$

Par hypothèse de récurrence,

$$\Gamma, [X \rightarrow S]\Delta, x : [X \rightarrow S]T_1 \vdash [X \rightarrow S]t_2 : [X \rightarrow S]T_2$$

En utilisant (T-ABS), nous avons

$$\Gamma, [X \rightarrow S]\Delta \vdash \lambda x : [X \rightarrow S]T_1. [X \rightarrow S]t_2 : [X \rightarrow S]T_2$$

En utilisant la définition de la fonction de substitution, nous concluons

$$\Gamma, [X \rightarrow S]\Delta \vdash [X \rightarrow S]\lambda x : T_1. t_2 : [X \rightarrow S]T_2$$

□

**Théorème IV.6** (de préservation du typage). *Soit  $\Gamma \vdash t : T$  et  $t \rightarrow t'$ . Alors  $\Gamma \vdash t' : T$ .*

*Démonstration.* . Nous procédons par induction sur l'arbre de dérivation du jugement de typage  $\Gamma \vdash t : T$ . Par rapport à la preuve du  $\lambda$ -calcul simplement typé, il nous suffit d'ajouter des cas pour (T-T-ABS) et (T-T-APP).

- (T-T-ABS). Ce cas n'est pas possible car il n'existe pas de règle d'évaluation.
- (T-T-APP). Nous avons  $t = t_1[T_1]$ . Par le lemme d'inversion, nous avons  $\Gamma \vdash t_1 : \forall X. T_{12}$  et  $T = [X \rightarrow T_1]T_{12}$ . Les deux règles d'évaluation possibles sont (E-T-ABS) et (E-T-APP).
- (E-T-APP). Nous utilisons l'hypothèse de récurrence et (E-T-APP).
- (E-T-ABS). Alors,  $t_1 = \Lambda X. t_{12}$  avec  $\Gamma, X \vdash t_{12} : T_{12}$  et  $t'_1 = [X \rightarrow T_1]t_{12}$ . Par IV.5, nous avons  $\Gamma \vdash [X \rightarrow T_1]t_{12} : [X \rightarrow T_1]T_{12}$ . Nous concluons en se souvenant que  $[X \rightarrow T_1]T_{12} = T$ .

□





# Chapitre V

## Système $F_{<}$ :

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 mécanismes 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 permis 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 abstractions de type. Par exemple, nous pouvons définir la fonction identité polymorphe  $\Lambda X. \lambda x : X. x$  et l'appliquer aux types  $\mathbb{R}$  et  $\mathbb{N}$ , mais également à n'importe quel autre type comme  $\mathbb{N} \rightarrow \mathbb{R}$  ou encore  $\mathbb{R} \rightarrow (\mathbb{N} \rightarrow \mathbb{R})$ .

Dans ce chapitre, nous allons unifier ces deux notions en un langage appelé Système  $F_{<}$ .<sup>1</sup> L'idée principale de Système  $F_{<}$  est d'élargir notre relation de sous-typage sur les variables de type pour les borner supérieurement<sup>2</sup> 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  $\mathbb{N} \rightarrow \mathbb{R}$  ou encore  $\mathbb{R} \rightarrow \mathbb{R}$ . Nous ne considérons donc plus seules les variables de type, mais liées à une borne supérieure. Nous notons  $X <: T$  pour dire que la variable de type  $X$  est bornée 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.

Dans ce chapitre, nous considérons également le type **Top**.

---

1. prononcé « système F sub »

2. Nous nous restreignons à la borne supérieure. Une borne inférieure sera ajoutée dans DOT.

## V.1 Syntaxe

La syntaxe de Système  $F_{<}$  est celle de Système  $F$  à laquelle nous ajoutons une borne supérieure aux variables des abstractions de type :

$t ::=$	terme	$T ::=$	type
$x$	var	$X$	type var
$tt$	app	$T \rightarrow T$	fonction
$\lambda x : T. t$	abs	$\forall X <: T. T$	universel
$\Lambda X <: T. t$	type abs	$Top$	Top
$t[T]$	type app		

Les valeurs restent les mêmes :

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

## V.2 Sémantique

Au niveau de la sémantique, les règles restent les mêmes que pour Système  $F$  :

$$\begin{array}{c}
\frac{t_1 \rightarrow t'_1}{t_1 t \rightarrow t'_1 t} \quad (\text{E-APP1}) \qquad \frac{t \rightarrow t'}{vt \rightarrow vt'} \quad (\text{E-APP2}) \\
\\
(\lambda x : T. t)v \rightarrow [x := v]t \quad (\text{E-APPABS}) \qquad \frac{t \rightarrow t'}{t[T] \rightarrow t'[T]} \quad (\text{E-T-APP}) \\
\\
(\Lambda X <: T_1. t)[T] \rightarrow [X \rightarrow T]t \quad (\text{E-T-ABS})
\end{array}$$

## V.3 Contexte de typage

Au niveau du contexte de typage, à la place d'avoir uniquement la variable de type, nous allons également ajouter la borne supérieure.

Il est nécessaire de faire attention à l'ordre d'apparition des variables. Par exemple, nous dirons que  $X <: Y, Y <: Top$  est mal formé car  $Y$  apparaît comme borne avant sa définition ( $Y <: Top$ ). Bien que ça soit une définition informelle, nous considérerons uniquement les contextes de typage tels qu'une variable  $Y$  n'apparaît jamais comme borne supérieure à gauche de sa définition.

La syntaxe du contexte de typage est donc :

$$\begin{array}{l} \Gamma ::= \\ \emptyset \\ \Gamma, x : T \\ \Gamma, X <: T \end{array} \quad \text{contexte}$$

## V.4 Règles de typage et de sous-typage

Comme le contexte comporte des hypothèses de sous-typage, nous ajoutons un contexte  $\Gamma$  dans les règles de sous-typage. Une affirmation  $S <: T$  devient  $\Gamma \vdash S <: T$ .

Les règles de typage deviennent :

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

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

$$\frac{\Gamma, X <: T_1 \vdash t : T}{\Gamma \vdash \Lambda X <: T_1. t : \forall X <: T_1. T} \quad (\text{T-T-ABS})$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_1. T \quad \Gamma \vdash T' <: T_1}{\Gamma \vdash t_1[T'] : [X \rightarrow T']T} \quad (\text{T-T-APP})$$

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

Quant aux règles de sous-typage, nous avons :

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

$$\Gamma \vdash S <: Top \quad (\text{S-TOP})$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad (\text{S-TVAR}) \qquad \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S-ARROW})$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1. S_2 <: \forall X <: T_1. T_2} \quad (\text{S-ALL})$$

Remarquons que dans (S-ALL), la borne supérieur de la variable  $X$  peut être différente.<sup>3</sup>

3. Une variante, appelée Kernel  $F_{<}$ , oblige la borne supérieure à être la même. Nous ne discuterons pas de celle-ci dans ce document.

## V.5 Sûreté

Les théorèmes de préservation et de progression restent vrais pour Système  $F_{<}$ . Cependant, nous ne les démontrerons pas car ils nécessitent des lemmes techniques ainsi qu'une définition formelle de contexte bien formé 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 [7].

## V.6 Indécidabilité du sous-typage

Étant donnés deux types  $S$  et  $T$ , la question  $S <: T$  a-t-elle toujours une réponse ? En d'autres termes, la question du sous-typage est-elle décidable ?

Il a été démontré qu'il n'existe pas d'algorithme correct et complet de sous-typage donnant une réponse sur toute question  $S <: T$ . Un algorithme  $A(\Gamma, S, T)$  est *correct* si lorsque  $A(\Gamma, S, T)$  a comme sortie oui,  $\Gamma \vdash S <: T$ . Un algorithme est *complet* si, lorsque  $\Gamma \vdash S <: T$ , la sortie de  $A(\Gamma, S, T)$  est oui.

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

Le problème de l'indécidabilité du sous-typage de Système  $F_{<}$  est important quand nous considérons l'écriture d'un algorithme de sous-typage pour ce calcul. En effet, cela implique que sur certaines entrées, l'algorithme peut diverger<sup>4</sup>.

---

4. Si bien sûr nous ne définissons pas des règles pour l'arrêter dans le cas où il n'y a pas de réponse.

# Chapitre VI

## DOT

Dans ce chapitre, nous présentons DOT, un calcul développé récemment pour le langage Scala. Ce calcul ajoute les types, alors appelés types chemin dépendants, dans les enregistrements. Des types rékursifs, c'est-à-dire des types qui peuvent faire référence à eux-mêmes, sont également définis<sup>1</sup>. Nous montrons que DOT peut être vu comme une extension de Système  $F_{<}$ : bien que sa syntaxe soit différente et ne possède pas de variables de type.

Plusieurs définitions du calcul DOT existent et sont dispersées à travers plusieurs documents comme [1], [17], [2] ou encore [18]. Dans ce document, nous avons fait le choix d'utiliser [18] car la syntaxe et les règles sont proches des calculs présentés dans ce mémoire.

### VI.1 Syntaxe

La syntaxe des termes de DOT est définie par la grammaire suivante :

$t ::=$	terme		
$x, y$	var		
$\lambda x : T. t$	abs	$d ::=$	decl
$x y$	app	$\{a = t\}$	champ
$\text{let } x = t \text{ in } t$	let	$\{A = T\}$	type
$\nu(x : T^x)d$	rec	$d \wedge d$	aggregation
$x.a$	champ proj		

et la syntaxe des types par la grammaire suivante :

---

1. Les types rékursifs et leurs règles présents dans DOT ne sont pas définis de manière habituelle. Plusieurs chapitres y sont consacrés dans [11]. Nous n'étudierons pas les différences dans ce document.

$S, T ::=$	type
$Top$	top
$Bottom$	bottom
$\forall(x : S)T^x$	fonction
$\{A : S..T\}$	type decl
$\{a : T\}$	champ decl
$x.A$	type proj
$\mu(x : T^x)$	rec
$S \wedge T$	inter

Nous retrouvons (var) et (abs) pour les variables et les abstractions. À la différence des autres calculs, une application n'est pas constituée de deux termes mais de deux variables<sup>2</sup>. Nous retrouvons la projection d'un champ avec (champ proj). Comme pour les applications, seule une variable peut être utilisée<sup>3</sup>.

(let) permet de créer des variables locales comme il est possible de le faire en OCaml. (let) est utilisé pour lier un terme à une variable et pouvoir utiliser cette dernière dans un autre terme<sup>4</sup>.

Enfin, (rec) est l'équivalent des enregistrements à la différence qu'il permet de définir des termes récurifs, c'est-à-dire des termes dont le corps peut faire référence à eux-mêmes à travers la variable  $x$  qui doit obligatoirement être accompagnée d'un type. Le corps d'un terme récurif est une déclaration. Une déclaration est soit un couple  $(a, t)$  où  $a$  est un nom de champ et  $t$  un terme comme pour les enregistrements, soit une déclaration de type représentée par un couple  $(A, T)$  où  $A$  est le nom de la déclaration et  $T$  le type associé. Un enregistrement peut contenir plusieurs déclarations de champ et de type en utilisant (aggregation). Ces déclarations peuvent être mutuellement dépendantes grâce à la variable  $x$ . Le domaine d'une déclaration  $d$ , noté  $dom(d)$ , est défini comme l'ensemble des labels.

Quant aux types, nous retrouvons  $Top$  et  $Bottom$  comme décrits dans le chapitre III. Le type fonction est également présent avec une syntaxe différente et il devient dépendant, c'est-à-dire que le type de retour peut dépendre du paramètre. (champ decl) est le type d'un champ d'un enregistrement. (type decl) est le type d'une déclaration d'un type dans un enregistrement et possède une borne inférieure  $S$  et une borne supérieure  $T$ . (inter) permet de typer le corps d'un enregistrement contenant plusieurs déclarations et ce dernier est encapsulé dans un type récurif grâce à (rec). Les types des déclarations peuvent être mutuellement dépendants grâce à la variable  $x$  du type récurif.

Enfin, les types dépendants (type proj), en parallèle de (champ proj), permettent d'accéder à un type interne d'un enregistrement.

Des exemples concrets seront donnés dans le chapitre VII.

2. Nous pouvons cependant utiliser (let) pour retrouver des applications entre termes.

3. Même remarque que pour les applications.

4. On dit aussi que  $x$  est un binding local.

## VI.2 Sémantique

La sémantique est laissée de côté actuellement.

## VI.3 Règles de typage

Nous séparons les règles de typage en deux catégories : celles pour les termes et celles pour les définitions.

Les règles de typage pour les termes sont les suivantes :

$$\begin{array}{c}
\Gamma, x : T, \Gamma' \vdash x : T \quad (\text{T-VAR}) \qquad \frac{\Gamma \vdash t : S \quad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \quad (\text{T-SUB}) \\
\\
\frac{\Gamma, x : T \vdash t : U \quad x \notin FV(T)}{\Gamma \vdash \lambda x : T. t : \forall (x : T) U^x} \quad (\text{ALL-I}) \\
\\
\frac{\Gamma \vdash x : \forall (z : S) T^z \quad \Gamma \vdash y : S}{\Gamma \vdash x y : [z := y] T^z} \quad (\text{ALL-E}) \\
\\
\frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash u : U \quad x \notin FV(U)}{\Gamma \vdash \text{let } x = t \text{ in } u : U} \quad (\text{LET}) \\
\\
\frac{\Gamma \vdash x : T \quad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \quad (\text{AND-I}) \\
\\
\frac{\Gamma, x : T \vdash d : T}{\Gamma \vdash \nu(x : T^x) d : \mu(x : T^x)} \quad (\{ \} \text{-I}) \qquad \frac{\Gamma \vdash x : T^x}{\Gamma \vdash x : \mu(z : T^z)} \quad (\text{VAR-PACK}) \\
\\
\frac{\Gamma \vdash x : \mu(z : T^z)}{\Gamma \vdash x : T^x} \quad (\text{VAR-UNPACK}) \qquad \frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \quad (\text{FLD-E})
\end{array}$$

Les règles de typage pour les définitions sont :

$$\begin{array}{c}
\frac{\Gamma \vdash \{A = T\}}{\Gamma \vdash \{A : T..T\}} \quad (\text{TYP-I}) \\
\\
\frac{\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \quad \text{dom}(d_1) \cap \text{dom}(d_2) = \emptyset}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \quad (\text{ANDDEF-I}) \\
\\
\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a : t\} : \{a : T\}} \quad (\text{FLD-I})
\end{array}$$

Nous retrouvons les règles (T-VAR) et (T-SUB) comme dans les précédents calculs. (ALL-I) est l'équivalent de (T-ABS) et (ALL-E) est l'équivalent de (T-APP). (LET) nous dit que le terme  $t$  et la variable  $x$  à laquelle nous lions ce terme doivent avoir le même type, le type du terme en entier est celui de  $u$ .

(AND-I) nous dit que si une variable a deux types  $T$  et  $U$ , alors elle a également le type intersection  $T \wedge U$ . ( $\{\}$ -I) type les termes récurifs et oblige la variable du terme récurif à avoir le même type que celui dans la déclaration. (FLD-E) est l'équivalent de (T-PROJ) que nous avons défini pour les enregistrements. Pour finir, (VAR-PACK) nous dit que nous pouvons contruire un type récurif à partir de n'importe quel autre type. De l'autre coté, (VAR-UNPACK) nous dit qu'un terme ayant un type récurif possède également le type à l'intérieur de ce dernier.

Quant aux définitions, (TYP-I) donne le même type à la borne inférieure et à la borne supérieure d'une déclaration de type. (FLD-I) ressemble à (T-RCD) restreint à un enregistrement avec un champ. (ANDDEF-I) type l'union de deux déclarations dont les champs ont obligatoirement des noms différents.

## Problème d'échappement

Supposons que nous ayons un type  $\mathbb{R}$  et les nombres et prenons l'exemple suivant :

$$\begin{aligned} \text{let } y = & \\ & \nu(x : \mu(z : \{A : \text{Bottom}..Top\} \wedge \{a : z.A\})) \{A = \mathbb{R}\} \wedge \{a = 5\} \\ \text{in } y.a \end{aligned}$$

Le type de l'expression est  $y.A$ . Cependant, la variable  $y$  n'existe pas en dehors du let car c'est une variable locale. C'est un exemple du *problème d'échappement*<sup>5</sup>. C'est pour éviter cela que dans la règle (LET) la condition  $x \notin FV(U)$  est présente.

Nous verrons dans le chapitre VII qu'il est nécessaire d'y faire attention lors de l'implémentation, en particulier lorsque nous faisons des bindings locaux de variables.

---

5. « Avoidance problem » en anglais.



## VI.4 Règles de sous-typage

$$\begin{array}{c}
\Gamma \vdash T <: Top \quad (\text{S-TOP}) \qquad \Gamma \vdash Bottom <: T \quad (\text{S-BOTTOM}) \\
\\
\frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad (\text{S-TRANS}) \qquad \Gamma \vdash T <: T \quad (\text{S-REFL}) \\
\\
\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1)T_1 <: \forall(x : S_2)T_2} \quad (\text{ALL} < : \text{ALL}) \\
\\
\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad (< : \text{SEL}) \qquad \frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\text{SEL} < : ) \\
\\
\Gamma \vdash T \wedge U <: T \quad (\text{AND-1-} < : ) \qquad \Gamma \vdash T \wedge U <: U \quad (\text{AND-2-} < : ) \\
\\
\frac{\Gamma \vdash S <: T \quad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \quad (< : \text{AND}) \\
\\
\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \quad (\text{TYP} < : \text{TYP}) \\
\\
\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a : T\} <: \{a : U\}} \quad (\text{FLD} < : \text{FLD})
\end{array}$$

Comme pour le chapitre III, nous avons (S-REFL), (S-TRANS), (S-TOP), (S-BOTTOM) et l'équivalent de (S-ARROW), (ALL< :ALL).

(< : SEL) (resp. (SEL < :)) nous dit qu'une projection de type est un super-type (resp. sous-type) de sa borne inférieure (resp. supérieure).

(FLD < : FLD) est l'équivalent de (S-RCD-DEPTH) pour un enregistrement à un unique champ.

À travers (TYP< :TYP), les déclarations de type sont naturellement contravariantes pour les bornes inférieures et covariantes pour les bornes supérieures.

Pour les intersections, les règles (AND-1-< : ) et (AND-2-< : ) nous donnent l'équivalent de (S-RCD-WIDTH). Quant à (< :AND), cette dernière règle nous dit que si un même type est sous-type de deux types différents, alors il est également sous-type de leur intersection.

Bien que l'analogie avec les enregistrements et les règles de sous-typage du chapitre III permette de comprendre l'utilité des règles, il faut remarquer que ces dernières sont plus expressives. En effet, le type intersection n'est pas défini uniquement pour les déclarations, mais pour n'importe quel type. Cela signifie qu'il est autorisé d'écrire  $\mu(x : T) \wedge \{a : S..T\}$  ou encore  $x.A \wedge \{a : S..T\} \wedge y.A \wedge \mu(x : U)$ . De même, la variable  $x$  dans un type dépendant n'est pas nécessairement un terme récursif mais peut être une abstraction. Par exemple, nous pouvons écrire *let*  $y = \lambda x : T. t$  *in*  $y.A$ . Bien que ça n'ait pas de sens, la syntaxe le permet. La difficulté de DOT réside dans cette expressivité.

## VI.5 Problème de mauvaises bornes

DOT permet de définir des déclarations de types avec une borne inférieure et une borne supérieure. Cette liberté dans les types, bien qu'intéressante à première vue, pose un problème dans le système de sous-typage. En effet, le type  $\{A : Top..Bottom\}$  est autorisé. Cependant, si  $x : \{A : Top..Bottom\}$ , alors nous pouvons montrer que  $Top <: Bottom$  en utilisant ( $SEL < :$ ), ( $< : SEL$ ) et la transitivité. Nous avons alors  $\Gamma, x : \{A : Top..Bottom\} \vdash S <: T$  pour tous  $S$  et  $T$ .<sup>6</sup>

Nous disons qu'un type est *bien formé* s'il ne possède pas de mauvaises bornes, sinon nous disons qu'il est *mal formé*. Nous disons qu'un contexte est *bien formé* s'il ne contient que des variables dont les types sont bien formés. Sinon, le contexte est dit *mal formé*.

## VI.6 Encodage de Système $F_{<}$

Bien que la syntaxe soit différente de Système  $F_{<}$ , et que DOT ne comporte pas de variables de type, nous allons montrer que DOT permet d'encoder Système  $F_{<}$ . De plus, les jugements de typage vrais pour le système de typage et de sous-typage de Système  $F_{<}$  restent vrais pour le système de typage et de sous-typage de DOT à travers cet encodage.

Notons  $\mathcal{V}_{DOT}$  l'ensemble des variables de DOT et  $\mathcal{V}_{T,F}$  l'ensemble des variables de type de Système  $F_{<}$ . Ces ensembles étant infinis dénombrables, il existe une fonction injective  $f$  entre  $\mathcal{V}_{T,F}$  et  $\mathcal{V}_{DOT}$ . Nous notons  $x_X \in \mathcal{V}_{DOT}$  l'image de  $X \in \mathcal{V}_{T,F}$  par la fonction  $f$ .

Nous définissons alors la fonction  $*$  qui à chaque terme (resp. type) de Système  $F_{<}$  associe un terme (resp. type) de DOT.

$$\begin{aligned}
X^* &= x_X.A \\
Top^* &= Top \\
(S \rightarrow T)^* &= \forall(x : S^*)T^* \\
(\forall X <: S.T)^* &= \forall(x_X : \{A : Bottom..S^*\})T^* \\
\\ 
x^* &= x \\
(\lambda x : T.t)^* &= \lambda x : T^*.t^* \\
(\Lambda X <: S.t)^* &= \lambda x : \{A : Bottom..S^*\}.t^* \\
(t \ u)^* &= let\ x = t^*\ in \\
&\quad let\ y = u^*\ in \\
&\quad x\ y \\
(t[U])^* &= let\ x = t^*\ in \\
&\quad let\ y_Y = \nu(z : \{A : U^*..U^*\})\{A = U^*\}\ in \\
&\quad x\ y_Y
\end{aligned}$$

6. Divers articles sur DOT [17] avancent que restreindre la déclaration (type)  $\{A = T\}$  à une égalité permet d'éviter ce problème lors de l'exécution.

Quant aux contextes, la fonction  $*$  est définie naturellement inductivement par :

$$\begin{aligned}(X <: Top)^* &= x_X : \{A : Bottom..Top\} \\ (x : T)^* &= x : T^*\end{aligned}$$

Cependant, DOT est plus riche syntaxiquement que Système  $F_{<}$  : car, par exemple, il n'est pas possible en  $F_{<}$  de donner une borne inférieure à la variable d'une abstraction alors que DOT le permet.

Notons  $\Gamma \vdash_F t : T$  (resp.  $\Gamma \vdash_D t : T$ ) un jugement de typage pour le système de typage de Système  $F_{<}$  (resp. DOT). Nous faisons de même pour  $\Gamma \vdash_F S <: T$ .

Nous utilisons la même technique que précédemment, c'est-à-dire l'induction sur les arbres de dérivation.

**Théorème VI.1.** *Si  $\Gamma \vdash_F S <: T$ , alors  $\Gamma^* \vdash_D S^* <: T^*$ .*

*Démonstration.* — (S-TVAR). Utilisation de  $(< : \text{SEL})$ .

- (S-ALL). Pour le membre de gauche, nous utilisons  $(\text{TYP} < : \text{TYP})$ .
- Les cas (S-REFL), (S-TRANS), (S-TOP) et (S-ARROW) sont directs en utilisant l'encodage et l'hypothèse de récurrence.

□

**Théorème VI.2.** *Si  $\Gamma \vdash_F t : T$ , alors  $\Gamma^* \vdash_D t^* : T^*$ .*

Nous acceptons ce théorème. Une preuve peut être trouvée dans [18] et se fait par induction sur la dérivation  $\Gamma \vdash_F t : T$ . Celle-ci repose sur deux lemmes intermédiaires pour la substitution des types dépendants.

Cette propriété de DOT implique que la question du sous-typage de DOT est indécidable[18, 1].<sup>7</sup>

## VI.7 Sûreté

Les théorèmes de préservation et de progression restent vrais. Cependant, nous ne les démontrerons pas dans ce document car ils nécessitent plusieurs lemmes techniques. Différentes preuves de la sûreté de DOT existent, en utilisant des techniques et des sémantiques différentes. Nous redirigeons le lecteur aux références données au début de ce chapitre pour plus d'informations.

---

7. Plus précisément, il est nécessaire de démontrer  $\Gamma^* \vdash_D S^* <: T^* \Rightarrow \Gamma \vdash_F S <: T$ . Montrer ce dernier lemme n'est pas évident à cause de (S-TRANS) qui peut faire apparaître un type qui ne provient pas obligatoirement d'un type de Système  $F_{<}$ . Néanmoins, d'après certains articles, (S-TRANS) peut être poussé plus haut dans l'arbre de dérivation si celui-ci est utilisé comme dernière règle. Cette question n'a pas été plus étudiée en profondeur mais nous accepterons ces faits pour la suite.



# Chapitre VII

## RML

Dans ce chapitre, nous proposons un algorithme de typage et de sous-typage ainsi qu'un langage de surface pour DOT. Une implémentation en OCaml du langage et des algorithmes sont disponibles sur GitHub[\[19\]](#) sous le nom de RML.

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.

Sur la page principale du projet se trouve, en anglais, une description complète de la syntaxe de RML, du but du langage ainsi que des exemples. 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.

### 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, une syntaxe proche de celle d'OCaml est utilisée. En particulier, la syntaxe des modules OCaml est utilisée pour les enregistrements 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.

Les tableaux ci-dessous regroupent l'ensemble des correspondances syntaxiques entre DOT et RML.

<i>DOT</i>	<i>RML</i>
Bottom	Nothing
Top	Any
$\{A : \text{Bottom} \dots \text{Top}\}$ $\{A : S \dots T\}$ $\{A : S \dots \text{Top}\}$ $\{A : \text{Bottom} \dots T\}$ $\{a : T\}$ $d_1 \wedge d_2$	type a type a = S .. T type a :> S type a <: T val a : T $d_1 \ d_2$
$\mu(x : T)$ $\mu(\text{self} : T)$	sig(x) T end sig T end
$\forall(x : S)T$ $\forall(x : S)T$	$S \rightarrow T$ (si $x \notin FV(T)$ ) forall(x : S) T
$T_1 \wedge T_2$ $T \wedge \{A : L \dots U\}$ $T \wedge \{A : L \dots U\} \wedge \{A' : L' \dots U'\}$	$T_1 \ \& \ T_2$ T with type a = L .. U T with type a = L .. U and a' = L' .. U'

TABLE VII.1 – Correspondance entre DOT et RML pour la syntaxe des types

<i>DOT</i>	<i>RML</i>
$\lambda x : T. t$	$\text{fun}(x : T) \rightarrow t$
$\{A = T\}$ $\{a = t\}$	type a = T let a = t
$\nu(x : T)d$ $\nu(\text{self} : T)d$	sig(x) T end : struct(x) d end sig T end : struct d end
let m = $\nu(x : T)d$ in t	let module M = ... in t

TABLE VII.2 – Correspondance entre DOT et RML pour la syntaxe des termes

Le langage de surface est entièrement décrit sur la page du projet[19]. 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 ASTs

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[14] est une librairie basée sur visitors[15] qui fournit des macros<sup>1</sup> 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 chaînes 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. d'obtenir 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 les fonctions dépendantes.

La figure 5 montre l'implémentation de la syntaxe abstraite des termes.

Plus d'informations et d'explications sur AlphaLib et son fonctionnement peuvent être trouvées dans [16].

## VII.3 Contexte de typage

Un contexte est implémenté comme un dictionnaire dont les clefs sont des atomes et les valeurs sont les types nominatifs de ces atomes. Cette implémentation se trouve dans le fichier `typing/contextType.ml` et contient également des fonctions pour afficher un contexte donné. Une fonction `find` est utilisée pour récupérer le type d'une variable depuis un contexte donné.

---

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 RML 5 – Implémentation de la grammaire des termes officiels de DOT en utilisant AlphaLib. `field_label` est un alias de type pour string.

## VII.4 Meilleures bornes d'un type dépendant

Un problème qui se pose lors de l'écriture des algorithmes, est : étant donné une variable  $x$  et un contexte, quelle est la borne inférieure ou la borne supérieure du type dépendant  $x.A$ ? La question se pose, par exemple, quand nous devons comparer deux types dépendants  $x.A$  et  $y.A'$ .

En d'autres termes, pour la borne supérieure, pour une variable  $x$  de type  $T$  et un champ  $A$ , quel est le plus petit type  $U$  tel que  $T <: \{A : L..U\}$ ?

Remarquons d'abord que la question n'a pas toujours de réponse. En effet, si  $x$  est de type `Top`<sup>2</sup>, le seul super-type de  $T$  est `Top`. En OCaml, nous représentons le résultat de l'algorithme par un type `option` où `None` est renvoyé si la question n'a pas de réponse et `Some(T)` si la réponse est  $T$ . Nous omettons le `Some` pour la description ci-dessous.

Cet algorithme n'est jamais discuté ni décrit dans les différents documents au sujet de DOT. Comprendre son utilité et son importance ainsi que son écriture n'a pas été facile. L'algorithme prend comme paramètres, en plus de la variable  $x$ , du type  $T$  et du label  $A$ , un contexte  $\Gamma$  et une direction. La direction est soit `Lower` pour la plus grande borne inférieure soit `Upper` pour la plus petite borne supérieure. L'algorithme travaille sur la structure de  $T$  :

- Si  $T = \text{Bottom}$ , nous renvoyons `Bottom` pour la borne supérieure et `None` pour la borne inférieure.
- Si  $T = \text{Top}$ , nous renvoyons `Top` pour la borne inférieure et `None` pour la borne supérieure.

---

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



- Si  $T = \forall(x : S_1)S_2$ , nous renvoyons *None* car nous ne pouvons pas comparer un enregistrement avec une fonction.
- Si  $T = \mu(y : S^y)$ , nous appelons récursivement l'algorithme sur  $S^x$  après avoir substitué  $y$  par  $x$  dans  $S$ .<sup>3</sup>
- Si  $T = T_1 \wedge T_2$ , nous appliquons l'algorithme récursivement sur  $T_1$  et sur  $T_2$ . Notons  $T'_1$  et  $T'_2$  les résultats. Plusieurs cas possibles :
  - Si  $T'_1 = T'_2 = \text{None}$ <sup>4</sup>, nous renvoyons *None*.
  - Si  $T'_1 = \text{None}$  et  $T'_2 \neq \text{None}$ , nous renvoyons  $T'_2$ .
  - Si  $T'_2 = \text{None}$  et  $T'_1 \neq \text{None}$ , nous renvoyons  $T'_1$ .
  - Sinon, nous renvoyons  $T'_1 \wedge T'_2$ .
- Si  $T = \{A : L..U\}$ , nous renvoyons  $U$  pour la borne supérieure et  $L$  pour la borne inférieure.
- Si  $T = \{a : U\}$ , nous renvoyons  $U$ <sup>5</sup>.
- Si  $T = y.A'$ , nous appelons récursivement l'algorithme avec les paramètres  $y$ , le type de  $y$ , le label  $A'$ , le contexte  $\Gamma$  et la direction et nous notons  $S$  la réponse.
  - Si  $S = \text{None}$ , nous renvoyons *None*. Cela signifie qu'il n'y avait déjà pas de réponse pour  $y$  (par exemple, si  $y$  est de type fonction).
  - Sinon, nous savons que  $S$  est le plus petit  $U$  tel que  $T_y <: \{A' : L..U\}$  où  $T_y$  est le type de  $y$ . Nous savons alors que  $S$  est la plus petite borne supérieure de  $T$ , c'est-à-dire le plus petit super-type de  $T$ . Nous appelons alors récursivement l'algorithme en remplaçant  $T$  par  $S$ .

L'algorithme actuel est satisfaisant sur différents exemples. Cependant, nous remarquons que le cas où  $T = y.A'$  n'est pas entièrement satisfaisant. Il serait nécessaire de montrer que la réponse pour  $T$  est la même que pour sa plus petite borne supérieure  $S$ .<sup>6</sup> Nous pouvons seulement dire que la réponse actuelle est un candidat, mais non le meilleur. De plus, l'algorithme ne termine pas toujours. En effet, prenons le cas où  $T_x = y.A$  et  $T_y = \mu(z : \{A : z.A..z.A\})$  et le label  $A'$ . Supposons que nous voulons la plus petite borne supérieure. Voici comment l'algorithme procède.

1. Comme  $T_x = y.A$ , nous devons appeler récursivement l'algorithme sur  $y$  avec le type  $T_y$  et le label  $A$ .
2. Comme  $T_y$  est un type récursif, nous appelons l'algorithme avec  $\{A : y.A..y.A\}$  (substitution de  $z$  par  $y$ ).
3. Comme le type en paramètre est une déclaration de type, nous renvoyons la borne supérieure, c'est-à-dire  $y.A$ .
4. Maintenant que nous avons calculé la borne supérieure de  $y.A$ , nous remplaçons à la première étape  $T_x$  avec cette valeur, qui est  $y.A$ . Or, c'est exactement la question initiale. L'algorithme va donc boucler.

L'implémentation OCaml est la fonction `best_bound_of_recursive_type`. `least_upper_bound_of_recursive_type` est l'algorithme pour la direction `Upper` tandis que `greatest_lower_bound_of_recursive_type` est pour la direction `Lower`.

3. Notons que la variable est importante dans ce cas, ce qui n'a pas été directement évident.

4. Cela peut se passer si  $T'_1$  et  $T'_2$  sont des fonctions.

5. Ceci afin de supporter de manière générale la projection.

6. Plusieurs tentatives de démonstrations ont échoué, et la possibilité que la complétude soit fautive a été envisagée. Cependant, vu les bons résultats, nous acceptons l'algorithme actuel.

Un algorithme semblable 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  $y.A'$ . L'implémentation OCaml est la fonction `least_upper_bound_of_dependent_function`.

## VII.5 Algorithme de typage

L'algorithme de typage se trouve dans le fichier `typer.ml` du dossier `typing`. L'implémentation utilise les règles (ALL-I), (ALL-E), (LET), ({}-I) et (FIELD-E) pour typer les termes correspondants. La règle (T-VAR) est remplacée par une recherche dans le contexte grâce à la fonction `find` discutée plus haut. Quant à (VAR-PACK)<sup>7</sup> et (VAR-UNPACK), celles-ci sont utilisées dans l'algorithme de sous-typage. La règle (T-SUB) est intégrée à l'algorithme de sous-typage comme l'implémentation de ( $< : \text{SEL}$ ) le montre ci-dessous.

La fonction principale est `typ_of_internal` qui prend en paramètre un contexte et un terme.

Cette fonction réalise 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 de la règle (LET) 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 (T-SUB) est utilisée dans (ALL-E) pour vérifier que l'argument est un sous-type que la fonction attend.

### Typage des enregistrements récurifs

Dans la syntaxe des termes de DOT, un enregistrement 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 côté développeur, cela n'est pas très pratique. Nous notons  $\nu(x)d$  l'enregistrement récursif  $\nu(x : T)d$  quand  $T$  n'est pas mentionné.

Pour simplifier l'écriture de programmes, nous ajoutons un algorithme pour inférer le type d'une liste de déclarations. Lorsqu'un enregistrement récursif  $\nu(x)d$  est rencontré, 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  $d = \{A = S\}$  et  $x : T$ , nous renvoyons  $T \wedge \{A : S..S\}$ .
- Si  $d = \{a = t\}$  et  $x : T$ , nous utilisons l'algorithme de typage sur  $t$  et le contexte  $\Gamma, x : T$  et notons  $S$  le résultat. Nous renvoyons  $T \wedge \{a : S\}$ .
- Si  $d = d_1 \wedge d_2$  et  $x : T$ , nous appelons récursivement l'algorithme sur  $d_1$  avec  $\Gamma, x : T$ , et notons  $T_1$  le résultat. Nous effectuons alors un second appel sur  $d_2$  avec  $\Gamma, x : T \wedge T_1$ .

Notons  $T$  le type renvoyé par l'algorithme, le type de l'enregistrement  $\nu(x)d$  est alors  $\mu(x : T)$ .

La figure 6 montre un exemple d'enregistrement récursif et le type inféré.

7. Son utilité dans l'algorithme est encore floue.

```

let module Test = struct
  type t = Int.t
  type s = self.t
  let plus = fun(x : self.t, y : self.s) -> Int.plus x y
end;;

(* Type inféré
Test :
sig(self)
  type t = Int.t .. Int.t
  type s = self.t .. self.t
  val plus : forall(x : self.t) forall(y : self.s) Int.t
end

*)

```

Code RML 6 – Exemple de typage d’un terme récursif dont le type n’est pas mentionné.

L’algorithme n’est pas parfait : il échouera par exemple si des champs sont mutuellement récursifs ou si une déclaration dépend d’une autre définie plus haut. La figure 7 montre un exemple sur lequel l’algorithme échoue. Cependant, celui-ci est satisfaisant pour une partie des exemples et permet d’écrire de programmes DOT moins verbeux.

```

let module Test = struct
  type t = Int.t
  type s = self.t
  (* fail dépend de plus, défini plus bas. *)
  let fail = fun(x : self.t, y : self.s) -> self.plus x y
  let plus = fun(x : self.t, y : self.s) -> Int.add x y
end;;

```

Code RML 7 – Exemple de typage d’un terme récursif sur lequel l’algorithme échoue. L’algorithme va tenter de typer la fonction fail, mais elle a besoin du type de la fonction plus, qui ne sera inféré que par après.

## 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  $T$ , de déterminer si  $S$  est un sous-type de  $T$ .

Remarquons que pour une question posée, il est possible d’utiliser plusieurs règles. En effet, pour répondre à  $S_1 \wedge S_2 <: T_1 \wedge T_2$ , nous pouvons utiliser ( $< : \text{AND}$ ), ( $\text{AND-1-} < :$ ) ou ( $\text{AND-2-} < :$ ).

De plus, l’ordre dans le pattern matching influence la réponse. Par exemple,

prenons le cas où  $S_1 = T_1 = \{A : S..T\}$  et  $T_1 = T_2 = \{a : U\}$ . La question  $S_1 \wedge S_2 <: T_1 \wedge T_2$  est alors vraie par réflexivité ou encore par l'arbre de dérivation suivant :

$$\frac{\frac{\Gamma \vdash \{A : S..T\} <: \{A : S..T\}}{\Gamma \vdash \{A : S..T\} \wedge \{a : U\} <: \{A : S..T\}} \quad \frac{\Gamma \vdash \{a : U\} <: \{a : U\}}{\Gamma \vdash \{A : S..T\} \wedge \{a : U\} <: \{a : U\}}}{\Gamma \vdash \{A : S..T\} \wedge \{a : U\} <: \{A : S..T\} \wedge \{a : U\}}$$

où ( $< : \text{-AND}$ ) est utilisé en premier suivi de ( $\text{AND-1-} < :$ ) et ( $\text{AND-2-} < :$ ) sur chaque branche. Cependant, si nous utilisions ( $\text{AND-1-} < :$ ) (ou ( $\text{AND-2-} < :$ )) en premier, nous serions incapables d'arriver au résultat voulu.

Aucun document sur DOT ne décrit actuellement l'implémentation d'un algorithme de sous-typage pour DOT, l'implémentation d'un tel algorithme à partir de règles théoriques n'étant pas simple comme peuvent le montrer [5] et [8] pour d'autres calculs.

Une difficulté supplémentaire de l'implémentation de DOT est que les règles de typage et de sous-typage sont mutuellement dépendantes. Il est naturel et courant d'avoir les règles de typage qui dépendent du sous-typage à travers ( $\text{T-SUB}$ ), mais non l'inverse. Dans DOT, les règles de sous-typage dépendent du typage des variables à travers ( $\text{SEL} < :$ ) ainsi que ( $< : \text{SEL}$ ) et demandent d'implémenter des algorithmes supplémentaires comme **best\_bound**.

Voici, dans l'ordre, comment l'algorithme gère les différents cas en fonction de la structure de  $S$  et  $T$ .

$S <: \text{Top}$  **ou**  $\text{Bottom} <: T$

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 précédemment. Il suffit d'appeler récursivement l'algorithme avec  $L$  et  $L'$  ainsi que  $U$  et  $U'$ . Si les deux appels renvoient vrais, nous renvoyons vrai. Nous utilisons la règle ( $\text{TYP} < : \text{TYP}$ ).

$\{a : S'\} <: \{a : T'\}$

Même cas que précédemment en utilisant la règle ( $\text{FLD} < : \text{FLD}$ ). Il suffit d'appeler l'algorithme avec la question  $S' <: T'$ .

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

Même cas que précédemment en utilisant la règle ( $\text{ALL} < : \text{ALL}$ ). Lors de l'appel récursif sur la question  $T_1 <: T_2$ , nous devons ajouter la variable  $x$  avec le type  $S_2$  dans le contexte comme la règle ( $\text{ALL} < : \text{ALL}$ ) le mentionne.

$x.A <: y.A'$

Les premiers cas posant des difficultés sont ceux des types dépendants, par exemple  $x.A$  et  $y.A'$ <sup>8</sup>. 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 atomes. Nous utilisons donc les fonctions fournies par **AlphaLib** pour le vérifier. Nous appelons cette règle (UN-REFL-TYP).
- Sinon, nous testons en premier (SEL < :). Nous ajoutons en plus (T-SUB) et nous utilisons **best\_bound** pour trouver le type de  $x$  et de  $y$ . La règle

$$\frac{\Gamma \vdash x : \{A : L..U\}}{\Gamma \vdash x.A <: U}$$

devient alors

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash T <: \{A : L..U\} \quad \Gamma \vdash U <: U'}{\Gamma \vdash x.A <: U'}$$

Si nous avons réussi à montrer en utilisant (SEL < :), nous renvoyons cette solution. Sinon, nous essayons (< : SEL). Si cette dernière échoue, cela signifie que  $x.A <: y.A'$  est faux : nous renvoyons donc non. L'utilisation de **best\_bound** et (T-SUB) pour (SEL< :) nous donne

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash \{A : L'..U\} <: T \quad \Gamma \vdash L <: L'}{\Gamma \vdash L <: x.A}$$

## Types rékursifs

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

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

- Si nous avons deux types rékursifs  $\nu(x_1 : S')$  et  $\nu(x_2 : T')$ , nous créons un nouvel atome  $x$  et renommons les variables internes  $x_1$  et  $x_2$  par celui-ci dans  $S'$  et  $T'$ . Un appel rékursif est alors effectué avec  $S'$  et  $T'$  après avoir étendu le contexte avec  $x : S'$ . Nous appelons cette règle (UN-REC).
- Si  $S = \nu(x : S')$  et  $T$  quelconque (resp.  $T = \nu(x : T')$  et  $S$  quelconque), nous ajoutons  $x$  dans le contexte avec le type  $S'$  (resp.  $T'$ ) et nous effectuons un appel rékursif avec  $S'$  et  $T$  (resp.  $S$  et  $T'$ ). Étendre 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 rékursifs qui ne se différencient que par leur variable interne, voir la figure 8.

$x.A <: T$  **ou**  $T <: x.A$

Comme pour le cas  $x.A <: y.A'$ , nous modifions la règle de sous-typage en utilisant (T-SUB) et **best\_bound**.

8. Les cas  $x.A <: U$  et  $U <: x.A$  sont traités ci-dessous.

```

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

```

Code RML 8 – 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 atomes, la question  $Top <: Bottom$  sera posée que nous utilisions (SEL  $< :$ ) ou ( $< :$  SEL).

## Intersections

L'ordre est également important pour les intersections et surtout en présence de types récurifs 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 tester séparément (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 ( $< :SEL$ ) en testant successivement (AND-1- $< :$ ) et (AND-2- $< :$ )

De plus, dans le cas ( $< :AND$ ), nous devons gérer les types récurifs 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 revenir à la question  $\mu(x : S_1 \wedge S_2) <: \mu(x : S_1 \wedge S_2)$ . Si nous utilisons directement ( $< :AND$ ), nous n'arriverions pas à montrer que c'est vrai.

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-UNPACK), (AND-1- $< :$ ), (AND-2- $< :$ ) et (VAR-PACK), nous montrons que  $\mu(x : S_1 \wedge S_2) <: T$  implique  $\mu(x : S_1) \wedge \mu(x : S_2) <: T$ .

$$\begin{array}{c}
 \text{(VAR-PACK)} \frac{\Gamma \vdash \mu(x : S_1 \wedge S_2) <: T}{\Gamma \vdash S_1 \wedge S_2 <: T} \\
 \text{(AND-1-} < : \text{ ET AND-2-} < : \text{)} \frac{\Gamma \vdash S_1 <: T \quad \Gamma \vdash S_2 <: T}{\Gamma \vdash \mu(x : S_1) <: T \quad \Gamma \vdash \mu(x : S_2) <: T} \\
 \text{(VAR-UNPACK)} \frac{\Gamma \vdash \mu(x : S_1) <: T}{\Gamma \vdash \mu(x : S_1) <: T} \quad \text{(VAR-UNPACK)} \frac{\Gamma \vdash \mu(x : S_2) <: T}{\Gamma \vdash \mu(x : S_2) <: T} \\
 \text{(} < : \text{AND)} \frac{\Gamma \vdash \mu(x : S_1) <: T \quad \Gamma \vdash \mu(x : S_2) <: T}{\Gamma \vdash \mu(x : S_1) \wedge \mu(x : S_2) <: T}
 \end{array}$$

La méthode utilisée dans l'algorithme consiste à renommer la variable interne de  $S_1$  et  $S_2$  en utilisant un même atome 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 règle 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  $< : \text{ALL}$ ).
- Si  $S$  est de la forme  $\nu(x : S')$ , (UN-REC) est utilisée.
- Si  $S$  est une intersection, nous pouvons utiliser ( $< : \text{AND}$ ), puis (AND-1- $< :$ ) sur le membre de gauche et (AND-2- $< :$ ) sur le membre de droite comme nous l'avons montré plus haut.
- Si  $S$  est de la forme  $\{A : L..U\}$ , (TYP  $< : \text{TYP}$ ) est utilisée.
- Si  $S$  est de la forme  $\{a : T\}$ , (FLD  $< : \text{FLD}$ ) est utilisée.

## Transitivité

Actuellement, la transitivité n'est pas gérée. En effet, pour la gérer, il faudrait trouver un type  $U$  tel que  $S < : U$  et  $T < : U$ , ce qui n'est pas possible, ou du moins compliqué. Pour contourner ce problème, il est nécessaire de reformuler les règles de sous-typage et d'y introduire explicitement (S-TRANS).

## VII.7 Arbre de dérivation

L'implémentation OCaml des algorithmes de typage et de sous-typage nécessitent un paramètre supplémentaire `history` qui permet de se souvenir des règles utilisées et de pouvoir ainsi reconstruire l'arbre de dérivation. 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.

L'affichage d'un tel arbre de dérivation étant souvent grand, nous ne donnons pas d'exemples dans ce document. Cependant, le lecteur intéressé est invité à tester en ajoutant les annotations correspondantes.

## VII.8 Sucres syntaxiques

La syntaxe de base de DOT n'est pas très élégante ni 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.

## Curryfication 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) -> 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`, le terme DOT correspondant étant  $\nu(x : T)d$ . 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`. Les noms de 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.

Comme les champs ne sont pas mutuellement dépendants et que les enregistrements ne peuvent pas contenir de types, l'algorithme de typage pour les modules présentés ci-dessus est utile pour utiliser une syntaxe proche d'OCaml sans devoir mentionner de types.

## 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. Cela se résume par les règles suivantes :

$$\begin{aligned} x \ t &\rightarrow \text{let } y = t \text{ in } x \ y \\ t \ y &\rightarrow \text{let } x = t \text{ in } x \ y \end{aligned}$$

Un point important<sup>9</sup> est qu'il faut éviter de réaliser un binding local d'une variable car cela pourrait provoquer des problèmes d'échappement. Cela signifie que DOT n'est pas stable par insertion de `let` variable-variable. Un exemple concret est donné par la figure 9.

Pour résoudre ce problème, nous pourrions, dans le cas d'un binding local `let x = y in u`, donner le type  $(= y)$  à la variable  $x$ . Lorsque nous rencontrons alors la variable  $x$ , nous utilisons le type de  $y$ . Cette méthode est utilisée dans le langage Mezzo[13] avec le type singleton.

## Applications de fonctions à plusieurs paramètres

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

- $f \ x \ y$  est interprété comme  $\text{let } f_x = (f \ x) \text{ in } (f_x \ y)$ .
- $f \ x \ y \ z$  est interprété comme  $\text{let } f_x = (f \ x) \text{ in } \text{let } f_y = (f_x \ y) \text{ in } (f_y \ z)$ .

---

9. Ce n'est pas mentionné dans les documents sur DOT.



```

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, x : Int.t) -> m.x;;

f M 4;;

```

Code RML 9 – 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 réalisés pour chaque terme, une liaison locale du module *M* est créée avec la variable *n* par exemple et le type de l’expression est *n.t*.

## 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, la sémantique des termes peut être laissée de côté. 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*. Dans l’implémentation, nous inférons le type de *t* grâce à l’algorithme de typage et vérifions que ce dernier est sous-type de *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

Plus d’exemples peuvent être trouvés sur la page du projet. Des exemples plus simples comme plus compliqués sont disponibles dans les dossiers `test` et dans la librairie standard `stdlib` qui comporte des booléens, des options, des types sommes ou encore des conditions (en utilisant le codage à la Church). Nous ne présentons ici que certains exemples montrant que les modules sont

des citoyens de première classe grâce à l'unification des enregistrements et des modules.

Reprenons l'exemple donné en introduction. En RML, le module `Point2D` avec des entiers peut être défini comme sur la figure 10. Il est important de noter que dans la plupart des exemples, les algorithmes de typage et de sous-typage sont appelés plusieurs fois, autant que l'algorithme de typage des modules ou encore les différentes variantes de `best_bound`. Les différents sucres syntaxiques sont également utilisés implicitement.

```
let module Point2D = struct(point)
  type t = { x : Int.t ; y : Int.t }
  let add = fun(p1 : point.t, p2 : point.t) ->
    { x = Int.plus p1.x p2.x ; y = Int.plus p1.y p2.y }
end;;

(*
Sortie de l'algorithme et signature.

Point2D :
sig(point)
  type t = sig('self)
    val x : Int.t
    val y : Int.t
  end .. sig('self)
    val x : Int.t
    val y : Int.t
  end
  val add : forall(p : point.t) forall(p : point.t) sig('self)
    val x : Int.t
    val y : Int.t
  end
end
*)
```

Code RML 10 – Point2D en RML. Nous voyons également comment des enregistrements peuvent être définis. Nous remarquons aussi que le type des enregistrements est le même que celui d'un module.

Nous pouvons définir le foncteur `MakePoint2D` avec le terme `fun` comme une autre fonction comme le montre la figure 11.

La figure 12 montre que le type du foncteur est le même que celui d'une fonction qui prend un module en paramètre et retourne un module.

Nous pouvons également définir des listes paramétrées par un type en utilisant un foncteur comme le montre la figure 13 et créer une liste d'entiers en utilisant `List Int`.

Cependant, cette implémentation de liste n'est pas très pratique car si nous voulons une liste d'entiers, nous devons créer un module intermédiaire. La figure 14 propose une autre implémentation de listes polymorphes qui de plus sont covariantes (grâce à `type t <: self.t`). Le mot clef `with` dénote une in-

```

let module MakePoint2D =
  fun(typ : sig
    type t
    val plus : self.t -> self.t -> self.t
  end) ->
  struct(point)
    type t = { x : typ.t ; y : typ.t }
    let add = fun(p1 : point.t, p2 : point.t) ->
      let x' = typ.plus p1.x p2.x in
      let y' = typ.plus p1.y p2.y in
      { x = x' ; y = y' }
  end;;

```

Code RML 11 – MakePoint2D en RML. Le paramètre de la fonction est un module qui contient au moins un champ `t` et une fonction `plus`.

tersection<sup>10</sup>. Nous pouvons alors utiliser `List.list with type t = Int` pour obtenir une liste d'entiers. Malheureusement, l'algorithme actuel ne supporte pas cette implémentation et provoque un stack overflow car le champ `tail` repose la question du sous-typage de la liste en boucle.<sup>11</sup>

## VII.11 Travail futur

Bien que l'implémentation actuelle donne des résultats satisfaisants sur différents cas, diverses améliorations peuvent être effectuées. Une liste peut être trouvée, en anglais, sur la page du projet[20]. En 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. De plus, comme le problème du sous-typage est indécidable, il existe des questions qui ne disposent pas de réponse.
- 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 [18].
- Un interpréteur interactif.
- Améliorer l'algorithme d'inférence de type pour les modules. En effet, comme nous l'avons vu, celui-ci est relativement naïf 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 typage et de sous-typage.
- 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

10. Comme en OCaml dans les foncteurs.

11. Une solution est en cours de développement pour supporter les questions qui ont déjà été posées.

```

(* Signature de MakePoint2D
forall(typ : sig(self)
  type t = Nothing .. Any
  val plus : self.t -> self.t -> self.t
end) sig(point)
  type t = sig('self)
    val x : typ.t
    val y : typ.t
  end .. sig('self)
    val x : typ.t
    val y : typ.t
  end
  val add : forall(p1 : point.t) forall(p2 : point.t) sig('self)
    val x : typ.t
    val y : typ.t
  end
end
*)

```

Code RML 12 – Signature de MakePoint2D en RML.

de type passe par la résolution d'équations et nécessite de travailler avec deux arbres différents.

- Un enregistrement récursif peut posséder plusieurs déclarations grâce à l'intersection. Il serait plus efficace d'implémenter cela grâce à une map qui associe la déclaration à un label. Cette solution n'a pas été implémentée directement pour rester fidèle à la grammaire de DOT.

## Gestion des questions déjà posées

Une solution au problème du stack overflow dans l'algorithme de sous-typage est de se souvenir des types  $(S, T)$  des questions qui ont déjà été posées. Nous ajoutons comme paramètre à l'algorithme la liste des questions déjà posées.

Initialement, cette liste est vide. Ensuite, à chaque appel de la forme  $\Gamma \vdash S <: T$ , nous regardons si le couple  $(S, T)$  est dans la liste. Si c'est le cas, nous retournons oui, sinon nous ajoutons le couple  $(S, T)$  et appliquons l'algorithme de sous-typage défini précédemment.

Cependant, bien que l'idée semble simple à implémenter, l'implémentation pose problème à cause de la gestion des variables internes. Plus d'informations et un début d'implémentation sont disponibles dans la branche `already_asked_2` du dépôt du projet.

```

let module List = fun(elem : sig type t end) -> struct(list)
  type t = sig
    val head : Unit.t -> elem.t
    val tail : Unit.t -> list.t
    val size : Int.t
    val is_empty : Bool.t
  end
  let empty : list.t = struct
    let head = fun (x : Unit.t) -> Unimplemented (* Must fail... *)
    let tail = fun (x : Unit.t) -> Unimplemented (* Must fail... *)
    let size = Int.zero
    let is_empty = Bool.true
  end
  let cons : elem.t -> list.t -> list.t =
    fun(x : elem.t, l : list.t) -> struct
      let head = fun(x' : Unit.t) -> x
      let tail = fun(l' : Unit.t) -> l
      let size = Int.succ l.size
      let is_empty = Bool.false
    end
end;;

```

Code RML 13 – Une implémentation de listes polymorphes en RML en utilisant un foncteur. Le type `t` représente le type liste. Le module `elem` est le type des éléments de la liste. Remarquons que dans cette implémentation, une liste ne peut contenir que des éléments du même type, ce dernier étant fixé par le paramètre du foncteur.

```

let module List = struct(sci)
  type list = sig
    type t
    val is_empty : Bool.t
    val head : self.t
    val tail : sci.list with type t <: self.t
  end
  let nil : sci.list with type t = Nothing = struct
    type t = Nothing
    let is_empty = Bool.true
    let head = Unimplemented (* Must fail *)
    let tail = Unimplemented (* Must fail *)
  end
  let cons = fun(elem : sig
    type t
  end,
    head : elem.t,
    tail : sci.list with type t <: elem.t) ->
  struct
    type t = elem.t
    let is_empty = Bool.false
    let head = head
    let tail = tail
  end
end
end;;

```

Code RML 14 – Une implémentation de listes polymorphes en RML en utilisant le mot clef `with`. Contrairement à l’implémentation de la figure 13, les éléments de la liste peuvent avoir un type différent.

# 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  $\lambda$ -calcul non typé et le  $\lambda$ -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 Système  $F_{<}$ .

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 infinies. 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 dépendants dans le langage ne facilite pas l'implémentation. Pour finir, il nous a été nécessaire d'écrire des algorithmes secondaires comme `best_bound`, non décrits dans les différents articles discutant de DOT.

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[3]. Ce dernier catégorise les types en genres[12] 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 semblent encore plus complexes que DOT.





# Bibliographie

- [1] ADMIN, N. Thesis - Dependent Object Types. <https://infoscience.epfl.ch/record/223518>, 2016.
- [2] AMIN, N., AND ROMPF, T. Type Soundness Proofs with Definitional Interpreters. *POPL* (2017).
- [3] ANDREAS, R. 1ML. <https://people.mpi-sws.org/~rossberg/1ml/>, 2015-2016.
- [4] GOUBAULT-LARRECQ, J. Cours intitulé  $\lambda$ -calcul et langages fonctionnels. <http://www.lsv.fr/~goubault/Lambda/lambda.pdf>.
- [5] PIERCE, B. C. *TAPL - Chapter 16 - Metatheory of subtyping*. The MIT Press, 2002.
- [6] PIERCE, B. C. *TAPL - Chapter 23 - System F*. The MIT Press, 2002.
- [7] PIERCE, B. C. *TAPL - Chapter 26 - Bounded quantification*. The MIT Press, 2002.
- [8] PIERCE, B. C. *TAPL - Chapter 28 - Metatheory of bounded quantification*. The MIT Press, 2002.
- [9] PIERCE, B. C. *TAPL - Chapter 5 - The Untyped Lambda-Calculus*. The MIT Press, 2002.
- [10] PIERCE, B. C. *TAPL - Chapter 9 - The Simply Typed Lambda-Calculus*. The MIT Press, 2002.
- [11] PIERCE, B. C. *TAPL - Part IV - Recursive Types*. The MIT Press, 2002.
- [12] PIERCE, B. C. *TAPL - Partie 4 - Higher Order Systems*. The MIT Press, 2002.
- [13] PITTEr, F., AND PROTZENKO, J. Mezzo. <http://protz.github.io/mezzo/>.
- [14] POTTIER, F. AlphaLib. <https://gitlab.inria.fr/fpottier/alphaLib>.
- [15] POTTIER, F. Visitors. <https://gitlab.inria.fr/fpottier/visitors>.
- [16] POTTIER, F. Visitors Unchained. <http://gallium.inria.fr/~fpottier/publis/fpottier-visitors-unchained.pdf>, 2017.
- [17] ROMPF, T., AND AMIN, N. Type Soundness for Dependent Object Types (DOT). *OOPSLA* (2016).
- [18] ROMPF, T., AMIN, N., GRÜTTER, S., ODESKY, M., AND STUCKI, S. The Essence of Dependent Object Types. *WF* (2016).
- [19] WILLEMS, D., AND POTTIER, F. RML - ML modules and functors as first-class citizens. <https://github.com/dannywillems/RML>.
- [20] WILLEMS, D., AND POTTIER, F. RML - ML modules and functors as first-class citizens - Issues. <https://github.com/dannywillems/RML/issues>.