Introduction à la théorie des langages de programmation

David Baelde, ENS Rennes

11 septembre 2024 (travail en cours)

Table des matières

Ce document correspond à la partie théorique du cours PRG1 donné à l'ENS Rennes dans le cadre du L3 SIF. Pour l'instant il s'agit d'un résumé, rappelant les principales définitions et résultats. Il devrait s'enrichir en de vraies notes de cours alors que celui-ci se stabilise. N'hésitez pas à remonter vos remarques sur les erreurs, imprécisions ou zones d'ombre que vous rencontrerez. Je remercie les étudiants des promotions 2021–2023 pour leurs retours lors des séances des premières éditions du cours.

Chapitre 1

Approche équationnelle pour un fragment pur d'OCaml

Dans ce chapitre introductif, nous étudions la possibilité de raisonner équationnellement, comme on le fait tout le temps en mathématiques, pour prouver la correction de programmes OCaml. On reste à un niveau semi-formel, et on abandonnera face à certains obstacles, mais ce n'est pas grave : l'objectif est d'introduire une démarche, d'identifier des obstacles, et de commencer à jouer avec certains concepts qui reviendront un peu plus loin dans le cours, cette fois dans un cadre pleinement formel.

Notations. Les expressions (aussi appelées programmes) sont notées au moyen de la lettre e. Pour nommer des expression complexes et simplifier l'écriture lors d'un calul ou raisonnement, on utilisera toujours une majuscule, par exemple $Fact$. On utilise une police différenciée pour noter les constantes (par exemple, constantes entières) et les variables : n, m, p pour les constantes ; $x, y, fact$ pour les variables. Les constructeurs seront notés None, Nil, etc.

On ne considère que des expressions typées, mais sans expliciter les règles de typage, pour ne pas alourdir la discussion.

1.1 Objectif

On cherche à définir une relation *entre* programmes, qu'on notera $\equiv$ et appellera *équivalence* pour la distinguer de l'égalité $=$ qu'on écrit *dans* les programmes. L'équivalence devra satisfaire deux conditions :

- Si un programme OCaml e s'évalue en une valeur v, alors $e \equiv v$. Sinon, on ne pourra pas utiliser notre égalité pour prouver la correction de programmes.
- Si deux programmes sont déclarés égaux, i.e. $e \equiv e'$, alors on peut remplacer l'un par l'autre dans tout contexte sans changer le résultat de l'évaluation : si $p[x := e]$ s'évalue en v alors $p[x := e']$ aussi. Cette condition demande simplement que notre relation $\equiv$ permette le raisonnement équationnel.

Remarque 1. *Une conséquence de ces deux conditions est qu'on aura $e \equiv e'$ quand les deux expressions s'évaluent en la même valeur. On ne demande pas la réciproque.*

Remarque 2. *Nos deux conditions ne définissent pas $\equiv$ de façon unique. Rien ne dit si deux fonctions qui ont le même graphe, e.g. $e :=$ **fun** $x \mapsto x+x$ et $e' :=$ **fun** $x \mapsto 2\times x$, sont considérées comme des valeurs égales. Les conditions ne forcent donc pas $e \equiv e'$. Cette égalité semble cependant désirable pour faciliter des preuves de programmes.*

On s'aperçoit rapidement que ces deux conditions mènent à des contradictions si l'on essaie d'inclure les états mutables (e.g. références) dans notre démarche. Même problème avoir les fonctions d'entrée/sortie.

Les exceptions ne mènent pas à de telles contradictions. On remarquera simplement que l'équivalence doit nécessairement distinguer un programme qui termine d'un programme qui lève une exception ; de même pour deux programmes qui lèvent des exceptions distinctes. Par contre, les exceptions sont incompatibles avec l'équation $(\textbf{fun}\ x \mapsto e)\ e' \equiv e$, qui semble désirable. On se simplifiera la vie en les laissant de côté.

On se restreint donc dans ce chapitre préliminaire à un fragment "pur" d'OCaml. On n'utilisera qu'un petit nombre de constructions d'OCaml : définition et application de fonction, opérateur **let** et sa variante récursive, filtrage, ainsi que les types de base **int** et **bool** et les opérations primitives sur ceux-ci.

Exemple 1.1. *La factorielle naïve est définie ainsi :*

$$Fact := \big(\textbf{let rec}\ fact = \textbf{fun}\ x \mapsto \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x \times fact\ (x-1)\ \textbf{in}\ fact\big)$$

On voudrait pouvoir démontrer, par récurrence sur $\mathsf{n} \in \mathbb{N}$*, que* $Fact\ \mathsf{n} \equiv \mathsf{m}$ *où* $\mathsf{m} = \mathsf{n}!$ *(cette dernière égalité portant sur les entiers mathématiques et non les expressions).*

1.2 Théorie équationnelle

On définit $\equiv$ comme la plus petite congruence [1] sur les expressions satisfaisant les équations suivantes :

$$((\textbf{fun}\ x \mapsto e)\ e') \quad \equiv \quad e[x := e'] \tag{1.1}$$

$$(\textbf{match}\ \mathsf{C}\ e\ \textbf{with}\ \ldots \mid \mathsf{C}\ x \mapsto e' \mid \ldots) \quad \equiv \quad e'[x := e] \tag{1.2}$$

$$(\textbf{let}\ x = e\ \textbf{in}\ e') \quad \equiv \quad e'[x := e] \tag{1.3}$$

On ajoute à cette liste des règles raisonnables non précisées pour le **if then else**, ainsi que les opérations primitives comme l'addition ou l'égalité. On ne considère ici que des expressions bien typées. On peut s'autoriser les sucres syntaxiques habituels, par exemple écrire $\textbf{fun}\ x\ y \mapsto e$ pour $\textbf{fun}\ x \mapsto \textbf{fun}\ y \mapsto e$, ou encore $e\ e'\ e''$ pour $(e\ e')\ e''$… mais il faut bien veiller à les décomposer pour mener les calculs précis sur les expressions.

Exemple 1.2. *On a* $1 + 1 \equiv 2$*, mais aussi* $1 + 3 \equiv 2 + 2$*, ou encore* $(\textbf{let}\ x = 2\ \textbf{in}\ x + x) \equiv 4$.

Pour la construction **let rec**, il n'y a pas *une* équation simple qu'on souhaiterait inclure à notre définition. Comme on s'intéresse en fait seulement aux définitions récursives de fonctions, il est avantageux de considérer une construction fictive $\textbf{fixfun}\ f\ x \mapsto e$, qu'on peut lire comme $\textbf{let rec}\ f\ x = e\ \textbf{in}\ f$, et de l'équiper de l'équation naturelle suivante :

$$\textbf{fixfun}\ fx \mapsto e \quad \equiv \quad \textbf{fun}\ x \mapsto e[f := \textbf{fixfun}\ fx \mapsto e] \tag{1.4}$$

Exercice 1.1. *On se donne la fonction suivante, de type* $\textbf{int} \to (\textbf{int} \to \textbf{int})$ [2] *:*

$$Fact' := \big(\textbf{fixfun}\ f'\ a \mapsto \textbf{fun}\ x \mapsto \textbf{if}\ x = 0\ \textbf{then}\ a\ \textbf{else}\ (f'\ (x \times a))\ (x-1)\big)$$

Démontrer qu'on a, pour tous $\mathsf{a}, \mathsf{n} \in \mathbb{N}$*,* $((Fact'\ \mathsf{a})\ \mathsf{n}) \equiv \mathsf{m}$ *où* m *est la constante entière* $\mathsf{a} \times \mathsf{n}!$*. En bonus, proposer une caractérisation des fonctions récursives terminales en termes de forme des calculs dans la théorie équationnelle.*

1. Une congruence est une relation d'équivalence qui "passe au contexte" : pour tous p, e, e' tels que $e \equiv e'$, on a $p[x := e] \equiv p[x := e']$.

2. On explicite ici des parenthèses qui sont inutiles en OCaml du fait des règles de précédence du langage ; il faut se familiariser rapidement avec cela car on y reviendra peu. La même chose se passe, dans cet exemple, pour les applications de $Fact'$ et f'.

Remarque 3. *Notre équivalence en "fait trop" par rapport à OCaml. Posons* $Div :=$ **let rec** $f\ x = f\ x$ **in** $f\ x$*. Ce programme ne n'évalue en aucune valeur ; on dit qu'il ne termine pas, ou qu'il* diverge*. De même, en OCaml,* $(\textbf{fun}\ x \mapsto 1)\ Div$ *ne termine pas. Pourtant, par définition, on a* $(\textbf{fun}\ x \mapsto 1)\ Div \equiv 1$*. On pourrait corriger ce problème, mais ce serait lourd. D'une certaine façon, l'équivalence qu'on a définit ainsi correspondrait mieux à une version paresseuse d'OCaml.*

Exercice 1.2. *Programmez dans notre fragment pur d'OCaml une fonction de renversement de liste, et démontrez sa correction. Essayez de le faire sans utiliser d'opération sur les listes autre que le constructeur* _::_*, en évitant notamment la concaténation* _@_*.*

Exercice 1.3. *On se donne un type d'arbres binaires :*

$$\textbf{type}\ \text{`}a\ \mathsf{t} = \mathsf{Leaf}\ \textbf{of}\ \text{`}a \mid \mathsf{Node}\ \textbf{of}\ \text{`}at \times \text{`}a\ \mathsf{t}$$

Implémenter une fonction $Parcours$: $\text{`}a\ \mathsf{t} \to \text{`}a\ \mathsf{list}$ *renvoyant les valeurs apparaissant aux feuilles d'un arbre selon un parcours en profondeur de gauche à droite. Bonus : pouvez-vous rendre cette fonction récursive terminale ?*

Exercice 1.4. *Même question mais avec un parcours en largeur.*

1.3 Substitutions et liaisons de variables

Si l'opération $e[x := e']$ a un sens intuitif simple, elle recèle en réalité plusieurs pièges. Deux problèmes sont causés par les variables *liées* :

- Il ne faut pas substituer une variable liée.
  Par exemple, $(\textbf{fun}\ x \mapsto x)[x := 1]$ est toujours $(\textbf{fun}\ x \mapsto x)$. Sinon on aurait $(\textbf{fun}\ x \mapsto (\textbf{fun}\ x \mapsto x)\ 0)\ 1 \equiv 1$.
- Il ne faut pas qu'une variable libre dans l'expression à substituer soit *capturée* par un lieur de l'expression dans laquelle on substitue.
  Par exemple, $(\textbf{fun}\ x \mapsto y)[y := x]$ n'est pas $(\textbf{fun}\ x \mapsto x)$. Pour pouvoir exprimer cette substitution il faut d'abord renommer la variable liée x. On obtient alors $(\textbf{fun}\ z \mapsto x)$.

Nous reviendrons de façon précise sur ce point dans la suite du cours.

Chapitre 2

Relation d'évaluation pour un fragment pur d'OCaml

Pour cette deuxième et dernière séance préliminaire, on laisse le raisonnement équationnel de côté, et on s'attache à définir formellement l'évaluation OCaml, sur notre fragment pur. Cela nous permet d'introduire dans les grandes lignes la sémantique dite "à grands pas", et sa proximité avec l'implémentation d'un interprète. On évoquera aussi les difficultés avec la substitution, et on introduira la notion de clôture.

Relation d'évaluation

On cherche à se rapprocher du mode de calcul effectif d'OCaml. En effet, la théorie équationnelle peut suggérer des calculs ne correspondant pas à l'évaluation des programmes. À l'extrême, elle peut donner un résultat pour des programmes qui ne terminent pas en OCaml. Par exemple :

$$(\textbf{fun}\ x \mapsto 0)\ \big((\textbf{fixfun}\ f\ x \mapsto f\ x)\ ()\big) \equiv 0.$$

Nous allons donc définir une relation binaire $\Downarrow$ sur les expressions. Intuitivement, $e \Downarrow v$ devra signifier que l'expression e s'évalue en la *valeur* v. Cette relation devra satisfaire les conditions données en Figure 2.1.

Plusieurs relations satisfont ces conditions. En particulier, la relation pleine les satisfait. En fait, la relation d'évaluation qui nous intéresse est *la plus petite*[1] relation satisfaisant ces conditions. On définit habituellement les conditions comme des règles d'inférence, et les éléments en relation sont ceux qui peuvent être dérivés par un arbre fini composé de ces règles.

Exemple 2.1. *On a* $(\textbf{let}\ f = \textbf{fun}\ x \mapsto x\ \textbf{in}\ f)\ 0 \Downarrow 0$ *comme en témoigne l'arbre suivant, où* $\mathcal{I}d := \textbf{fun}\ x \mapsto x$ *:*

$$\dfrac{\dfrac{}{0 \Downarrow 0} \quad \dfrac{\dfrac{\overline{\textbf{fun}\ x \mapsto x \Downarrow \mathcal{I}d} \quad \overline{f[f := \mathcal{I}d] \Downarrow \mathcal{I}d}}{\textbf{let}\ f = \textbf{fun}\ x \mapsto x\ \textbf{in}\ f \Downarrow \mathcal{I}d} \quad \dfrac{}{x[x := 0] \Downarrow 0}}{(\textbf{let}\ f = \textbf{fun}\ x \mapsto x\ \textbf{in}\ f)\ 0 \Downarrow 0}$$

On pourrait démontrer, par induction sur les arbres de dérivation, que :

- L'évaluation est une fonction : $e \Downarrow v$ et $e \Downarrow v'$ entraîne $v = v'$.
- Les valeurs (i.e. les objets à droite du $\Downarrow$) sont forcément des fonctions, des constantes, ou bien un constructeur appliqué à une valeur.
- $e \Downarrow v$ implique $e \equiv v$ (on a vu que la réciproque n'est pas vraie).

1. C'est bien défini pour les conditions qu'on a posé. Pourquoi ?

$$
\begin{aligned}
n &\Downarrow n \text{ pour } n \in \mathbb{N} \\
(\textbf{fun } x \mapsto e) &\Downarrow (\textbf{fun } x \mapsto e) \\
(\textbf{fixfun } f\ x \mapsto e) &\Downarrow (\textbf{fixfun } f\ x \mapsto e) \\
\mathbf{C}\ e \Downarrow \mathbf{C}\ v &\quad \textbf{si} \quad e \Downarrow v \\
\\
(e\ e') \Downarrow v'' &\quad \textbf{si} \quad e' \Downarrow v',\ e \Downarrow (\textbf{fun } x \mapsto e'') \textbf{ et } e''[x := v'] \Downarrow v'' \\
(e\ e') \Downarrow v'' &\quad \textbf{si} \quad e' \Downarrow v',\ e \Downarrow (\textbf{fixfun } f\ x \mapsto e'') \\
&\qquad\qquad \textbf{et } e''[x := v'][f := \textbf{fixfun } f\ x \mapsto e''] \Downarrow v'' \\
(\textbf{let } x = e \textbf{ in } e') \Downarrow v' &\quad \textbf{si} \quad e \Downarrow v \textbf{ et } e'[x := v] \Downarrow v' \\
(\textbf{if } e \textbf{ then } e' \textbf{ else } e'') \Downarrow v' &\quad \textbf{si} \quad e \Downarrow \textbf{true et } e' \Downarrow v' \\
(\textbf{if } e \textbf{ then } e' \textbf{ else } e'') \Downarrow v'' &\quad \textbf{si} \quad e \Downarrow \textbf{false et } e'' \Downarrow v'' \\
e + e' \Downarrow m &\quad \textbf{si} \quad e \Downarrow n,\ e' \Downarrow n',\ m = n + n' \quad (n, n', m \in \mathbb{N}) \\
e = e' \Downarrow \textbf{true} &\quad \textbf{si} \quad e \Downarrow n,\ e' \Downarrow n \quad (n \in \mathbb{N}) \\
e = e' \Downarrow \textbf{false} &\quad \textbf{si} \quad e \Downarrow n,\ e' \Downarrow n', n \neq n' \quad (n, n' \in \mathbb{N}) \\
(\textbf{match } e \textbf{ with } \ldots \mid \mathbf{C}\ x \to e' \mid \ldots) \Downarrow v' &\quad \textbf{si} \quad e \Downarrow \mathbf{C}\ v \textbf{ et } e'[x := v] \Downarrow v'
\end{aligned}
$$

FIGURE 2.1 – Évaluation de notre fragment pur d'OCaml, avec substitutions

Environnements

La relation d'évaluation est fidèle à OCaml, à haut niveau : si $e \Downarrow v$ alors OCaml évalue e en v — sur le principe on aurait la réciproque aussi, moyennant l'ajout de suffisamment de clauses dans la définition de $\Downarrow$. Pour aller plus loin, on peut voir notre définition de $\Downarrow$ comme le schéma d'un interprète abstrait. Mais là, il y a un écart net avec ce qu'on souhaiterait en pratique : l'utilisation des substitutions induit un coût déraisonnable par rapport au du coût réel d'évaluation des programmes. En effet, l'évaluation de $\big((\textbf{fun } x \mapsto \textbf{if true then } 0 \textbf{ else } e)\ 1\big)$ donne la valeur 0 en un temps indépendant de la taille de e, ce qui n'est pas le cas si l'on doit calculer $e[x := 1]$. Pour résoudre cela, nous allons introduire un modèle d'évaluation plus réaliste, reposant sur la notion d'*environnement*.

Un environnement est une fonction (de domaine fini) des variables dans les valeurs. Intuitivement, l'environnement va donner leurs valeurs aux variables, ce qui remplacera la substitution des variables par leurs valeurs. Malheureusement, cela ne suffit pas : si l'on évalue $\big((\textbf{fun } x \mapsto \textbf{fun } y \mapsto x)\ 1\big)$ en $\textbf{fun } y \mapsto 1$, cela serait correct mais passerait de nouveau par une substitution de y par x. Pour éviter cela, nous allons représenter les valeurs fonctionnelles par une expression (construite via **fun** ou **fixfun**) attachée à un environnement (qui représente essentiellement la substitution qu'on n'a pas appliquée à la fonction).

Une valeur est donc une constante, un constructeur appliqué à une valeur, ou une *clôture*, i.e. une fonction attachée à un environnement :

$$v ::= n \mid \mathbf{C}\ v \mid \langle E, \textbf{fun } x \mapsto e \rangle \mid \langle E, \textbf{fixfun } f\ x \mapsto e \rangle$$

La relation d'évalution avec environnements, notée $E \vdash e \Downarrow v$, est définie comme la plus petite relation satisfaisant les règles de la Figure 5.5. On y note $E + (x \mapsto v)$ l'environnement qui envoie x sur v et coïncide avec E sur les autres variables.

Exercice 2.1. *Proposer une implémentation des environnements comme des fonctions OCaml, puis comme des listes d'association. En quel sens ces deux implémentations sont elles équivalentes ?*

$$\frac{}{E \vdash n \Downarrow n} \qquad \frac{}{E \vdash x \Downarrow E(x)} \qquad \frac{}{E \vdash (\textbf{fun}\ x \mapsto M) \Downarrow \langle E, \textbf{fun}\ x \mapsto M \rangle}$$

$$\frac{E \vdash e \Downarrow \langle E'', \textbf{fun}\ x \mapsto e'' \rangle \quad E \vdash e' \Downarrow v' \quad E'' + (x \mapsto v') \vdash e'' \Downarrow v''}{E \vdash (e\ e') \Downarrow v''}$$

$$\frac{E \vdash e \Downarrow v \quad E + (x \mapsto v) \vdash e' \Downarrow v'}{E \vdash \textbf{let}\ x = e\ \textbf{in}\ e' \Downarrow v'}$$

$$\frac{}{E \vdash (\textbf{fixfun}\ f\ x \mapsto e) \Downarrow \langle E, \textbf{fixfun}\ f\ x \mapsto e \rangle}$$

$$\frac{\begin{array}{l} E \vdash e \Downarrow \langle E'', \textbf{fixfun}\ f\ x \mapsto e'' \rangle \\ E \vdash e' \Downarrow v' \\ E'' + (x \mapsto v') + (f \mapsto \langle E'', \textbf{fixfun}\ f\ x \mapsto M' \rangle) \vdash e'' \Downarrow v'' \end{array}}{E \vdash e\ e' \Downarrow v''}$$

Les règles pour l'addition, l'égalité, la conditionnelle, manquent, mais l'environnement n'y joue pas un rôle crucial.

FIGURE 2.2 – Évaluation de notre fragment pur d'OCaml, avec clôtures

Exercice 2.2. *Proposer un énoncé reliant l'évaluation avec environnements et la théorie équationnelle. Il va falloir traduire l'environnement en une substitution.*

Chapitre 3

Lambda-calcul

Dans ce chapitre on introduit le cœur des langages de programmation fonctionnels : le λ-calcul. Il s'agit d'un langage minimal, où le seul concept est celui de fonction, ce qui peut être déstabilisant. Ce langage sera étendu dans les chapitres suivants.

3.1 Syntaxe

On se donne un ensemble infini $\mathcal{X}$ dont les éléments seront appelés *variables* et notés x, y, z.

Définition 3.1 (Termes du λ-calcul). *Les* termes *du λ-calcul (notés M, N, etc.) sont donnés par la grammaire suivante :*

$$M, N ::= x \in \mathcal{X} \mid M\ N \mid \lambda x.M$$

Autrement dit, l'ensemble des termes est le plus petit ensemble tel que :
- les variables sont des termes ;
- si l'on a deux termes M et N, alors l'application de l'un à l'autre, notée $M\ N$ est un terme ;
- enfin l'on peut former à partir d'une variable x et d'un terme M un nouveau terme $\lambda x.M$ qu'on appelle une abstraction.

Les termes doivent être vus comme des arbres avec des variables aux feuilles et des noeuds binaires décorés par les constructions d'application et d'abstraction. Quand on raisonne par induction sur un terme, il s'agit d'une induction sur cet arbre ou, de façon équivalente, sur la définition inductive de l'ensemble des termes donnée ci-dessus. On pourrait parfois avoir envie de raisonner par induction sur la hauteur ou la taille d'un terme (toujours vu comme un arbre) mais cela ne nous arrivera pas ici.

L'intention derrière ces notations est que $\lambda x.M$ représente la fonction qui à x associe M. C'est ce qu'on note (fun $x \to M$) en OCaml. L'application $M\ N$ est ce qu'on note de la même façon en OCaml : le passage d'un argument N à une fonction M. Cette intention va être exprimée formellement dans la prochaine section, où l'on donnera une sémantique à nos termes.

Exemple 3.1. *Le terme $\lambda x.x$ représente, intuitivement, la fonction identité. Le terme $\lambda x.\lambda y.x$ est une fonction qui prend deux arguments et renvoie le premier. Le terme $\lambda x.\lambda y.(x\ y)$ est une fonction qui prend deux arguments et passe le second en argument du premier.*

L'application est associative à droite : on écrira $(M\ N_1\ N_2)$ pour $((M\ N_1)\ N_2)$. On lui donne aussi une précédence plus élevée qu'à l'abstraction : ainsi $\lambda x.\lambda y.(x\ y)$ sera simplement noté $(\lambda x.\lambda y.\ x\ y)$.

3.2 Substitution, variables libres et liées

L'opération de base sur les variables est la substitution, c'est à dire le remplacement dans un terme M d'une variable x par un terme N. Cela sera noté $M[x := N]$. Notre objectif est de définir cette opération, ce qui pose des problèmes surprenants.

Exemple 3.2. *Intuitivement, $(\lambda x.x)$ est la fonction identité. Avec une définition naïve de la substitution, on pourrait se retrouver à avoir $(\lambda x.x)[x := y]$ égal à $(\lambda x.y)$, la fonction constante égale à y : ce serait une catastrophe. On ne veut pas non plus avoir $(\lambda x.y)[y := x] = (\lambda x.x)$.*

Pour éviter ces problèmes, il nous faut introduire le concept de variables libres et liées. Une même variable peut avoir, dans un même terme, des occurrences libres et des occurrences liées, c'est à dire capturées par un lambda.

Exemple 3.3. *Dans $((\lambda x.x\ y)\ x)$ la variable x a deux occurrences en tant que terme – on ignore son utilisation dans le lieur λx. La première est liée par le lambda, la seconde est libre. La seule occurrence de y est libre – on suppose en effet que les variables x et y sont distinctes, donc le λx ne capture pas l'occurrence de y.*

On définit formellement les ensembles des variables libres et liées d'un terme : on dit qu'une variable est libre dans un terme quand elle a une occurrence libre dans ce terme, et liée quand elle apparait en position de lieur. On utilise ici des notations issues de la terminologie anglaise : on parle de *free* et *bound* variable.

Définition 3.2 (Variables libres)**.** *On définit* $\mathsf{fv}(M)$ *inductivement :*

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(M\ N) &= \mathsf{fv}(M) \cup \mathsf{fv}(N) \\ \mathsf{fv}(\lambda x.M) &= \mathsf{fv}(M) \setminus \{x\} \end{aligned}$$

Définition 3.3 (Variables liées)**.** *On définit* $\mathsf{bv}(M)$ *inductivement :*

$$\begin{aligned} \mathsf{bv}(x) &= \emptyset \\ \mathsf{bv}(M\ N) &= \mathsf{bv}(M) \cup \mathsf{bv}(N) \\ \mathsf{bv}(\lambda x.M) &= \mathsf{bv}(M) \cup \{x\} \end{aligned}$$

Intuitivement, l'identifiant utilisé dans une occurrence libre de variable est pertinent : il parle de quelquechose de précis "en dehors" du terme. Inversement, l'identifiant utilisé dans une occurrence liée n'est qu'une référence interne au terme : c'est une façon de désigner le bon lieur.

On peut tout de suite décrire les deux problèmes de l'exemple 3.2, qui décrivent complètement les pièges à retenir concernant la substitution :

(a) On ne substitue que les occurrences libres d'une variable.

(b) Quand on substitue une variable par N, les variables libres de N doivent rester libres.

On traduit ceci en une première définition de substitution, qui évite les deux problèmes. De ce fait, elle ne sera pas toujours bien définie, ce qu'on règlera dans un second temps.

Définition 3.4. *La substitution $M[x := N]$ est définie quand*

(a) $\mathsf{bv}(M) \cap \{x\} = \emptyset$

(b) $\mathsf{bv}(M) \cap \mathsf{fv}(N) = \emptyset$

par les équations suivantes :

$$\begin{array}{rcl} x[x := N] & = & N \\ y[x := N] & = & y \text{ pour } y \neq x \\ (M_1\ M_2)[x := N] & = & (M_1[x := N])\ (M_2[x := N]) \\ (\lambda y.M)[x := N] & = & (\lambda y.(M[x := N])) \end{array}$$

Dans le dernier cas de la définition, x et y sont des variables différentes, par l'hypothèse (a). Une modification simple consisterait à ne pas propager la substitution de x sous un lieur utilisant le même identifiant, ce qui permettrait de se passer de la condition (a). On aurait alors :

$$(\lambda y.M)[x := N] = \begin{cases} \lambda y.(M[x := N]) & \text{quand } y \neq x \\ \lambda y.M & \text{quand } y = x \end{cases}$$

La condition $\mathsf{bv}(M) \cap \mathsf{fv}(N) = \emptyset$ ne peut être évacuée aussi facilement : quand elle n'est pas satisfaite, il va falloir changer l'identifiant utilisé pour les liaisons problématiques.

Exercice 3.1. *Que vaut* $\mathsf{fv}(M[x := N])$*, quand cette substitution est bien définie ? On pourra donner une première réponse en termes d'inclusion vis-à-vis des ensembles* $\mathsf{fv}(M)$ *et* $\mathsf{fv}(N)$*, avant de donner une réponse exacte.*

3.2.1 Alpha-renommage

Quand on veut substituer y par x dans $\lambda x.y$, la définition précédente ne s'applique pas. La solution est de changer d'abord le terme cible : puisque les variables liées ne sont que des références internes, on ne change intuitivement rien en considérant $\lambda z.y$ au lieu de $\lambda x.y$. Le résultat de la substitution de y par x est alors $\lambda z.x$, la fonction constante égale à x, ce qui correspond à l'intuition : la fonction constante égale à y devient la fonction constante égale à x après substitution de y par x. L'opération consistant à changer l'identifiant utilisé pour un lieur est appelée α-renommage.

Pour définir des transformations en profondeur dans des termes on utilise la notion de contexte. Un contexte C est un terme avec un "trou", qu'on représentera par une variable spéciale $\Box$. On note $C[M]$ le terme obtenu à partir de C en remplaçant le trou par M (sans se préoccuper des variables libres ou liées). Par exemple, si C est $(\Box\ x)$ alors $C[\lambda x.x]$ est $((\lambda x.x)\ x)$. Et si C est $(\lambda x.\ x\ \Box)$, alors $C[x]$ est $\lambda x.\ x\ x$.

Définition 3.5 (Renommage). *La relation d'*α*-renommage* $\to_\alpha$ *sur les termes est définie ainsi :*

$$C[(\lambda x.M)] \to_\alpha C[(\lambda y.M[x := y])] \quad \textit{quand } y \notin \mathsf{fv}(M) \textit{ et } x, y \notin \mathsf{bv}(M)$$

La condition $x, y \notin \mathsf{bv}(M)$ ci-dessus assure que la substitution est bien définie, et la condition supplémentaire $y \notin \mathsf{fv}M$ assure que l'opération ne change pas la signification de notre terme. Sans elle, on pourrait α-renommer $(\lambda x.x\ y)$ en $(\lambda y.\ y\ y)$, ce qui est gênant.

Exercice 3.2. *Trouver un terme* M *tel que* $(\lambda x.x) \to_\alpha^* M$ *et* $x \notin \mathsf{bv}(M)$*. Même question avec* $(\lambda x.\lambda x.x)$*.*

Exercice 3.3. *Quand* $M \to_\alpha N$*, que peut-on dire des variables libres de* M *et* N *?*

Proposition 3.1. *Pour tout terme* M *et tout ensemble fini* S *de variables, il existe* M' *tel que* $M \to_\alpha^* M'$ *et* $\mathsf{bv}(M') \cap S = \emptyset$*.*

Définition 3.6 (Substitution, version totale). *Étant donnés* M*,* x *et* N*, on définit* $M[x := N]$ *comme* $M'[x := N]$ *(au sens de la version précédente de la substitution) pour un* M' *quelconque tel que* $\mathsf{bv}(M') \cap \mathsf{fv}(N) = \emptyset$ *et* $\mathsf{bv}(M') \cap \{x\} = \emptyset$ *(l'existence est assurée par le résultat précédent).*

3.3 Réduction

On peut enfin définir la β-réduction, qui correspond intuitivement à l'évaluation d'une expression en λ-calcul.

Définition 3.7 (Beta-réduction). *La relation $\to_\beta$ est définie par :*

$$C[(\lambda x.M)\ N] \to_\beta C[M[x := N]]$$

Dans une telle réduction, on dit que $(\lambda x.M)\ N$ est le redex*, i.e. le sous-terme que l'on réduit vraiment.*

Exemple 3.4. *Deux suites de réductions possibles à partir du même terme, où l'on ne parenthèse volontairement pas au maximum (faites l'exercice d'expliciter les applications imbriquées pour bien faire apparaître les redexes) :*

$$(\lambda x.\lambda y.\ x\ y)\ (\lambda x.x)\ z \to_\beta^* (\lambda y.\ (\lambda x.x)\ y)\ z \to_\beta^* (\lambda y.y)\ z \to_\beta^* z$$

$$(\lambda x.\lambda y.\ x\ y)\ (\lambda x.x)\ z \to_\beta^* (\lambda y.\ (\lambda x.x)\ y)\ z \to_\beta^* (\lambda x.x)\ z \to_\beta^* z$$

Exemple 3.5. *Le terme $(\lambda x.x)(\lambda x.x)(\lambda x.x)$ ne peut être β-réduit que vers $(\lambda x.x)(\lambda x.x)$, qui lui même se β-réduit vers $(\lambda x.x)$.*

Exercice 3.4. *On définit $M_0 = (\lambda x.x)$ et $M_{n+1} = (\lambda x.x)\ M_n$. Analyser les séquences de réductions possibles de M_n, pour $n \in \mathbb{N}$ quelconque : sont-elles toujours finies ? quel est leur nombre ?*

Il existe des termes dont la réduction ne termine pas.

Exemple 3.6. *Le terme $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ se β-réduit en lui-même. On le note traditionnellement Ω.*

Exemple 3.7. *Variante : le terme $M = (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$ se β-réduit en $(f\ M)$. Sa version abstraite $Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$ est appelée combinateur de point fixe de Curry.*

De façon surprenante, les types de données usuels peuvent être encodés au moyen des fonctions du lambda-calcul, ce qui fait donc de notre langage minimal un langage de programmation "complet". On donne ci-dessous quelques exemples ; le TD permettra d'aller plus loin.

Exemple 3.8. *On code $\overline{\mathsf{true}} = (\lambda x, y.\ \ x)$ et $\overline{\mathsf{false}} = (\lambda x, y.\ \ y)$. On définit alors* (if M then N else P) *comme une notation pour $(M\ N\ P)$. On peut vérifier les résultats suivants, qui justifient le codage :*

$$\text{if } \overline{\mathsf{true}} \text{ then } N \text{ else } P \to_\beta^* N$$

$$\text{if } \overline{\mathsf{false}} \text{ then } N \text{ else } P \to_\beta^* P$$

Exercice 3.5. *Donner un terme* $\overline{\textbf{not}}$ *tel que* $\overline{\textbf{not}}\ \overline{\textbf{true}} \to_\beta^* \overline{\textbf{false}}$ *et* $\overline{\textbf{not}}\ \overline{\textbf{false}} \to_\beta^* \overline{\textbf{true}}$*. On poursuit en cherchant des codages des diverses opérations booléennes, en énonçant leur correction de façon analogue : donner des codages* aussi concis que possible *pour la conjonction, la disjonction, le ou exclusif et la négation de la conjonction (nand).*

Exemple 3.9. *Le codage de Church d'un entier $n \in \mathbb{N}$ est défini par*

$$\overline{n} = (\lambda f.\lambda x.\ \ f^{(n)}\ x)$$

où $f^{(n)}$ est l'itérée n fois de f. On a par exemple $\overline{2} = (\lambda f.\lambda x.\ \ f\ (f\ x))$. On verra en TD comment coder les opérations usuelles sur les entiers dans ce style.

Exercice 3.6. *Pour préparer le TD sur les entiers Church, on peut chercher à coder en OCaml pur la fonction* **List.tl***, renvoyant la queue d'une liste,* sans pattern matching ni **let rec***, mais en utilisant seulement l'opération* **List.fold_left***.*

Chapitre 4

Terminaison, stratégies

On a vu la dernière fois qu'il existe des termes admettant une réduction infinie. On introduit maintenant les notions de terminaison faible et forte pour analyser ce phénomène. On use et abuse pour l'occasion des définitions inductives par règles d'inférence, pour se préparer à la suite.

Définition 4.1 (Forme normale). *Un terme M qui n'admet aucune réduction (i.e. $N \not\to_\beta$) est appelé une* forme normale.

Définition 4.2 (Normalisation faible). *Un terme M normalise faiblement s'il se réduit vers une forme normale : il existe N tel que $M \to_\beta^* N \not\to_\beta$.*

De façon équivalente, on peut définir inductivement le fait qu'un terme normalise faiblement, noté $M\ wn$ pour *weakly normalizing*, via les règles suivantes :

$$\frac{M \not\to_\beta}{M\ wn} \qquad \frac{M \to_\beta M' \quad M'\ wn}{M\ wn}$$

Dans ces règles, certaines prémisses reposent sur une notion déja définie (ici, la β-réduction) et ne sont donc pas à dériver via le jeu de règles, mais simplement à vérifier. On peut démontrer formellement l'équivalence des deux définitions :

- Si $M \to_\beta^* N \not\to_\beta$ alors $M\ wn$ est dérivable : par induction sur la longueur de la réduction de M vers N.
- Si $M\ wn$ est dérivable alors $M \to_\beta^* N \not\to_\beta$: par induction sur la dérivation de M.

À noter : dans la suite, on écrira souvent "$M\ wn$" au lieu de "$M\ wn$ est dérivable", par simplicité.

Attention, un terme peut normaliser faiblement et néanmoins admettre une réduction infinie : prendre, par exemple, $M = (\lambda x.\lambda y.y)\ \Omega$. Cela justifie une notion plus forte de normalisation.

Définition 4.3 (Normalisation forte). *On dit qu'un terme M normalise fortement s'il n'admet pas de réduction infinie $M \to_\beta M_1 \to_\beta M_2 \ldots$*

De façon équivalente, on définit $M\ sn$ pour *strongly normalizing* inductivement via une unique règle :

$$\frac{M'\ sn \text{ pour tous } M \to_\beta M'}{M\ sn}$$

Cette règle a autant de prémisses que M a de réduits possibles. Ainsi, on a immédiatemment $M\ sn$ quand M est une forme normale. L'équivalence se démontre de nouveau :

— Si $M\ sn$ est dérivable alors M n'admet pas de réduction infinie : On "voit" assez bien que les réductions correspondent aux branches de l'arbre, leur longueur est donc bornée par la hauteur de l'arbre, finie car l'arbre est fini. Ce résultat sur la longueur des réductions de M se montre par induction sur la dérivation de $M\ sn$.
— Dans l'autre sens : si M n'admet pas de réduction infinie, alors par le lemme de König la longueur de ses réductions est bornée; par induction sur cette borne on construit une dérivation de $M\ sn$.

Exemple 4.1. *On peut démontrer que $\Omega\ sn$ n'est pas dérivable : par l'absurde en considérant une dérivation de taille minimale, ou par induction sur une hypothétique dérivation.*

4.1 Confluence et raisonnement équationnel

Quand plusieurs réductions sont possibles, on peut se demander s'il est important de choisir la bonne. Ou encore, est-il possible qu'un même terme se réduise vers des formes normales différentes ? Le théorème de confluence, qui sera démontré en TD, nous indique que les différents choix de réduction n'ont pas trop d'importance.

Théorème 4.1 (Confluence). *Si $M \to_\beta^* N_1$ et $M \to_\beta^* N_2$, alors il existe M' tel que $N_1 \to_\beta^* M'$ et $N_2 \to_\beta^* M'$.*

Ce résultat va faciliter le raisonnement équationnel sur le λ-calcul, défini ainsi :

Définition 4.4 (Convertibilité). *On note $=_\beta$ la clôture réflexive symétrique transitive de $\to_\beta$. De façon équivalente :*

$$\frac{}{M =_\beta M} \qquad \frac{N =_\beta M}{M =_\beta N} \qquad \frac{M =_\beta M' \quad M' =_\beta M''}{M =_\beta M''} \qquad \frac{M \to_\beta M'}{M =_\beta M'}$$

Exemple 4.2. *On a $(\overline{2} + \overline{1}) + \overline{4} =_\beta \overline{3} + (\overline{2} + \overline{2})$.*

De façon encore équivalente, on a $M =_\beta M'$ quand il existe une suite de réductions et expansions reliant les deux termes :

$$M \to_\beta^* M_1 \leftarrow_\beta^* M_2 \to_\beta^* M_3 \leftarrow_\beta^* \ldots \to_\beta^* M_n \leftarrow_\beta^* M'$$

(Cela n'a pas d'importance de commencer ou finir par une réduction plutôt qu'une expansion, car on considère ici $\to^*$ et $\leftarrow^*$, qui sont réflexives.) Par confluence, et par induction sur n, cela implique l'existence d'un terme N tel que $M \to_\beta^* N$ et $M' \to_\beta^* N$: voila qui est plus facile à vérifier !

On introduit maintenant une forme classique d'équivalence de programmes.

Définition 4.5. *Deux termes M et N sont observationnellement équivalent si, pour tout contexte C, $C[M]$ normalise ssi $C[N]$ normalise.*

Exemple 4.3. *Quelques exemples d'équivalences et non-équivalences :*
— *$\overline{true}$ et $\overline{false}$ ne sont pas observationnellement équivalents.*
— *Ω et $Y_f = (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ ne sont pas équivalents. Avec le contexte $C = (\lambda f.\Box)(\lambda x.z)$ on ne peut toujours pas normaliser $C[\Omega]$ (c'est technique à démontrer, mais on l'admet) mais $C[Y_f] \to_\beta^2 (f\ Y_f)[f := \lambda x.z] \to_\beta z$.*
— *Par contre, Ω et $\Omega\ \Omega$ sont observationnellement équivalents. Encore une fois, on l'admet.*

Proposition 4.1. *La β-convertibilité implique l'équivalence observationnelle.*

Démonstration. Soit $M =_\beta N$, et soit un contexte C tel que $C[M]$ normalise – on montrerait de manière analogue que si $C[N]$ normalise alors $C[M]$ aussi. La convertibilité passe au contexte : on a $C[M] =_\beta C[N]$. Par confluence, on en conclut qu'il existe w tel que $C[M] \to_\beta^* w \leftarrow_\beta^* C[N]$. Comme $C[M] \to_\beta^* w'$ en forme normale, on en déduit de nouveau par confluence que $w \to_\beta^* w'$, et enfin $C[N] \to_\beta^* w$, donc $C[N]$ normalise. □

La convertibilité est donc correcte vis-a-vis de l'équivalence observationnelle. Mais elle n'est pas complète : il existe des termes observationnellement équivalents mais non convertibles, par exemple x et $\lambda y.x\ y$.

4.2 Stratégies de réduction

On a vu qu'il peut y avoir plusieurs redexes possibles dans un terme. La confluence indique que ces choix n'ont pas d'importance sur le résultat du calcul. Néanmoins, la stratégie de réduction (i.e. la façon de choisir quel redex traiter à chaque étape) peut avoir une importance sur la longueur des réductions et même sur la terminaison du processus.

4.2.1 Standardisation

Pour la terminaison, la stratégie externe gauche est la plus importante. On rappelle qu'un redex est un sous-terme de la forme $\big((\lambda x.M)\ N\big)$.

Définition 4.6 (Stratégie externe gauche)**.** *On dit qu'un redex est externe s'il n'est pas sous-terme (strict) d'un autre redex. La stratégie externe gauche consiste à toujours choisir le redex externe le plus à gauche.*

Exemple 4.4. *Considérons le terme suivant :*

$$\big((\lambda x.(\lambda y.y)\ x)\ M\big)\ \big((\lambda x.x)\ N\big)$$

Le redex le plus à gauche est $(\lambda y.y)\ x$, mais il n'est pas externe. On a deux redexes externes, et le plus à gauche est $\big((\lambda x.(\lambda y.y)\ x)\ M\big)$.

Théorème 4.2 (Standardisation)**.** *Si M est faiblement normalisant, alors la stratégie externe gauche permet de calculer sa forme normale.*

Nous ne montrerons pas ce résultat : c'est très technique et pas central d'un point de vue langages de programmation. Intuitivement, $M\ N$ ne peut normaliser que si M normalise : c'est pour cela que la stratégie travaille à gauche d'abord. De même, pour normaliser $\big((\lambda x.M)\ N\big)$ il faudra nécessairement réduire ce redex, mais pas forcément les redexes internes.

Exemple 4.5. *Si l'on travaille à droite d'abord, on peut perdre son temps, infiniment longtemps : considérer $(\lambda x.y)\ \Omega$.*

Exemple 4.6. *Pour $(\lambda x.x\ \Omega)\ (\lambda x.z)$ seule la réduction du redex externe (unique) permet d'atteindre la forme normale. La réduction du redex interne (unique aussi) nous ramène au terme de départ.*

On peut caractériser les réductions selon la stratégie externe gauche via une définition inductive par règles :

$$\frac{}{(\lambda x.M)\ N \to_{EG} M[x := N]} \qquad \frac{M \to_{EG} M'}{\lambda x.M \to_{EG} \lambda x.M'}$$

$$\frac{M\ N \text{ pas un } \beta\text{-redex} \quad M \to_{EG} M'}{M\ N \to_{EG} M'\ N} \qquad \frac{M \not\to_\beta \quad N \to_{EG} N'}{M\ N \to_{EG} M\ N'}$$

4.2.2 Appels par nom et par valeur

On introduit une typologie de stratégies importantes dans une optique plus pratique de programmation.

Définition 4.7. *Une stratégie (ou réduction) est dite faible si elle ne réduit pas sous les abstractions. Cela exclut bien sûr de trouver une forme normale : on considèrera dans ce cas qu'un terme est en forme normale si tous ses redexes pour la β-réduction sont sous des abstractions.*

Tous les langages de programmation "pratiques" ont des stratégies faibles : on ne s'intéresse jamais à réduire une fonction pour elle même, on ne réduit que ses applications à certains arguments.

Définition 4.8. *Une stratégie est en appel par valeur si on ne réduit $(\lambda x.M)\ N$ que quand N est une forme normale.*

OCaml est en appel par valeur. La stratégie interne droite est en appel par valeur. Mais sur le terme $(\lambda x.\lambda y.M)\ N_1\ N_2$ la réduction de N_1 avant N_2 serait aussi qualifiée d'appel par valeur : l'appel par valeur n'est pas une stratégie mais une famille de stratégies.

Définition 4.9. *Une stratégie est en appel par nom si les arguments des fonctions ne sont pas réduits avant l'appel de fonction. Autrement, on ne réduit pas N dans $(\lambda x.M)\ N$. Plus généralement, on ne réduit N dans $M\ N$ que si M est une forme normale qui ne soit pas une abstraction.*

La stratégie externe gauche est en appel par nom. Historiquement, la version faible de la stratégie externe gauche était une stratégie possible en Algol (et c'est dans ce contexte que la terminologie a été établie) mais aujourd'hui aucun langage majeur n'utilise cette stratégie. Par abus, on pourrait dire que les langages de macro apportent un peu d'appel par nom dans leurs langages hôtes.

L'appel par nom s'apparente à la paresse, mais la paresse est plus complexe, impliquant de ne pas réduire plusieurs fois un terme : on réduit $(\lambda x.M)\ N$ avant de réduire N, si on a de la chance N n'aura même pas à être réduite, mais si on doit le réduire alors une forme de partage est introduite pour ne pas avoir à réduire séparemment les différentes copies de N créées par la substitution $M[x := N]$.

Chapitre 5

Sémantique à grands pas

Une sémantique est dite "à grands pas" quand elle met en relation un programme avec son résultat final : c'est ce qu'on faisait avec la relation d'évaluation du chapitre 2. Par contraste, la relation de β-réduction du λ-calcul est dite "à petits pas" car elle explicite chaque pas de calcul, permettant de voir l'évaluation comme une suite de petits pas – mais aussi, de constater que certains calculs ne terminent pas, ce que la sémantique à grands pas ne pourra pas "dire" que comme l'absence de résultat.

Nous définissons dans ce chapitre plusieurs sémantiques à grand pas, sous la forme de relations $\Downarrow$. On pourrait les noter $\Downarrow_1, \Downarrow_2$, etc. ou quelquechose d'un peu plus informatif, mais j'ai choisi de garder des notations légères et de préciser à l'utilisation à quelle sémantique on fait référence.

5.1 Stratégies pour le λ-calcul pur

On commence par donner la version "à grands pas" de deux stratégies de réduction faibles pour le λ-calcul pur, en appel par valeur et par nom. L'objectif est de spécifier, via une relation fonctionnelle, un calcul déterministe qui à partir de tout terme donne sa forme normale (si elle existe) pour la stratégie de réduction considérée.

Dans la suite, on appellera forme normale faible un terme irréductible pour des stratégies faibles (Définition 4.7). Qu'on soit en appel par valeur ou par nom (Définitions 4.8 et 4.9) n'a pas d'importance, car on est irréductible en appel par valeur ssi on est irréducible en appel par nom.

Proposition 5.1. *On considère une stratégie ne réduisant pas sous les abstractions, mais réduisant tout terme contenant au moins un redex qui n'est pas situé sous une abstraction. Un terme N est une forme normale pour cette stratégie ssi il appartient au plus petit ensemble $\mathcal{V}$ tel que :*

- *les abstractions (i.e. les termes de la forme $\lambda x.M$) sont dans $\mathcal{V}$;*
- *si $N_1, \ldots, N_k \in \mathcal{V}$ alors $(x\ N_1 \ldots N_k) \in \mathcal{V}$.*

5.1.1 Appel par nom

Dans cette section, on appelle appel par nom la stratégie externe gauche faible. L'objectif est de donner une relation d'évaluation correspondant à cette stratégie.

Définition 5.1. *On définit inductivement la relation binaire $\Downarrow$ sur les λ-termes par les règles de la figure 5.1.*

Exemple 5.1. *En posant comme d'habitude $\delta = (\lambda x.x\ x)$ et $\Omega = (\delta\ \delta)$, on a :*

- *δ s'évalue en lui-même : $\delta \Downarrow \delta$ est dérivable ;*

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\frac{M \Downarrow \lambda x.M' \quad M'[x := N] \Downarrow R}{M\ N \Downarrow R} \qquad \frac{M \Downarrow x\ N_1 \dots N_k \quad N \Downarrow N'}{M\ N \Downarrow x\ N_1 \dots N_k\ N'}$$

FIGURE 5.1 – Lambda-calcul pur en appel par nom

— *Ω ne s'évalue pas : $\Omega \not\Downarrow$, i.e. pour tout N on a $\Omega \not\Downarrow N$;*
— *néanmoins $(\lambda x.y)\ \Omega \Downarrow y$ est dérivable.*

La relation spécifiée est fonctionnelle : pour tous M, N_1, N_2 tels que $M \Downarrow N_1$ et $M \Downarrow N_2$, on a $N_1 = N_2$. Cela se démontre par induction sur la dérivation de $M \Downarrow N_1$. C'est un bon signe puisqu'on espère que notre évaluation corresponde à la stratégie déterministe d'appel par nom.

Proposition 5.2. *Dans le contexte de la définition de la figure 5.1, pour tous M et N tels que $M \Downarrow N$, on a $N \in \mathcal{V}$.*

Exercice 5.1. *Prouver ce résultat.*

Quand on travaille sur une relation d'évaluation plutôt que de réduction, on parlera de *valeur* d'un programme plutôt que de sa *forme normale*. Malgré ce changement de terminologie, des liens forts peuvent exister entre les deux approches ; c'est le cas pour l'appel par nom.

Proposition 5.3. *Avec la définition de la figure 5.1, on a, pour tous termes M et N :*
— *Si $M \Downarrow N$ alors $M \to_\beta^* N$ selon la stratégie externe, gauche et faible et $N \in \mathcal{V}$.*
— *Si $M \to_\beta^* N$ selon la stratégie externe gauche et faible et $N \in \mathcal{V}$, alors $M \Downarrow N$.*

Démonstration. La sens direct (i.e., évaluation implique réduction) se démontre par induction sur la dérivation de $M \Downarrow N$; on l'a vu en cours. La réciproque est à traiter ci-dessous en exercice. □

Exercice 5.2. *Supposons qu'on remplace la troisième règle par la suivante :*

$$\frac{M \Downarrow \lambda x.M'}{M\ N \Downarrow M'[x := N]}$$

Que devient la proposition 5.3 ? quelles directions du résultat restent vraies ?

Exercice 5.3.

(a) Démontrer que pour toute forme normale faible $N \in \mathcal{V}$, on peut dériver $N \Downarrow N$.
(b) Démontrer que, pour tous M, M' et N tels que $M \to_\beta M'$ en appel par nom et $M' \Downarrow N$, on a $M \Downarrow N$.
(c) En déduire le sens réciproque dans la proposition 5.3.

Le résultat précédent nous indique que, si l'on a $M \not\Downarrow$ pour un terme M, c'est que M a une réduction infinie selon notre stratégie. Dans un langage de programmation plus général, on pourrait aussi avoir $M \not\Downarrow$ en raison d'une erreur à l'exécution ; c'est un accident que la sémantique à grands pas nous permette ici de parle de non-terminaison du calcul.

Variante close

Si l'on ne s'intéresse qu'aux termes clos, alors l'ensemble pertinent de formes normales (et de valeurs) est l'ensemble $\mathcal{V}_c$ des termes clos de $\mathcal{V}$ c'est à dire les abstractions closes.

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.M' \quad M'[x := N] \Downarrow R}{M\ N \Downarrow R}$$

FIGURE 5.2 – Lambda-calcul pur en appel par nom, variante close

Proposition 5.4. *Dans le contexte de la figure 5.2, pour tout terme clos M et pour tout terme N, on a $M \Downarrow N$ ssi ($M \to_\beta^* N$ selon la stratégie externe gauche faible et $N \in \mathcal{V}_c$).*

Pour cette relation d'évaluation, un terme non clos ne peut être évalué. On peut donc avoir $M \not\Downarrow$ sans avoir non-terminaison.

5.1.2 Appel par valeur

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\frac{M \Downarrow \lambda x.M' \quad N \Downarrow N' \quad M'[x := N'] \Downarrow R}{M\ N \Downarrow R} \qquad \frac{M \Downarrow x\ N_1 \dots N_k \quad N \Downarrow N'}{MN \Downarrow x\ N_1 \dots N_k\ N'}$$

FIGURE 5.3 – Lambda-calcul pur en appel par valeur

Définition 5.2. *La relation d'évaluation $\Downarrow$ pour l'appel par valeur est donnée par les règles de la figure 5.3.*

Proposition 5.5. *Pour tous termes M et N, on a $M \Downarrow N$ ssi ($M \to_\beta^* N$ selon une stratégie interne faible et $N \in \mathcal{V}$).*

Le schéma général de la preuve est comme pour la proposition 5.3. Ce résultat est vrai bien que la stratégie en appel par valeur ne soit pas déterministe : sur un terme $M_1\ M_2$ elle permet de réduire dans n'importe quel ordre M_1 et M_2 (et même d'entrelacer leurs réductions) ; la seule contrainte est qu'on ne réduit pas un redex $(\lambda x.M')\ M''$ à la racine quand M'' peut encore être réduit.

Variante close

On donne en figure 5.4 la variante de l'appel par valeur pour les termes clos, pour faciliter la comparaison avec la sémantique avec environnements de la section suivante.

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.M' \quad N \Downarrow N' \quad M'[x := N'] \Downarrow R}{M\ N \Downarrow R}$$

FIGURE 5.4 – Lambda-calcul pur en appel par valeur, variante close

Il est clair que $M \Downarrow N$ dans cette variante entraîne $M \Downarrow N$ dans la version précédente, et donc $M \to^* N \in \mathcal{V}$.

Exercice 5.4. *Démontrer la réciproque quand M est clos.*

5.2 Environnements

Les relations que l'on a définies sont fonctionnelles : pour un terme M, il existe au plus un N tel que $M \Downarrow N$. Au delà de ça, ce N peut être calculé en suivant récursivement la définition de la relation d'évaluation. Par exemple, pour l'évaluation de la

définition 5.2, si M est une application $M_1 M_2$, on évalue M_1. On obtient $M_1 \Downarrow M_1'$ et l'on suit la recette par la dernière ou avant-dernière règle en fonction de la forme de M_1' :

- Si M_1' est une abstraction $\lambda x.M_1''$, on évalue M_2 pour obtenir M_2'. Il ne reste plus qu'à évaluer $M_1''[x := M_2']$ pour obtenir le résultat final N tel que $M \Downarrow N$.
- Sinon, on est forcément dans un cas d'application de la quatrième règle et on finit le calcul en évaluant M_2.

Si l'on implémentait un évaluateur ainsi, les calculs de substitutions seraient compliqués à implémenter (attention aux problèmes de captures de variables) et occasionneraient un coût déraisonnable. On évite cela via la notion d'environnement. L'objectif est de passer à une relation ternaire $E \vdash M \Downarrow N$ où E est une application des variables dans les valeurs. Une première intuition est que

$$x_1 \mapsto N_1, \ldots, x_k \mapsto N_k \vdash M \Downarrow N$$

va correspondre à

$$M[x_1 := N_1, \ldots, x_k := N_k] \Downarrow N.$$

Autrement dit, l'environnement est une substitution que l'on va calculer implicitement (et paresseusement) lors de l'évaluation.

Pour simplifier cette section, on va se restreindre à des valeurs closes, i.e. des abstractions, et considérer une stratégie d'évaluation en appel par valeur. Cela correspond par ailleurs à ce qu'on aura dans la plupart des langages de programmation.

$$\frac{}{E \vdash x \Downarrow E(x)} \qquad \frac{}{E \vdash \lambda x.M \Downarrow \langle E, \lambda x.M \rangle}$$

$$\frac{E \vdash M \Downarrow \langle E', \lambda x.M' \rangle \quad E \vdash N \Downarrow N' \quad E', x \mapsto N' \vdash M' \Downarrow R}{E \vdash M\ N \Downarrow R}$$

FIGURE 5.5 – Appel par valeur avec environnements

Un traitement naïf des abstractions mène à une erreur, i.e. une incohérence avec la sémantique par réduction. Supposons qu'on prenne simplement les règles suivantes :

$$\frac{}{E \vdash \lambda x.M \Downarrow \lambda x.M} \qquad \frac{E \vdash M \Downarrow \lambda x.M' \quad E \vdash N \Downarrow N' \quad E, x \mapsto N' \vdash M' \Downarrow R}{E \vdash M\ N \Downarrow R}$$

Considérons le terme $(((\lambda x.\lambda y.x)\ M_1)\ M_2)$ où M_1 et M_2 sont des abstractions closes, donc des valeurs. Pour évaluer ce terme il faut d'abord évaluer son sous-terme gauche. Construisons la seule dérivation possible :

$$\frac{\dfrac{}{\emptyset \vdash (\lambda x.\lambda y.x) \Downarrow (\lambda x.\lambda y.x)} \quad \dfrac{}{\emptyset \vdash M_1 \Downarrow M_1} \quad \dfrac{}{x \mapsto M_1 \vdash (\lambda y.x) \Downarrow (\lambda y.x)}}{\emptyset \vdash (\lambda x.\lambda y.x)\ M_1 \Downarrow (\lambda y.x)}$$

La variable libre x est louche puisqu'on avait évalué un terme clos ! De fait, on ne pourra évaluer x dans $y \mapsto E_2$: il n'y a aucun R tel que $\emptyset \vdash (((\lambda x.\lambda y.x)\ M_1)\ M_2) \Downarrow R$, alors qu'on aimerait pouvoir dériver ici une évaluation vers M_1. Voici la seule dérivation possible, qu'on ne peut compléter :

$$\frac{\dfrac{\vdots}{\emptyset \vdash ((\lambda x.\lambda y.x)\ M_1) \Downarrow (\lambda y.x)} \quad \dfrac{}{\emptyset \vdash M_2 \Downarrow M_2} \quad y \mapsto M_2 \vdash x \Downarrow \ldots}{\emptyset \vdash (((\lambda x.\lambda y.x)\ M_1)\ M_2) \Downarrow \ldots}$$

Pour remédier à ce problème, on enrichit les abstractions en des *clôtures* qui empaquètent une l'abstraction avec l'environnement dans laquelle elle doit être considérée.

Définition 5.3. *On définit de façon mutuellement inductive les valeurs et les environnements :*

- *Une valeur est nécessairement une* clôture*, notée $\langle E, \lambda x.M \rangle$ et composée d'un environnement et d'une abstraction.*
- *Un environnement est une fonction partielle des variables dans les valeurs.*

Le fait que cette définition est inductive signifie que la construction doit être bien fondée : un environnement contient des valeurs, qui contiennent des environnements, qui contiennent des valeurs, etc. mais cette chaîne doit s'arrêter. Le seul cas de base possible pour cela est celui d'un environnement de domaine vide.

Définition 5.4. *La relation d'évaluation $E \vdash M \Downarrow N$ pour les termes clos en appel par valeur avec environnements est donnée par les règle de la figure 5.5.*

Pour pouvoir énoncer en quoi cette sémantique correspond aux notions précédentes, il nous faut définir plus précisément en quoi un environnement peut être vu comme une substitution.

Définition 5.5. *Si M est un terme et E un environnement, on définit par induction sur E le terme ME comme suit. L'environnement s'écrit nécessairement*

$$E = \{x_1 \mapsto \langle E_1, \lambda x.M_1 \rangle, \ldots, x_k \mapsto \langle E_k, \lambda x.M_k \rangle\}$$

et l'on pose alors :

$$ME \stackrel{def}{=} M[x_1 := \lambda x.(M_1 E_1)] \ldots [x_k := \lambda x.(M_k E_k)]$$

Noter que pour définir ME on a d'abord besoin de définir les $M_i E_i$, ce qu'on peut supposer du fait du caractère inductif des environnements.

Cette définition nous permet de relier cette sémantique ($E \vdash M \Downarrow N$) à la précédente ($M \Downarrow N$, cf. figure 5.4) et, de là, à la sémantique à petits pas du λ-calcul.

Proposition 5.6. *Soit M un terme clos.*

- *S'il existe une abstraction close N telle que $M \to_\beta^* N$ en appel par valeur, alors il existe E et N' tel que $\emptyset \vdash M \Downarrow \langle E, N' \rangle$ et $N'E = N$.*
- *Réciproquement, s'il existe E et N' tel que $\emptyset \vdash M \Downarrow \langle E, N' \rangle$ alors $M \to_\beta^* N'E$ en appel par valeur, et $N'E$ est une abstraction close.*

Exercice 5.5. *Pour démontrer ce résultat par induction, il va falloir le généraliser à des environnements non vides. La condition "M clos" va alors devenir* $\mathsf{fv}(M) \subseteq \mathsf{dom}(E)$*. Mais cela ne suffit pas… saurez-vous identifier le problème ? proposer une solution ?*

5.3 Le langage mini-ML

On construit progressivement un mini ML, en donnant sa sémantique à grands pas. Un point crucial est de bien distinguer mémoire et environnement. Les valeurs vont par ailleurs s'éloigner encore des expressions (programmes) avec l'apparition des *adresses* mémoire.

Définition 5.6 (Programmes)**.** *Les programmes sont donnés par la grammaire suivante, où $x \in \mathcal{X}$ et $n \in \mathbb{Z}$:*

$$\begin{array}{rcl} M & ::= & x \mid \textbf{fun}\ x \mapsto M \mid (M\ M') \\ & \mid & n \mid M \oplus M' \mid M \otimes M' \mid \ldots \mid \textbf{ifz}\ M\ \textbf{then}\ M'\ \textbf{else}\ M'' \\ & \mid & \textbf{let}\ x = M\ \textbf{in}\ M' \mid () \mid (M_1; M_2) \\ & \mid & \textbf{ref}\ M \mid !M \mid M := M' \\ & \mid & \textbf{fixfun}\ f\ x \mapsto M \end{array}$$

On a utilisé $\oplus$ et $\otimes$ pour distinguer ces constructions syntaxiques des opérations usuelles sur $\mathbb{Z}$. La conditionnelle **ifz** n **then** ... correspond intuitivement au **if** $n = 0$ **then** ... usuel.

On se donne un ensemble $\mathcal{A}$ d'adresses : c'est un ensemble infini d'identifiants, sur lequel on n'aura besoin d'aucune structure particulière, il est donc analogue (mais distinct de) l'ensemble $\mathcal{X}$ des variables.

Définition 5.7 (Environnements, valeurs, états)**.** *On définit simultanément environnements et valeurs, de façon mutuellement inductive :*

- *Un environnement est une fonction partielle des variables dans les valeurs.*
- *Une valeur est, au choix :*
  - $()$ *;*
  - *un entier* $n \in \mathbb{Z}$ *;*
  - *une clôture* $\langle E, M \rangle$ *où* M *est un terme construit au moyen de* **fun** *ou* **fixfun** *;*
  - *une adresse* $A \in \mathcal{A}$*.*

Un état de la mémoire est une fonction partielle de $\mathcal{A}$ *dans les valeurs.*

La sémantique est donnée en figure 5.6. La relation d'évaluation, notée $E \vdash \sigma, M \Downarrow \sigma', V$ est d'arité 5, impliquant un environnement, un état initial et un état final, le terme (ou programme) à réduire et la valeur résultant de l'évaluation.

Exercice 5.6. *En Mini-ML, contrairement au* λ*-calcul pur, la non-terminaison n'est plus le seul obstacle à l'évaluation, mais diverses erreurs peuvent se produire à l'exécution. Formellement, pour* E, σ, M *donnés, diverses raisons peuvent faire qu'il n'existe aucun* σ', V *tels que* $E \vdash \sigma, M \Downarrow \sigma', V$ *soit dérivable. La non-terminaison reste bien sûr une de ces raisons, mais cela peut aussi venir du fait qu'on utilise une opération arithmétique sur une expression dont la valeur n'est pas un entier. Pouvez-vous recenser les autres raisons ?*

Exercice 5.7. *Vérifier sur les exemples suivants que la sémantique de Mini-ML correspond bien à celle d'OCaml (à l'exception de l'ordre d'évaluation des sous-expressions) :*

- **let** $x = 42$ **in let** $f =$ **fun** $() \mapsto x$ **in let** $x = 3$ **in** $f\ ()$
- **let** $x =$ **ref** 42 **in let** $f =$ **fun** $() \mapsto !x$ **in** $x := 3; f()$
- **let** $x =$ **ref** 42 **in let** $y = x$ **in** $y := 42; !x$
- **let** $x =$ **ref** 42 **in let** $f() = x := !x + 1$ **in** $f\ (); !x$

On remarquera le traitement fondamentalement différent des environnements et des états : lors de l'exécution un unique état est chaîné d'une étape d'exécution à une autre, tandis que les environnements de définition des fonctions sont restaurés lors d'un appel de fonction (liaison statique des variables).

Exercice 5.8. *Évaluer* **let** $f = \big($**let** $c =$ **ref** 0 **in fun** $x \mapsto (c := !c+1; !c)\big)$ **in** $f\ (); f\ ()$.

Expressions arithmétiques

$$\frac{}{E \vdash \sigma, n \Downarrow \sigma, n} \qquad \frac{E \vdash \sigma, M \Downarrow \sigma_1, m \quad E \vdash \sigma_1, N \Downarrow \sigma', n}{E \vdash \sigma, (M \oplus N) \Downarrow \sigma', (m+n)} \qquad \text{etc.}$$

$$\frac{E \vdash \sigma, M \Downarrow \sigma', 0 \quad E \vdash \sigma', M' \Downarrow \sigma'', R}{E \vdash \sigma, (\textbf{ifz } M \textbf{ then } M' \textbf{ else } M'') \Downarrow \sigma'', R} \qquad \frac{E \vdash \sigma, M \Downarrow \sigma', n \quad E \vdash \sigma', M'' \Downarrow \sigma'', R}{E \vdash \sigma, (\textbf{ifz } M \textbf{ then } M' \textbf{ else } M'') \Downarrow \sigma'', R}\ n \in \mathbb{Z}^*$$

Cœur fonctionnel

$$\frac{}{E \vdash \sigma, x \Downarrow \sigma, E(x)} \qquad \frac{}{E \vdash \sigma, (\textbf{fun } x \mapsto M) \Downarrow \sigma, \langle E, \textbf{fun } x \mapsto M \rangle}$$

$$\frac{E \vdash \sigma, M \Downarrow \sigma_1, \langle E', \textbf{fun } x \mapsto M' \rangle \quad E \vdash \sigma_1, N \Downarrow \sigma_2, V \quad E' + (x \mapsto V) \vdash \sigma_2, M' \Downarrow \sigma', R}{E \vdash \sigma, (M\ N) \Downarrow \sigma', R}$$

Séquençage

$$\frac{E \vdash \sigma, M \Downarrow \sigma', V \quad E + (x \mapsto V) \vdash \sigma', M' \Downarrow \sigma'', R}{E \vdash \sigma, \textbf{let } x = M \textbf{ in } M' \Downarrow \sigma'', R}$$

$$\frac{E \vdash \sigma, M \Downarrow \sigma', V \quad E \vdash \sigma', M' \Downarrow \sigma'', R}{E \vdash \sigma, (M; M') \Downarrow \sigma'', R} \qquad \frac{}{E \vdash \sigma, () \Downarrow \sigma, ()}$$

Références

$$\frac{E \vdash \sigma, M \Downarrow \sigma', V}{E \vdash \sigma, (\textbf{ref } M) \Downarrow \sigma'', A}\ A \notin \mathsf{dom}(\sigma'),\ \sigma'' = \sigma' + (A \mapsto V)$$

$$\frac{E \vdash \sigma, M \Downarrow \sigma', A}{E \vdash \sigma, !M \Downarrow \sigma', \sigma'(A)} \qquad \frac{E \vdash \sigma, M \Downarrow \sigma', A \quad E \vdash \sigma', M' \Downarrow \sigma'', V}{E \vdash \sigma, (M := M') \Downarrow \sigma'' + (A \mapsto V), R}$$

Dans la règle pour l'assignation, $\sigma'' + (A \mapsto V)$ est l'état identique à σ'' sauf pour $\sigma''(A) = V$. On pourrait choisir d'exiger $A \in \mathsf{dom}(\sigma')$ dans cette règle puisqu'on l'assurerait *in fine* par typage.

Définition récursive

$$\frac{}{E \vdash \sigma, (\textbf{fixfun } f\ x \mapsto M) \Downarrow \sigma, \langle E, \textbf{fixfun } f\ x \mapsto M \rangle}$$

$$\frac{\begin{array}{l} E \vdash \sigma, M \Downarrow \sigma_1, \langle E', \textbf{fixfun } f\ x \mapsto M' \rangle \\ E \vdash \sigma_1, N \Downarrow \sigma_2, V \\ E' + (x \mapsto V, f \mapsto \langle E', \textbf{fixfun } f\ x \mapsto M' \rangle) \vdash \sigma_2, M' \Downarrow \sigma', R \end{array}}{E \vdash \sigma, (M\ N) \Downarrow \sigma', R}$$

FIGURE 5.6 – Sémantique à grands pas de mini-ML