Sémantique opérationnelle et typage

Samuel Mimram

21 novembre 2024

Table des matières

1 Sémantique opérationnelle **2**
1.1 Induction structurelle . . . 2
1.2 Le langage IMP . . . 3
1.2.1 Syntaxe . . . 3
1.2.2 Sémantique intuitive . . . 4
1.2.3 Quelques exemples . . . 4
1.3 Sémantique à grands pas . . . 5
1.3.1 Sémantique des expressions arithmétiques et booléennes . . . 5
1.3.2 Sémantique des commandes . . . 5
1.3.3 Déterminisme de l'évaluation . . . 7
1.3.4 Équivalence observationnelle . . . 9
1.4 Sémantique à petits pas . . . 9
1.4.1 Définition de la réduction . . . 9
1.4.2 Équivalence avec la réduction à grands pas . . . 12
1.4.3 Déterminisme de la réduction . . . 15

2 Typage **15**
2.1 Le langage mini-ML . . . 16
2.2 Typage . . . 17
2.2.1 Types . . . 17
2.2.2 Typage . . . 17
2.2.3 Premières propriétés du système de type . . . 18
2.2.4 Programmes non-typables . . . 19
2.2.5 Unicité du typage . . . 19
2.3 Sémantique . . . 21
2.3.1 Évaluation . . . 21
2.3.2 Substitution et renommage . . . 24
2.3.3 Réduction . . . 25
2.3.4 Formes normales . . . 26
2.3.5 Équivalence des sémantiques . . . 27
2.3.6 Extensions du langage . . . 28
2.4 Sûreté du typage . . . 29
2.5 Points fixes . . . 31
2.5.1 Opérateurs de points fixes en OCaml . . . 32
2.5.2 Opérateur de point fixe en mini-ML . . . 33
2.6 Inférence de type . . . 33
2.6.1 Types principaux . . . 34
2.6.2 Typage comme résolution d'un système d'équations . . . 35
2.6.3 Résolution des systèmes d'équations . . . 37
2.6.4 Polymorphisme . . . 39

3 Bibliographie **40**

A Évaluation de IMP **42**

B Typage en mini-ML **43**

Ce document contient les notes des cours donnés pour la préparation à l'agrégation d'informatique à l'université *Sorbonne Université Sciences* à partir de 2021. Elles couvrent la sémantique opérationnelle (section 1) et le typage (section 2), en s'efforçant de suivre le programme en vigueur (épreuve spécifique *fondements de l'informatique*).

1 Sémantique opérationnelle

Afin de pouvoir montrer des propriétés sur des programmes, il faut commencer par définir la *sémantique opérationnelle* du langage de programmation considéré, c'est-à-dire décrire formellement ce que font les programmes. Nous allons voir qu'il y a deux approches pour ce faire. Dans la sémantique opérationnelle à *grands pas*, on cherche à définir le résultat d'un programme, en formalisant son *évaluation* vers un résultat (une *valeur*). Dans la sémantique opérationnelle à *petits pas*, on entre plus en détails en décrivant sa *réduction*, c'est-à-dire la suite d'étapes de calcul d'un programme, qui va lui permettre d'aboutir au résultat, quand il y en a un (certains programmes bouclent et n'aboutissent jamais à un résultat).

Nous introduisons ici un petit langage de programmation appelé IMP (section 1.2) et donnons sa sémantique opérationnelle dans les deux approches (à grands pas en section 1.3 et à petits pas en section 1.4). Nous montrons ensuite les propriétés principales telles que le déterminisme de l'évaluation (l'exécution d'un programme produit toujours le même résultat) et le fait que les deux approches coincident. Nous verrons que les preuves nécessitent trois types de raisonnement : par induction structurelle sur la syntaxe (rappelé en section 1.1), par induction sur les dérivations et par récurrence sur la longueur des réductions.

1.1 Induction structurelle

Commençons par rappeler les définitions inductives et les principes de raisonnement associés, sur un exemple. On définit les *expressions arithmétiques* par la grammaire

$$a \quad ::= \quad \underline{n} \quad | \quad a \texttt{ + } a' \quad | \quad a \texttt{ * } a'$$

où n est un entier quelconque. Cela signifie que l'ensemble **Aexp** des expressions arithmétiques est le plus petit ensemble tel que

— pour tout $n \in \mathbb{N}$, on a $\underline{n} \in \mathbf{Aexp}$,
— pour toute paire d'expressions $a, a' \in \mathbf{Aexp}$, on a $a \texttt{ + } a' \in \mathbf{Aexp}$,
— pour toute paire d'expressions $a, a' \in \mathbf{Aexp}$, on a $a \texttt{ * } a' \in \mathbf{Aexp}$.

Par exemple, on a l'expression arithmétique

$$((\underline{6} \texttt{ + } \underline{5}) \texttt{ * } \underline{3}) \texttt{ + } (\underline{2} \texttt{ + } \underline{2})$$

Formellement, les expressions sont des arbres d'opérations : on s'autorise à utiliser des parenthèses pour rendre la notation non-ambigüe bien que celles-ci ne fassent pas partie de la syntaxe.

Étant donné un ensmble X, on peut définir des fonctions $\mathbf{Aexp} \to X$ en utilisant une définition inductive :

Théorème 1 (Définition inductive)**.** *Supposons donnés*
— *un élément* $d_{\underline{n}} \in X$ *pour tout entier* $n \in \mathbb{N}$,
— *une fonction* $d_{\texttt{+}} : X \times X \to X$,
— *une fonction* $d_{\texttt{*}} : X \times X \to X$.
Alors il existe une unique fonction

$$f : \mathbf{Aexp} \to X$$

telle que

$$f(\underline{n}) = d_{\underline{n}} \qquad f(a \texttt{ + } a') = d_{\texttt{+}}(f(a), f(a')) \qquad f(a \texttt{ * } a') = d_{\texttt{*}}(f(a), f(a'))$$

qui est appelée la fonction définie inductivement *par les données ci-dessus.*

Exemple 2. On peut définir une fonction $[\![-]\!] : \mathbf{Aexp} \to \mathbb{Z}$, appelée *évaluation* d'une expression arithmétique, par induction par

$$d_{\underline{n}} = n \qquad d_{\texttt{+}} = + \qquad d_{\texttt{*}} = \times$$

ce que l'on notera généralement plutôt

$$[\![\underline{n}]\!] = n \qquad [\![a \texttt{ + } a']\!] = [\![a]\!] + [\![a']\!] \qquad [\![a \texttt{ * } a']\!] = [\![a]\!] \times [\![a']\!]$$

dans la suite. Par exemple, on a

$$[\![(\underline{6} \texttt{ + } \underline{5}) \texttt{ * } \underline{3}) \texttt{ + } (\underline{2} \texttt{ + } \underline{2})]\!] = 37$$

Exemple 3. On peut définir le nombre d'opérations dans une expressions comme la fonction $|-| : \mathbf{Aexp} \to \mathbb{N}$ définie par induction par

$$|\underline{n}| = 0 \qquad |a \texttt{ + } a'| = |a| + |a'| + 1 \qquad |a \texttt{ * } a'| = |a| + |a'| + 1$$

Par exemple, on a

$$[\![(\underline{6} \texttt{ + } \underline{5}) \texttt{ * } \underline{3}) \texttt{ + } (\underline{2} \texttt{ + } \underline{2})]\!] = 4$$

De même, il sera souvent utile de raisonner par induction sur les expressions en utilisant le principe suivant :

Théorème 4 (Induction structurelle)**.** *Soit $P(a)$ une propriété sur les expressions arithmétiques telle que*
— *$P(\underline{n})$ est vraie pour tout $n \in \mathbb{N}$,*
— *si $P(a)$ et $P(a')$ sont vraies alors $P(a \texttt{ + } a')$ est vraie,*
— *si $P(a)$ et $P(a')$ sont vraies alors $P(a \texttt{ * } a')$ est vraie.*
Alors $P(a)$ est vraie pour toute expression $a \in \mathbf{Aexp}$.

Exemple 5. Montrons que pour toute expression a, on a la propriété $P(a)$ suivante : « $[\![a]\!] \geqslant 0$ ».
On le montre par induction structurelle :
— pour $n \in \mathbb{N}$, on a $[\![\underline{n}]\!] = n \geqslant 0$,
— si $[\![a]\!] \geqslant 0$ et $[\![a']\!] \geqslant 0$ alors $[\![a \texttt{ + } a']\!] = [\![a]\!] + [\![a']\!] \geqslant 0$,
— si $[\![a]\!] \geqslant 0$ et $[\![a']\!] \geqslant 0$ alors $[\![a \texttt{ * } a']\!] = [\![a]\!] \times [\![a']\!] \geqslant 0$.
On en déduit que pour toute expression a, on a $[\![a]\!] \geqslant 0$.

Les principes énoncés ci-dessus se généralisent bien sûr à tous les langages définis par une grammaire.

1.2 Le langage IMP

1.2.1 Syntaxe

Le langage *IMP* est un exemple représentatif de langage impératif (d'où son nom), c'est-à-dire dont les instructions modifient un état (la mémoire). On suppose fixé un ensemble dénombrable $\mathcal{V} = \{x, y, \ldots\}$ dont les éléments sont appelés *variables*. Les programmes sont constitués des trois catégories syntaxiques suivantes :
— les *expressions arithmétiques*

$$a \quad ::= \quad \underline{n} \quad | \quad x \quad | \quad a \texttt{ + } a' \quad | \quad a \texttt{ * } a'$$

où n est un entier quelconque (0, 1, 2, etc.),
— les *expressions booléennes*

$$b \quad ::= \quad \texttt{true} \quad | \quad \texttt{false} \quad | \quad a \texttt{ < } a' \quad | \quad \texttt{not } b$$

— les *commandes*

$$c \quad ::= \quad \texttt{skip} \quad | \quad x \texttt{ := } a \quad | \quad c\texttt{; } c' \quad | \quad \texttt{if } b \texttt{ then } c \texttt{ else } c' \quad | \quad \texttt{while } b \texttt{ do } c$$

Dans les définitions ci-dessus les lettres a et a' (resp. b et b', resp. c et c') dénotent toujours des expressions arithmétiques (resp. des expressions booléennes, resp. des commandes). On note respectivement **Aexp**, **Bexp** and **Cexp** les ensembles des expressions arithmétiques, des expressions booléennes et des commandes. Il s'agit ici de trois définitions inductives successives : on définit d'abord les expressions arithmétiques, puis les expressions booléennes, puis les commandes. Les noms de variables sont implicitement supposés distincts des autres expressions : `true` ou `3+2` ne sont pas des noms de variables valides.

1.2.2 Sémantique intuitive

Le langage permet de manipuler des variables stockées en mémoire et qui ne peuvent contenir que des entiers. La signification des expressions est la suivante :

- les expressions arithmétiques décrivent des entiers, soit donnant directement la notation $\underline{n}$ d'un entier n, soit en faisant référence au contenu d'une variable x, soit en calculant la somme a `+` a' ou le produit a `*` a' de deux expressions arithmétiques a et a',
- les expressions booléennes décrivent des booléens, soit en les donnant directement (`true`, `false`), soit en comparant par a `<` a' le résultat de deux expressions arithmétiques a et a', soit en calculant la négation `not` b d'une expression booléenne b,
- enfin les commandes représentent des instructions avec *effet de bord*, c'est-à-dire qu'elles modifient leur environnement (ici, la mémoire qui stocke le contenu des variables) : `skip` est l'instruction qui signifie « ne rien faire », x `:=` a signifie que l'on affecte le résultat de l'évaluation de l'expression arithmétique a à la variable x, c`;` c' signifie exécuter c puis c', et enfin `if` b `then` c `else` c' est un *branchement conditionnel* qui signifie qu'il faut exécuter soit c soit c' suivant que l'expression booléenne b s'évalue à vrai ou faux.

Le choix des opérations présentes dans les expressions arithmétiques et booléennes n'a rien de particulièrement canonique. Par souci concision, on se limite à deux opérations arithmétiques mais on pourrait en ajouter d'autres sans difficulté (soustraction, division, etc.), il en est de même pour les expressions booléenes (on pourrait rajouter l'égalité d'expressions arithmétiques, la conjonction d'expressions booléenes, etc.). Pour les commandes, la situation est un peu moins immédiate, et dépend de l'effet de bord considéré.

1.2.3 Quelques exemples

Exemple 6. Un exemple de programme dans ce langage est

```
if x < 2 * 2 then (x := 1; y := 2) else x := 0
```

Ici, on compare `x` à `2 * 2`, puis on prend la négation du résultat, c'est-à-dire qu'on cherche à savoir si on a `x` $\geqslant 4$. Si c'est le cas on assigne 1 à `x` et 2 à `y`, sinon on assigne 0 à `x`.

Exemple 7. Un exemple plus réaliste de programme est

```
y := 1; while not (x < 1) do (y := y * x; x := x - 1)
```

(on suppose ici qu'on a la soustraction dans les opérations arithmétiques). Celui-ci calcule, dans la variable `y`, la factorielle de la valeur initialement contenue dans `y` (lorsque celle-ci est strictement positive).

Exemple 8. Le programme suivant calcule le résultat de la division de `a` par `b` dans `q` (quotient) et `r` (reste) en utilisant l'algorithme d'Euclide:

```
q := 0; r := a; if not (r < b) then q := q + 1 else r := r - b
```

Exemple 9. Le programme suivant calcule le plus grand diviseur commun de `a` et `b` dans `a` en utilisant l'algorithme d'Euclide:

```
while not (a = b) do (if a < b then b := b - a else a := a - b)
```

1.3 Sémantique à grands pas

On chercher maintenant à définir la *sémantique* d'un programme, c'est-à-dire « ce qu'il fait ». Comme nous avons trois catégories syntaxiques, il nous faudra donner sucessivement leurs sémantiques respectives. En suivant ce qui a été fait en section 1.1, on pourrait tout d'abord penser définir une fonction $[\![-]\!] : \mathbf{Aexp} \to \mathbb{N}$ qui donne le résultat de l'évaluation d'une expression arithmétique. Mais on ne voit pas comment définir $[\![x]\!]$ dans le cas d'une variable x. En effet, celle-ci dépend de l'état courant de la mémoire, qu'il nous faut d'abord formaliser, puis ajouter en paramètre de l'évaluation.

1.3.1 Sémantique des expressions arithmétiques et booléennes

Un *état* d'un programme est l'ensemble des valeurs contenues dans les variables, que l'on suppose ici être toujours des entiers naturels : on peut le formaliser comme une fonction de $\mathcal{V}$ dans $\mathbb{N}$ et on note ici $\mathbf{Etats} = \mathbb{N}^{\mathcal{V}}$ l'ensemble des états. On définit l'*évaluation* d'une expression arithmétique au moyen d'un fonction

$$[\![-]\!]^{\mathbf{Aexp}} : \mathbf{Aexp} \times \mathbf{Etats} \to \mathbb{N}$$

qui, à une expression arithmétique a et un état σ associe un entier naturel, noté $[\![a]\!]^{\mathbf{Aexp}}_{\sigma}$, correspondant à l'évaluation de a dans l'état σ. Cette définition se fait par induction sur les expressions arithmétiques par

$$\begin{aligned}
[\![\underline{n}]\!]^{\mathbf{Aexp}}_{\sigma} &= n \\
[\![x]\!]^{\mathbf{Aexp}}_{\sigma} &= \sigma(x) \\
[\![a \texttt{ + } a']\!]^{\mathbf{Aexp}}_{\sigma} &= [\![a]\!]^{\mathbf{Aexp}}_{\sigma} + [\![a']\!]^{\mathbf{Aexp}}_{\sigma} \\
[\![a \texttt{ * } a']\!]^{\mathbf{Aexp}}_{\sigma} &= [\![a]\!]^{\mathbf{Aexp}}_{\sigma} \times [\![a']\!]^{\mathbf{Aexp}}_{\sigma}
\end{aligned}$$

Notons $\mathbb{B} = \{\top, \bot\}$ pour l'ensemble des booléens, où $\top$ correspond à vrai et $\bot$ à faux. On définit de même l'interprétation des expressions booléennes par une fonction

$$[\![-]\!]^{\mathbf{Bexp}} : \mathbf{Bexp} \times \mathbf{Etats} \to \mathbb{B}$$

pour laquelle on note $[\![b]\!]^{\mathbf{Bexp}}_{\sigma}$ l'image d'une paire (b, σ), définie par induction sur les expressions booléennes par

$$\begin{aligned}
[\![\texttt{true}]\!]^{\mathbf{Bexp}}_{\sigma} &= \top \\
[\![\texttt{false}]\!]^{\mathbf{Bexp}}_{\sigma} &= \bot \\
[\![a \texttt{ < } a']\!]^{\mathbf{Bexp}}_{\sigma} &= \begin{cases} \top & \text{si } [\![a]\!]^{\mathbf{Aexp}}_{\sigma} < [\![a']\!]^{\mathbf{Aexp}}_{\sigma} \\ \bot & \text{si } [\![a]\!]^{\mathbf{Aexp}}_{\sigma} \geqslant [\![a']\!]^{\mathbf{Aexp}}_{\sigma} \end{cases} \\
[\![\texttt{not } b]\!]^{\mathbf{Bexp}}_{\sigma} &= \begin{cases} \top & \text{is } [\![b]\!]^{\mathbf{Bexp}}_{\sigma} = \bot \\ \bot & \text{is } [\![b]\!]^{\mathbf{Bexp}}_{\sigma} = \top \end{cases}
\end{aligned}$$

1.3.2 Sémantique des commandes

Les commandes ne renvoient pas à proprement parler de résultat, on ne s'intéresse ici qu'à la façon dont elles modifient l'état courant, et on s'attend donc à pouvoir à spécifier leur évaluation par une fonction de type

$$\mathbf{Cexp} \times \mathbf{Etats} \to \mathbf{Etats}$$

qui étant donné une command et un état, renvoie l'état obtenu après exécution de cette commande. Il y a deux obstacles à cela. Le premier est que l'on ne s'attend pas à ce que cette fonction soit toujours définie, parce que certains programmes bouclent. Par exemple, on ne voit pas ce que pourrait signifier l'état *après* l'exécution du programme

$$\texttt{while true do } x \texttt{ := } x + 1$$

$$\frac{}{\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma}\ (\textsc{skip})$$

$$\frac{}{\langle\ x\ \texttt{:=}\ a \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma[x \mapsto [\![a]\!]_\sigma^{\mathbf{Aexp}}]}\ (\textsc{set})$$

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \langle\ c_2 \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}{\langle\ c_1\texttt{;}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}\ (\textsc{seq})$$

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}{\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}\ (\textsc{if}_\top) \qquad \text{si } [\![b]\!]_\sigma^{\mathbf{Bexp}} = \top$$

$$\frac{\langle\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}{\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}\ (\textsc{if}_\bot) \qquad \text{si } [\![b]\!]_\sigma^{\mathbf{Bexp}} = \bot$$

$$\frac{\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}\ (\textsc{while}_\top) \qquad \text{si } [\![b]\!]_\sigma^{\mathbf{Bexp}} = \top$$

$$\frac{}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma}\ (\textsc{while}_\bot) \qquad \text{si } [\![b]\!]_\sigma^{\mathbf{Bexp}} = \bot$$

FIGURE 1 – Règles de l'évaluation des commandes.

puisque celui-ci ne s'arrête jamais. Le second est d'ordre méthodologique. On pourrait spécifier le résultat de l'exécution comme une fonction partielle

$$\mathbf{Cexp} \times \mathbf{Etats} \rightharpoonup \mathbf{Etats}$$

mais il est plus naturel de spécifier le résultat de l'exécution d'un programme donné par une série de règles, et d'en déduire a posteriori que le résultat est défini de façon unique quand il l'est. On va donc définir l'évaluation $\mathcal{E}$ comme une relation entre des commandes, des états et des états, c'est-à-dire comme un sous-ensemble

$$\mathcal{E} \subseteq \mathbf{Cexp} \times \mathbf{Etats} \times \mathbf{Etats}$$

On note

$$\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$$

un triplet (c, σ, σ') dans $\mathcal{E}$, qui doit se lire comme « partant de l'état σ, l'évaluation de la commande c peut aboutir à l'état σ' ». On va définir cet ensemble à l'aide d'un ensembles de règles de la forme

$$\frac{\langle\ c_1 \mid \sigma_1\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma_1' \qquad \ldots \qquad \langle\ c_n \mid \sigma_n\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma_n'}{\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}\ (\textsc{nom})$$

où les $\langle\ c_i \mid \sigma_i\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma_i'$ sont appelées *prémisses* et $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$ la *conclusions*. Celles-ci signifient qu'on définit $\mathcal{E}$ comme le plus petit ensemble tel que, pour chacune des règles telles que ci-dessus, si $(c_i, \sigma_i, \sigma_i') \in \mathcal{E}$ pour tout i avec $1 \leqslant i \leqslant n$, alors $(c, \sigma, \sigma') \in \mathcal{E}$. Ou de façon plus concise : à chaque fois qu'on a toutes les hypothèses au dessus de la barre, on peut en déduire la conclusion en dessous de la barre. L'annotation (NOM) permet de donner un nom à chacune des règles afin de les identifier. Ces règles sont données à la figure 10. Dans la seconde règle, la notation $\sigma[x \mapsto n]$ dénote, étant donnés un état σ, une variable x et un entier n, l'état tel que

$$\sigma[x \mapsto n](y) = \begin{cases} n & \text{si } x = y, \\ \sigma(y) & \text{si } x \neq y. \end{cases}$$

Les quatre dernières règles sont accompagnées d'une condition (sur la valeur de $[\![b]\!]_\sigma^{\mathbf{Bexp}}$) indiquée à droite, restreignant leur utilisation aux cas où cette condition est satisfaite. On peut constater que ces règles formalisent notre intuition :

— `skip` ne change pas l'état,
— x `:=` a met le résultat de l'évaluation de a dans x,
— c`;` c' revient à exécuter c puis c' partant de l'état obtenu,
— `if` b `then` c `else` c' revient à exécuter c ou c' suivant le résultat de l'évaluation de b,
— `while` b `do` c revient à exécuter c puis `while` b `do` c, ou bien à ne rien faire, suivant le résultat de l'évaluation de b.

Notons que l'intérêt de définir cette relation comme une plus petite relation close par les règles de déduction : on ne pourrait pas faire cette définition par induction structurelle sur le programme, car le programme en conclusion de la règle (WHILE$_\top$) apparaît aussi en prémisse !

Lorsque l'on a $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$, on peut le justifier par un arbre, formé des règles ci-dessus, dont la racine (figurée en bas) est $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$. Par exemple, dans un état σ tel que $\sigma(y) = 1$, on peut montrer que le programme

$$\texttt{if } y \texttt{ < } 2 \texttt{ then (skip; } z \texttt{ := } 5\texttt{) else skip}$$

arrive dans l'état σ' tel que $\sigma'(z) = 5$ et $\sigma'(x) = \sigma(x)$ pour $x \neq z$, en utilisant l'arbre de dérivation suivant :

$$\dfrac{\dfrac{\dfrac{}{\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma}\ (\textsc{skip}) \qquad \dfrac{}{\langle\ z \texttt{ := } 5 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma[z \mapsto 5]}\ (\textsc{set})}{\langle\ \texttt{skip; } z \texttt{ := } 5 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma[z \mapsto 5]}\ (\textsc{seq})}{\langle\ \texttt{if } y \texttt{ < } 2 \texttt{ then (skip; } z \texttt{ := } 5\texttt{) else skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma[z \mapsto 5]}\ (\textsc{if}_\top)$$

Le *principe d'induction sur les dérivations* suivant est souvent utile pour raisonner sur celles-ci. Celui-ci formalise l'intuition suivante. Supposons donnée une propriété P sur les évaluations telles que pour toute règle de la figure 10, si P est vérifiée pour toute hypothèse alors elle est aussi vérifiée pour la conclusion. Alors elle est vraie pour toute évaluation. En effet, cette dernière admet une dérivation, et on peut voir que toutes les évaluations qui apparaissent dans cette évaluation sont vérifiées de proche en proche, en partant des feuilles puis en allant vers la racine.

Théorème 10 (Induction sur les dérivations)**.** *Considérons une propriété $P(\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$ sur les évaluations. Supposons que pour chacune des règles*

$$\dfrac{\langle\ c_1 \mid \sigma_1\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'_1 \qquad \ldots \qquad \langle\ c_n \mid \sigma_n\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'_n}{\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}$$

de la figure 10, si $P(\langle\ c_i \mid \sigma_i\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'_i)$ est valide pour tout i avec $1 \leqslant i \leqslant n$ alors $P(\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$. Alors $P(\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$ est valide pour toute évaluation $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$.

1.3.3 Déterminisme de l'évaluation

On souhaite maintenant montrer que, lorsqu'un programme renvoit un résultat dans un état donné, celui-ci est toujours le même : on dit que l'évaluation est *déterministe*. Ce résultat est crucial puisqu'il formalise le fait que la notion de résultat d'un programme est bien définie (un programme renvoie au plus un résultat).

Théorème 11 (Déterminisme)**.** *L'évaluation des commandes est déterministe : étant donnés une commande c et trois états σ, σ' et σ'', si $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$ et $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''$ alors $\sigma' = \sigma''$.*

Démonstration. Supposons donnée une dérivation π de $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$. On raisonne par induction sur π pour montrer la propriété $P(\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$ suivante :

> pour toute dérivation π' de $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''$ de la commande c dans l'état σ, on a $\sigma' = \sigma''$.

On traite séparément chacune des règles, en ne détaillant que quelques cas significatifs.

— Cas de la règle

$$\dfrac{}{\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma}\ (\textsc{skip})$$

Montrons la propriété $P(\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma)$. Considérons une évaluation $\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''$. Celle-ci admet une dérivation qui se termine par une règle de la forme

$$\frac{\cdots}{\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}$$

Cette règle est nécessairement (SKIP), car c'est la seule qui permette de dériver l'évaluation d'une commande `skip`. On en déduit $\sigma'' = \sigma$.

— Cas de la règle

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}{\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}\ (\text{IF}_\top)$$

Supposons que $[\![b]\!]_\sigma^{\mathbf{Bexp}} = \top$ (car c'est la condition de l'utilisation de cette règle) et que $P(\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$ est satisfaite et montrons la propriété

$$P(\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$$

Considérons une évaluation

$$\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''$$

celle-ci admet une dérivation qui se termine par une règle de la forme

$$\frac{\cdots}{\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}$$

cette règle est nécessairement ($\text{IF}_\top$) ou ($\text{IF}_\bot$) car la commande en conclusion est de la forme `if ... then ... else ...`, et ça ne peut être que ($\text{IF}_\top$), car on a supposé $[\![b]\!]_\sigma^{\mathbf{Bexp}} = \top$. Cette dérivation est donc de la forme

$$\frac{\dfrac{\vdots}{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}}{\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}\ (\text{IF}_\top)$$

On en déduit $\sigma' = \sigma''$ en utilisant $P(\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$.

— Cas de la règle

$$\frac{\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma_1' \qquad \langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma_1'\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}\ (\text{WHILE}_\top)$$

Supposons $[\![b]\!]_\sigma^{\mathbf{Bexp}} = \top$ et que

$$P(\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma_1') \qquad \text{et} \qquad P(\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma_1'\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$$

sont satisfaites. Considérons une évaluation $\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''$, son arbre de dérivation est nécessairement de la forme

$$\frac{\dfrac{\vdots}{\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma_1''} \qquad \dfrac{\vdots}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma_1''\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma''}\ (\text{WHILE}_\top)$$

En utilisant $P(\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma_1')$ on montre $\sigma_1' = \sigma_1''$, puis en utilisant $P(\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma_1'\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma')$ on déduit finalement $\sigma' = \sigma''$.

— Les autres règles se traitent de façon similaire. □

Le théorème ci-dessus indique que l'on peut, a posteriori, définir une fonction partielle

$$\mathbf{Cexp} \times \mathbf{Etats} \rightharpoonup \mathbf{Etats}$$

qui à une commande c et un état σ associe l'état σ' tel que $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$ (et n'est pas définie sinon).

1.3.4 Équivalence observationnelle

Il est naturel de considérer que deux programmes sont équivalents lorsqu'ils donnent les mêmes résultats dans les mêmes états, c'est-à-dire qu'on ne peut les distinguer. Formellement, on définit la relation d'*équivalence observationnelle* $\sim$ sur les commandes par $c_1 \sim c_2$ si pour tous états σ et σ' on a

$$\langle\, c_1 \mid \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \text{si et seulement si} \qquad \langle\, c_2 \mid \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$$

On peut par exemple montrer :

Proposition 12. *Pour toute commande c, on a*

$$\texttt{if true then } c \texttt{ else skip} \sim c$$

Démonstration. Par inspection des règles, on montre que

$$\langle\, \texttt{if true then } c \texttt{ else skip} \mid \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$$

est équivalent à avoir une dérivation de la forme

$$\dfrac{\dfrac{\vdots}{\langle\, c \mid \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}}{\langle\, \texttt{if } b \texttt{ then } c \texttt{ else skip} \mid \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'}\ (\textsc{if}_\top)$$

ce qui est équivalent à avoir $\langle\, c \mid \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma'$ dérivable. □

Exercice 13. Montrer $\texttt{while } b \texttt{ do } c \sim \texttt{if } b \texttt{ then } (c\texttt{; while } b \texttt{ do } c) \texttt{ else skip}$.

1.4 Sémantique à petits pas

La sémantique à grands pas présentée à la section précédente associe, à une commande, l'état résultant de son exécution dans son intégralité. Celles-ci ne permet donc pas de raisonner sur les exécutions elles-mêmes, mais seulement leurs résultats. En particulier, elles ne permettent pas de raisonner sur les programmes qui ne terminent pas, puisque ceux-ci ne mènent pas à un résultat. Pourtant il existe des programmes intéressants qui ne terminent pas, par exemple un serveur web est toujours en attente d'une connexion de la part d'un client et ne renvoit jamais de résultat, il fait néanmoins quelque chose d'utile.

1.4.1 Définition de la réduction

Afin de palier à cette limitation, on peut utiliser la *sémantique opérationnelle à petits pas*, qui décrit l'effet de l'exécution de l'une des instructions d'une commande. Le prix à payer est bien sûr que nous décomposons maintenant l'évaluation en une suite de petites étapes.

Cette sémantique est formalisée comme un sous-ensemble de

$$\mathbf{Cexp} \times \mathbf{Etats} \times \mathbf{Cexp} \times \mathbf{Etats}$$

autrement dit d'une relation sur $\mathbf{Cexp} \times \mathbf{Etats}$, dont les éléments (c, σ, c', σ') sont notés

$$\langle\, c \mid \sigma \,\rangle \longrightarrow \langle\, c' \mid \sigma' \,\rangle$$

Un tel élément peut être lu comme « dans l'état σ, la commande c va commencer par exécuter une instruction qui aboutit à l'état σ', après quoi la commande c' reste encore à exécuter ». Autrement dit, on axiomatise ici la *réduction* des commandes (et non plus leur évaluation). On commence d'abord par de même formaliser la réduction des expressions arithmétiques et booléennes respectivement comme des sous-ensembles de

$$\mathbf{Aexp} \times \mathbf{Etats} \times \mathbf{Aexp} \times \mathbf{Etats} \qquad\qquad \mathbf{Bexp} \times \mathbf{Etats} \times \mathbf{Bexp} \times \mathbf{Etats}$$

$$\frac{}{\langle\ x \mid \sigma\ \rangle \longrightarrow \langle\ \underline{\sigma(x)} \mid \sigma\ \rangle}$$

$$\frac{\langle\ a_1 \mid \sigma\ \rangle \longrightarrow \langle\ a_1' \mid \sigma'\ \rangle}{\langle\ a_1\ \texttt{+}\ a_2 \mid \sigma\ \rangle \longrightarrow \langle\ a_1'\ \texttt{+}\ a_2 \mid \sigma'\ \rangle} \qquad \frac{\langle\ a_1 \mid \sigma\ \rangle \longrightarrow \langle\ a_1' \mid \sigma'\ \rangle}{\langle\ a_1\ \texttt{*}\ a_2 \mid \sigma\ \rangle \longrightarrow \langle\ a_1'\ \texttt{*}\ a_2 \mid \sigma'\ \rangle}$$

$$\frac{\langle\ a \mid \sigma\ \rangle \longrightarrow \langle\ a' \mid \sigma'\ \rangle}{\langle\ \underline{n}\ \texttt{+}\ a \mid \sigma\ \rangle \longrightarrow \langle\ \underline{n}\ \texttt{+}\ a' \mid \sigma'\ \rangle} \qquad \frac{\langle\ a \mid \sigma\ \rangle \longrightarrow \langle\ a' \mid \sigma'\ \rangle}{\langle\ \underline{n}\ \texttt{*}\ a \mid \sigma\ \rangle \longrightarrow \langle\ \underline{n}\ \texttt{*}\ a' \mid \sigma'\ \rangle}$$

$$\frac{}{\langle\ \underline{m}\ \texttt{+}\ \underline{n} \mid \sigma\ \rangle \longrightarrow \langle\ \underline{m+n} \mid \sigma\ \rangle} \qquad \frac{}{\langle\ \underline{m}\ \texttt{*}\ \underline{n} \mid \sigma\ \rangle \longrightarrow \langle\ \underline{m \times n} \mid \sigma\ \rangle}$$

FIGURE 2 – Règles de réduction des expressions arithmétiques.

$$\frac{\langle\ a_1 \mid \sigma\ \rangle \longrightarrow \langle\ a_1' \mid \sigma'\ \rangle}{\langle\ a_1\ \texttt{<}\ a_2 \mid \sigma\ \rangle \longrightarrow \langle\ a_1'\ \texttt{<}\ a_2 \mid \sigma'\ \rangle}$$

$$\frac{\langle\ a \mid \sigma\ \rangle \longrightarrow \langle\ a' \mid \sigma'\ \rangle}{\langle\ \underline{n}\ \texttt{<}\ a \mid \sigma\ \rangle \longrightarrow \langle\ \underline{n}\ \texttt{<}\ a' \mid \sigma'\ \rangle} \qquad \frac{\langle\ b \mid \sigma\ \rangle \longrightarrow \langle\ b' \mid \sigma'\ \rangle}{\langle\ \texttt{not}\ b \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{not}\ b' \mid \sigma'\ \rangle}$$

$$\frac{}{\langle\ \underline{m}\ \texttt{<}\ \underline{n} \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{true} \mid \sigma\ \rangle} \quad \text{si } m < n \qquad \frac{}{\langle\ \texttt{not false} \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{true} \mid \sigma\ \rangle}$$

$$\frac{}{\langle\ \underline{m}\ \texttt{<}\ \underline{n} \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{false} \mid \sigma\ \rangle} \quad \text{si } m \geqslant n \qquad \frac{}{\langle\ \texttt{not true} \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{false} \mid \sigma\ \rangle}$$

FIGURE 3 – Règles de réduction des expressions booléennes.

$$\frac{\langle\ a \mid \sigma\ \rangle \longrightarrow \langle\ a' \mid \sigma'\ \rangle}{\langle\ x\ \texttt{:=}\ a \mid \sigma\ \rangle \longrightarrow \langle\ x\ \texttt{:=}\ a' \mid \sigma'\ \rangle}\ (\text{SET})$$

$$\frac{}{\langle\ x\ \texttt{:=}\ \underline{n} \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{skip} \mid \sigma[x \mapsto n]\ \rangle}\ (\text{SET}_n)$$

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow \langle\ c_1' \mid \sigma'\ \rangle}{\langle\ c_1\texttt{;}\ c_2 \mid \sigma\ \rangle \longrightarrow \langle\ c_1'\texttt{;}\ c_2 \mid \sigma'\ \rangle}\ (\text{SEQ}_\text{G}) \qquad \frac{}{\langle\ \texttt{skip;}\ c \mid \sigma\ \rangle \longrightarrow \langle\ c \mid \sigma\ \rangle}\ (\text{SEQ}_\text{D})$$

$$\frac{\langle\ b \mid \sigma\ \rangle \longrightarrow \langle\ b' \mid \sigma'\ \rangle}{\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{if}\ b'\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma'\ \rangle}\ (\text{IF})$$

$$\frac{}{\langle\ \texttt{if true then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow \langle\ c_1 \mid \sigma\ \rangle}\ (\text{IF}_\top)$$

$$\frac{}{\langle\ \texttt{if false then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \longrightarrow \langle\ c_2 \mid \sigma\ \rangle}\ (\text{IF}_\bot)$$

$$\frac{}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{if}\ b\ \texttt{then}\ (c\texttt{; while}\ b\ \texttt{do}\ c)\ \texttt{else skip} \mid \sigma\ \rangle}\ (\text{WHILE})$$

FIGURE 4 – Règles de réduction des commandes.

en utilisant une notation similaire pour leur éléments. Comme précédemment, on définit ces ensembles par un ensemble de règles d'inférence, données à la figure 2 pour les expressions arithmétiques, à la figure 3 pour les expressions booléeenes, et à la figure 4 pour les commandes. Celles ci-décrivent comment évaluer les expressions arithmétiques et booléennes : on commence par évaluer celles-ci par la gauche dans le cas d'opérations binaires. Elles décrivent aussi où exécuter une instruction dans une commande :

— (SET) : lorsqu'on a une instruction d'assignation, on l'exécute,
— (SEQ_G) : dans une commande c_1 ; c_2, on exécute une instruction de c_1,
— (SEQ_D) : dans une commande c_1 ; c_2, lorsqu'il n'y a plus d'instruction à exécuter dans c_1, on exécute c_2,
— (IF_G) : dans une commande `if` b `then` c `else` c', on réduit b si possible, s'il est vrai on exécute c, s'il est faux on exécute c',
— (WHILE_D): dans une comande `while` b `do` c, si b est vrai on exécute c puis `while` b `do` c, sinon on s'arrête.

Notons que lorsqu'une commande c est entièrement exécutée, elle donne lieu à une réduction de la forme

$$\langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{skip} \mid \sigma'\ \rangle$$

où on utilise `skip` pour indiquer qu'il n'y a plus rien à exécuter.

Remarque 14. La règle pour les boucles `while` n'est pas celle à laquelle on pourrait s'attendre au premier abord. Pour comprendre cette formulation, remarquons qu'une règle de la forme

$$\frac{\langle\ b \mid \sigma\ \rangle \longrightarrow \langle\ b' \mid \sigma'\ \rangle}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{while}\ b'\ \texttt{do}\ c \mid \sigma\ \rangle}$$

ne conviendrait pas, car il faut pouvoir évaluer intégralement b à chaque tour de boucle et non pas une seule fois.

Une *suite de réductions* est une paire de suites finies $(c_i)_{1\leqslant i\leqslant n}$ de commandes et $(\sigma_i)_{1\leqslant i\leqslant n}$ d'états, toutes deux de même longueur n, telles que l'on ait $c_1 = c$, $\sigma_1 = \sigma$, $c_n = c'$, σ', et $\langle\ c_i \mid \sigma_i\ \rangle \longrightarrow \langle\ c_{i+1} \mid \sigma_{i+1}\ \rangle$ pour tout i avec $1 \leqslant i < n$, c'est-à-dire

$$\langle\ c_1 \mid \sigma_1\ \rangle \to \langle\ c_2 \mid \sigma_2\ \rangle \to \langle\ c_3 \mid \sigma_3\ \rangle \to \ldots \to \langle\ c_n \mid \sigma_n\ \rangle$$

On écrit parfois

$$\langle\ c_1 \mid \sigma_1\ \rangle \xrightarrow{*} \langle\ c_n \mid \sigma_n\ \rangle$$

pour indiquer qu'il existe une telle suite de réductions. On appelle n la *longueur* de la suite de réductions. Il est courant de raisonner par récurrence sur cette longueur pour montrer des propriétés sur les réductions.

Exemple 15. Partant d'un état σ_0 avec $\sigma_0(\texttt{x}) = 3$. L'exécution du programme factorielle donné en théorème 7 commence de la façon suivante :

$$\begin{aligned}
& \langle\ \texttt{y := 1; while not (x < 1) do (y := y * x; x := x - 1)} \mid \sigma_0\ \rangle \\
\longrightarrow\ & \langle\ \texttt{skip; while not (x < 1) do (y := y * x; x := x - 1)} \mid \sigma_1\ \rangle && \text{avec } \sigma_1 = \sigma_0[\texttt{y} \mapsto 1] \\
\longrightarrow\ & \langle\ \texttt{while not (x < 1) do (y := y * x; x := x - 1)} \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{if not (x < 1) then (y := y * x; x := x - 1);}\ W\ \texttt{else skip} \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{if not (3 < 1) then (y := y * x; x := x - 1);}\ W\ \texttt{else skip} \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{if not false then (y := y * x; x := x - 1);}\ W\ \texttt{else skip} \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{if true then (y := y * x; x := x - 1);}\ W\ \texttt{else skip} \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{(y := y * x; x := x - 1);}\ W \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{(y := 1 * x; x := x - 1);}\ W \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{(y := 1 * 3; x := x - 1);}\ W \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{(y := 3; x := x - 1);}\ W \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{(skip; x := x - 1);}\ W \mid \sigma_1\ \rangle \\
\longrightarrow\ & \langle\ \texttt{x := x - 1;}\ W \mid \sigma_2\ \rangle && \text{avec } \sigma_2 = \sigma_1[\texttt{y} \mapsto 3] \\
\longrightarrow\ & \langle\ \texttt{x := 3 - 1;}\ W \mid \sigma_2\ \rangle \\
\longrightarrow\ & \langle\ \texttt{x := 2;}\ W \mid \sigma_2\ \rangle \\
\longrightarrow\ & \langle\ \texttt{while not (x < 1) do (y := y * x; x := x - 1)} \mid \sigma_3\ \rangle && \text{avec } \sigma_3 = \sigma_2[\texttt{x} \mapsto 1] \\
\longrightarrow\ & \ldots
\end{aligned}$$

où W est une abbréviation pour

$$\texttt{while not (x < 1) do (y := y * x; x := x - 1)}$$

Une commande c est *réductible* dans un état σ s'il est existe une commande c' et un état σ' tels que $\langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ c' \mid \sigma'\ \rangle$, et *irréductible* sinon. Cette propriété peut être caractérisée de la façon suivante :

Lemme 16. *Étant donnés une une commande c et un état σ, on a équivalence entre*

(i) *c est irréductible dans l'état σ,*

(ii) *$c = \texttt{skip}$.*

Démonstration. L'implication (ii) vers (i) est immédiate car il n'existe pas de règle permettant de réduire la commande `skip`. Pour l'implication (i) vers (ii), on considère toutes les formes possibles pour c et on montre que dans chacun des cas, à l'exception de $c = \texttt{skip}$, il existe une réduction possible, par induction sur c.

- Si $c = x\ \texttt{:=}\ a$, alors c est réductible par (SET).
- Si $c = c_1\texttt{;}\ c_2$, alors on distingue deux cas.
  - Si $c_1 \neq \texttt{skip}$ alors c_1 est réductible par hypothèse d'induction et donc c est réductible par (SEQ_G).
  - Si $c_1 = \texttt{skip}$ alors c est réductible par (SEQ_D).
- Les autres cas sont laissés en exercice. Il requièrent de d'abord montrer les lemmes auxiliaires suivants :
  - une expression arithmétique est irréductible si et seulement si elle est de la forme $\underline{n}$,
  - une expression booléenne est réductible si et seulement si elle est de le forme `true` ou `false`. □

1.4.2 Équivalence avec la réduction à grands pas

Au cours de la réduction, on peut considérer qu'on a obtenu un résultat lorsque l'on a atteint une commande irréductible. Partant de ce point de vue, on peut montrer que les sémantiques à grand pas et à petit pas coïncident (la preuve utilise des lemmes auxiliaires qui sont montrés par la suite) :

Théorème 17. *Pour toute commande c et état σ et σ', on a équivalence entre*

(i) *$\langle\ c \mid \sigma\ \rangle \twoheadrightarrow \sigma'$ (dans la sémantique à grands pas),*

(ii) *$\langle\ c \mid \sigma\ \rangle \xrightarrow{*} \langle\ \texttt{skip} \mid \sigma'\ \rangle$ (dans la sémantique à petits pas).*

Démonstration. L'implication (i) vers (ii) se montre par induction sur la dérivation de $\langle\ c \mid \sigma\ \rangle \twoheadrightarrow \sigma'$.

- Si la dernière règle est (SKIP), on conclut immédiatement.
- Si la dernière règle est

$$\frac{}{\langle\ x\ \texttt{:=}\ a \mid \sigma\ \rangle \twoheadrightarrow \sigma[x \mapsto [\![a]\!]^{\mathbf{Aexp}}_\sigma]}\ (\text{SET})$$

  Par le théorème 18, on a

$$\langle\ a \mid \sigma\ \rangle \xrightarrow{*} \langle\ \underline{[\![a]\!]^{\mathbf{Aexp}}_\sigma} \mid \sigma\ \rangle$$

  et donc, par le théorème 20,

$$\langle\ x\ \texttt{:=}\ a \mid \sigma\ \rangle \xrightarrow{*} \langle\ x\ \texttt{:=}\ \underline{[\![a]\!]^{\mathbf{Aexp}}_\sigma} \mid \sigma\ \rangle$$

  On conclut avec la règle (SET_n).
- Si la dernière règle est

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \twoheadrightarrow \sigma' \qquad \langle\ c_2 \mid \sigma'\ \rangle \twoheadrightarrow \sigma''}{\langle\ c_1\texttt{;}\ c_2 \mid \sigma\ \rangle \twoheadrightarrow \sigma''}\ (\text{SEQ})$$

  Par hypothèse d'induction, on a

$$\langle\ c_1 \mid \sigma\ \rangle \xrightarrow{*} \langle\ \texttt{skip} \mid \sigma'\ \rangle \qquad\qquad \langle\ c_2 \mid \sigma'\ \rangle \xrightarrow{*} \langle\ \texttt{skip} \mid \sigma''\ \rangle$$

Le théorème 20 nous permet de déduire du premier que l'on a aussi

$$\langle\ c_1\texttt{;}\ \ c_2 \mid \sigma\ \rangle \longrightarrow^* \langle\ \texttt{skip;}\ \ c_2 \mid \sigma'\ \rangle$$

et l'on conclut avec la suite de réductions

$$\langle\ c_1\texttt{;}\ \ c_2 \mid \sigma\ \rangle \longrightarrow^* \langle\ \texttt{skip;}\ \ c_2 \mid \sigma'\ \rangle \overset{(\text{SEQ}_\text{D})}{\longrightarrow} \langle\ c_2 \mid \sigma'\ \rangle \longrightarrow^* \langle\ \texttt{skip} \mid \sigma''\ \rangle$$

— Si la dernière règle est

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma'}{\langle\ \texttt{if}\ \ b\ \ \texttt{then}\ \ c_1\ \ \texttt{else}\ \ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma'}\ (\text{IF}_\top)$$

où $[\![b]\!]^{\mathbf{Bexp}}_\sigma = \top$. Par hypothèse d'induction, on a

$$\langle\ c_1 \mid \sigma\ \rangle \longrightarrow^* \langle\ \texttt{skip} \mid \sigma'\ \rangle$$

Par le théorème 18, on a

$$\langle\ b \mid \sigma\ \rangle \longrightarrow^* \langle\ \texttt{true} \mid \sigma\ \rangle$$

et donc

$$\langle\ \texttt{if}\ \ b\ \ \texttt{then}\ \ c_1\ \ \texttt{else}\ \ c_2 \mid \sigma\ \rangle \longrightarrow^* \langle\ \texttt{if true then}\ \ c_1\ \ \texttt{else}\ \ c_2 \mid \sigma\ \rangle$$

par le théorème 20. On a donc la suite de réductions

$$\begin{aligned}\langle\ \texttt{if}\ \ b\ \ \texttt{then}\ \ c_1\ \ \texttt{else}\ \ c_2 \mid \sigma\ \rangle &\longrightarrow^* \langle\ \texttt{if true then}\ \ c_1\ \ \texttt{else}\ \ c_2 \mid \sigma\ \rangle \\ &\overset{(\text{SEQ}_\text{G})}{\longrightarrow} \langle\ c_1 \mid \sigma\ \rangle \\ &\longrightarrow^* \langle\ \texttt{skip} \mid \sigma'\ \rangle\end{aligned}$$

— Les autres cas sont laissés au lecteur.

L'implication (ii) vers (i) se montre par récurrence sur la longueur de la dérivation

$$\langle\ c \mid \sigma\ \rangle \longrightarrow^* \langle\ \texttt{skip} \mid \sigma_\infty\ \rangle$$

(on note ici σ_∞ l'état final de la réduction à petits pas). Si cette longueur est 0, on a $c = \texttt{skip}$ et le résultat suit immédiatement. Sinon, on a chemin de réductions de la forme

$$\langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ c' \mid \sigma'\ \rangle \longrightarrow^* \langle\ \texttt{skip} \mid \sigma_\infty\ \rangle$$

où $\langle\ c' \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty$ par hypothèse de récurrence. Il nous suffit donc de montrer que

si $\langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ c' \mid \sigma'\ \rangle$ et $\langle\ c' \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty$ alors $\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty$.

On montre cette propriété par induction sur la dérivation de $\langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ c' \mid \sigma'\ \rangle$.

— Si c'est

$$\frac{\langle\ a \mid \sigma\ \rangle \longrightarrow \langle\ a' \mid \sigma'\ \rangle}{\langle\ x\ \ \texttt{:=}\ \ a \mid \sigma\ \rangle \longrightarrow \langle\ x\ \ \texttt{:=}\ \ a' \mid \sigma'\ \rangle}\ (\text{SET})$$

alors on a

$$\langle\ x\ \ \texttt{:=}\ \ a \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma[x \mapsto [\![a]\!]^{\mathbf{Aexp}}_\sigma] \qquad \langle\ x\ \ \texttt{:=}\ \ a' \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma'[x \mapsto [\![a']\!]^{\mathbf{Aexp}}_{\sigma'}]$$

et le théorème 19 nous assure $\sigma = \sigma'$ et $[\![a]\!]^{\mathbf{Aexp}}_\sigma = [\![a]\!]^{\mathbf{Aexp}}_{\sigma'}$ ce qui nous permet de conclure.

— Si c'est (SET_n), c'est immédiat par (SET).

— Si c'est

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow \langle\ c_1' \mid \sigma'\ \rangle}{\langle\ c_1\texttt{;}\ \ c_2 \mid \sigma\ \rangle \longrightarrow \langle\ c_1'\texttt{;}\ \ c_2 \mid \sigma'\ \rangle}\ (\text{SEQ}_\text{G})$$

L'évaluation $\langle\ c_1'\texttt{;}\ \ c_2 \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty$ est nécessairement de la forme

$$\frac{\langle\ c_1' \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma'' \qquad \langle\ c_2 \mid \sigma''\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}{\langle\ c_1'\texttt{;}\ \ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}\ (\text{SEQ})$$

Comme $\langle\ c_1' \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma''$, par hypothèse d'induction, on a aussi $\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma''$ et on en déduit

$$\frac{\langle\ c_1 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma'' \qquad \langle\ c_2 \mid \sigma''\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}{\langle\ c_1\texttt{;}\ \ c_2 \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}\ (\text{SEQ})$$

— Si c'est

$$\frac{}{\langle\ \texttt{skip};\ c \mid \sigma\ \rangle \longrightarrow \langle\ c \mid \sigma\ \rangle}\ (\text{SEQ}_\text{D})$$

alors on a immédiatement

$$\frac{\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma \qquad \langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}{\langle\ \texttt{skip};\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}\ (\text{SEQ})$$

— Les cas (IF), (IF$_\top$) et (IF$_\bot$) sont laissés au lecteur.
— Si c'est

$$\frac{}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow \langle\ \texttt{if}\ b\ \texttt{then}\ (c;\ \texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else skip} \mid \sigma\ \rangle}\ (\text{WHILE})$$

On distingue selon les valeurs de $[\![b]\!]_\sigma^{\mathbf{Bexp}}$.
— Cas $[\![b]\!]_\sigma^{\mathbf{Bexp}} = \top$. L'évaluation du membre de droite vers σ_∞ est nécessairement de la forme

$$\frac{\dfrac{\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma' \qquad \langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}{\langle\ c;\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}\ (\text{SEQ})}{\langle\ \texttt{if}\ b\ \texttt{then}\ (c;\ \texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}\ (\text{IF}_\top)$$

et on conclut par

$$\frac{\langle\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma' \qquad \langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma'\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}{\langle\ \texttt{while}\ b\ \texttt{do}\ c \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma_\infty}\ (\text{WHILE}_\top)$$

— Cas $[\![b]\!]_\sigma^{\mathbf{Bexp}} = \bot$. L'évaluation du membre de droite vers σ_∞ est nécessairement de la forme

$$\frac{\langle\ \texttt{skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma}{\langle\ \texttt{if}\ b\ \texttt{then}\ (c;\ \texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else skip} \mid \sigma\ \rangle \longrightarrow\!\!\!\!\rightarrow \sigma}\ (\text{IF}_\bot)$$

et on conclut par (WHILE$_\bot$). □

Au cours de la preuve ci-dessus, nous avons utilisé les lemmes suivants qui se montrent indépendemment.

Lemme 18. *Soit σ un état :*
— *étant donnée une expression arithmétique a, on a*

$$\langle\ a \mid \sigma\ \rangle \xrightarrow{*} \langle\ \underline{[\![a]\!]_\sigma^{\mathbf{Aexp}}} \mid \sigma\ \rangle$$

— *étant donnée une expression booléenne b, on a*

$$\langle\ b \mid \sigma\ \rangle \xrightarrow{*} \langle\ \underline{[\![b]\!]_\sigma^{\mathbf{Bexp}}} \mid \sigma\ \rangle$$

avec $\underline{\top} = \texttt{true}$ et $\underline{\bot} = \texttt{false}$.

Démonstration. Par induction sur a (ou b). □

Lemme 19. *On a:*
— *si $\langle\ a \mid \sigma\ \rangle \longrightarrow \langle\ a' \mid \sigma'\ \rangle$ alors $\sigma = \sigma'$ et $[\![a]\!]_\sigma^{\mathbf{Aexp}} = [\![a']\!]_{\sigma'}^{\mathbf{Aexp}}$,*
— *si $\langle\ b \mid \sigma\ \rangle \longrightarrow \langle\ b' \mid \sigma'\ \rangle$ alors $\sigma = \sigma'$ et $[\![b]\!]_\sigma^{\mathbf{Bexp}} = [\![b']\!]_{\sigma'}^{\mathbf{Bexp}}$.*

Démonstration. Par induction sur la dérivation de $\langle\ a \mid \sigma\ \rangle \longrightarrow \langle\ a' \mid \sigma'\ \rangle$ (ou $\langle\ b \mid \sigma\ \rangle \longrightarrow \langle\ b' \mid \sigma'\ \rangle$). □

Lemme 20. *Les suites de réductions sont compatibles avec le contexte :*

$$\begin{array}{rcl}
\langle\ a \mid \sigma\ \rangle \xrightarrow{*} \langle\ a' \mid \sigma'\ \rangle & \textit{implique} & \langle\ x\ \texttt{:=}\ a \mid \sigma\ \rangle \xrightarrow{*} \langle\ x\ \texttt{:=}\ a' \mid \sigma'\ \rangle \\
\langle\ c_1 \mid \sigma\ \rangle \xrightarrow{*} \langle\ c_1' \mid \sigma'\ \rangle & \textit{implique} & \langle\ c_1;\ c_2 \mid \sigma\ \rangle \xrightarrow{*} \langle\ c_1';\ c_2 \mid \sigma'\ \rangle \\
\langle\ b \mid \sigma\ \rangle \xrightarrow{*} \langle\ b' \mid \sigma'\ \rangle & \textit{implique} & \\
\multicolumn{3}{c}{\langle\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma\ \rangle \xrightarrow{*} \langle\ \texttt{if}\ b'\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \sigma'\ \rangle}
\end{array}$$

Démonstration. Par récurrence sur la longueur de la suite de réductions. □

1.4.3 Déterminisme de la réduction

La relation de réduction est déterministe :

Théorème 21. *Étant donnés une commande c et un état σ, si*

$$\langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ c_1 \mid \sigma_1\ \rangle \qquad et \qquad \langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ c_2 \mid \sigma_2\ \rangle$$

alors $c_1 = c_2$ et $\sigma_1 = \sigma_2$.

Démonstration. Par induction sur la dérivation de $\langle\ c \mid \sigma\ \rangle \longrightarrow \langle\ c_1 \mid \sigma_1\ \rangle$ (de façon similaire au théorème 29). □

Il est important que l'évaluation soit déterministe (ce qu'on a montré au théorème 29) mais ça n'est pas nécessairement le cas pour la réduction. Par exemple, si on changeait les règles définissant la réduction des additions en

$$\frac{\langle\ a_1 \mid \sigma\ \rangle \longrightarrow \langle\ a_1' \mid \sigma'\ \rangle}{\langle\ a_1\ \texttt{+}\ a_2 \mid \sigma\ \rangle \longrightarrow \langle\ a_1'\ \texttt{+}\ a_2 \mid \sigma'\ \rangle} \qquad \frac{\langle\ a_2 \mid \sigma\ \rangle \longrightarrow \langle\ a_2' \mid \sigma'\ \rangle}{\langle\ a_1\ \texttt{+}\ a_2 \mid \sigma\ \rangle \longrightarrow \langle\ a_1\ \texttt{+}\ a_2' \mid \sigma'\ \rangle}$$

la réduction ne serait plus déterministe car on aurait

$$\langle\ (2+2)+(3+3) \mid \sigma\ \rangle \longrightarrow \langle\ 4+(3+3) \mid \sigma\ \rangle$$
$$\langle\ (2+2)+(3+3) \mid \sigma\ \rangle \longrightarrow \langle\ (2+2)+6 \mid \sigma\ \rangle$$

Le non-déterminisme indique ici que la réduction est « sous-spécifiée », dans le sens où le choix de l'ordre de l'évaluation de l'addition (par la gauche ou par la droite) est laissé à l'implémentation. Pour notre langage, les règles ci-dessus ne poseraient pas de problème (le résultat de l'évaluation d'un expression arithmétique sera toujours le même qu'on commence par la gauche ou par la droite), mais ça serait plus ennuyeux si on s'autorisait des opérations avec effet de bord dans les expressions. C'est par exemple le cas dans le langage C, où l'ordre d'évaluation des fonctions (comme l'addition) n'est pas spécifié, et le programme

```
#include <stdio.h>

int a;

int f(int i) {
  a = i;
  return i;
}

int main(void) {
  int b = f(1) + f(2);
  printf("a = %d\n", a);
  return 0;
}
```

peut donc afficher `a = 1` ou `a = 2` suivant le compilateur utilisé...

2 Typage

Le langage IMP étudié à la section précédente est constitué de trois catégories syntaxiques distinctes. Ceci signifie que l'on peut déterminer de façon univoque si on a affaire à une expression arithmétique, booléenne ou un commande, simplement en regardant la syntaxe du programme. Pour les langages plus réalistes, ça n'est pas le cas. Par exemple, étant donné le nom d'une variable x, on ne peut généralement pas déterminer si elle va contenir un entier, un flottant ou une

chaîne de caractères. Dans ce cadre, ou souhaiterait s'assurer que les opérations qu'on utilise sont toujours bien définies ; par exemple, qu'on ne va jamais essayer de calculer le résultat de

```
2 + true
```

ce qui n'a pas de sens (sauf à considérer que `true` peut être implicitement vu comme un entier, par exemple `1`, ce que nous ne ferons pas ici). Bien sûr, un programmeur n'écrira presque jamais une telle expression. En revanche, il arrive parfois d'écrire `2 + x` dans le programme, d'oublier que `x` est censé contenir un entier, et d'écrire ailleurs une affectation de la forme `x := true`. Ainsi, on veut non seulement éviter les « fautes directes », mais aussi s'assurer qu'il n'y aura jamais de calcul non défini au cours de l'exécution du programme, ce qui est bien moins simple, car on souhaite aussi éviter d'avoir à exécuter intégralement le programme pour faire cette vérification.

Nous allons montrer comment avoir de telles garanties en utilisant un système de types, qui associe à chaque expression un *type* qui décrit sa nature (un entier, un booléen, etc.). Pour illustrer notre propos, nous introduisons en section 2.1 une version miniature du langage ML (pour *méta-langage*) introduit par Robin Milner (dont OCaml est l'une des variantes), définissons un système de types pour celui-ci en section 2.2. Nous formalisons les sémantiques opérationnelles (à grands et à petits pas) de ce langage en section 2.3, ce qui nous permet de montrer qu'un programme typable n'aboutit jamais à des erreurs à l'exécution en section 2.4. Ainsi, la discipline de typage permet de façon *statique* (c'est-à-dire par une analyse au moment de la compilation, sans avoir à exécuter le programme) d'apporter des garanties fortes sur toutes les exécutions possibles du programme. En section 2.5, nous ajoutons des opérateurs de points fixes au langage afin d'obtenir un langage réaliste d'un point de vue calculatoire. Enfin, en section 2.6, nous décrivons, en l'absence d'informations de types, un algorithme qui *infère* le type le plus général d'un programme donné.

2.1 Le langage mini-ML

On suppose fixé un ensemble $\mathcal{V}$ de variables. Les *expressions*, appelées encore *programmes* ou *termes*, dans le langage mini-ML sont de la forme suivante

$$\begin{array}{rcl} t & ::= & \underline{n} \mid \underline{b} \mid \texttt{add} \\ & & \mid x \mid \texttt{fun}\ x \to t \mid t\ t' \\ & & \mid \texttt{(}t\texttt{,}t'\texttt{)} \mid \texttt{fst} \mid \texttt{snd} \end{array}$$

Un programme peut donc être
- $\underline{n}$: un entier n,
- $\underline{b}$: un booléen b (soit $\top$, soit $\bot$)
- `add` : l'addition de deux entiers,
- x : une variable,
- `fun` $x \to t$: la fonction qui à x associe l'expression t,
- $t\ t'$: l'application de la fonction t à l'argument t',
- `(`t`,`t'`)` : la paire constituée de t et de t',
- `fst`, `snd` : les deux projections qui permettent d'extraire les composantes d'une paire.

On notera **Exp** l'ensemble des expressions du langage. Dans la suite, on s'autorisera par fois à noter t `+` t' au lieu de `add` $t\ t'$. Par souci de concision, nous n'avons mis que l'addition `add` comme opération primitive, mais bien sûr on pourrait sans difficulté ajouter les autres opérations arithmétiques (soustraction, multiplication, etc.) et booléennes (conjonction, négation, comparaisons, etc.) usuelles. On pourrait aussi ajouter le branchement conditionnel `if` t `then` t' `else` t'' (voir section 2.3.6).

Comparons rapidement le langage mini-ML avec IMP. Tout d'abord il est *fonctionnel*, ce qui signifie que l'on peut déclarer des fonctions, sans nécessairement les nommer (avec la construction `fun` $x \to t$) et les manipuler comme n'importe quelle autre expression du langage. D'autre part il est *pur* : il n'y a pas d'opération qui modifie un état externe (comme la manipulation de variables auxquelles on peut assigner une valeur). Enfin, les fonctions peuvent manipuler des valeurs de n'importe quel type : il n'y a pas de distinction syntaxique entre les différents types de valeurs (expressions arithmétiques / booléennes / commandes), et c'est précisément ce qui va nous amener à introduire un système de types afin de réguler tout cela.

2.2 Typage

2.2.1 Types

On considère des *types* qui sont engendrés par la grammaire suivante:

$$A \quad ::= \quad \texttt{int} \quad | \quad \texttt{bool} \quad | \quad A \Rightarrow A' \quad | \quad A \times A'$$

Un type est donc
- l'un des types de base (`int` pour les entiers, `bool` pour les booléens),
- un type $A \Rightarrow A'$ qui indique une fonction prenant un argument de type A et renvoyant un résultat de type A',
- un type $A \times A'$ qui indique une paire dont la première composante est de type A et la seconde de type A'.

2.2.2 Typage

Nous allons maintenant définir la relation de *typage* $\vdash t : A$ qui décrit quand un programme t admet A comme type. Par exemple, on s'attend à ce que l'on ait

$$\vdash \texttt{fun } x \rightarrow \texttt{(}x \texttt{ + 1,true)} : \texttt{int} \Rightarrow (\texttt{int} \times \texttt{bool})$$

En d'autres termes, le programme

$$\texttt{fun } x \rightarrow \texttt{(}x \texttt{ + 1,true)}$$

admet le type

$$\texttt{int} \Rightarrow (\texttt{int} \times \texttt{bool})$$

ce qui indique que c'est une fonction prenant un entier en argument et renvoyant une pair constituée d'un entier et d'un booléen.

Il s'avèrera nécessaire de propager le type des variables (libres) d'un programme : le programme `fun` $x \rightarrow x$ `+` y n'est pas bien typé (puisque y n'est pas définie), mais il le devient si on suppose que y est de type `int`, ce qui motive la généralisation suivante. Un *environnement de typage*, ou *contexte*, Γ est une liste

$$x_1 : A_1, \ldots, x_n : A_n$$

de paires $x_i : A_i$ constituées d'une variable x_i et d'un type A_i. Le *domaine* d'un tel environnement est l'ensemble des variables $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$. On notera $(x : A) \in \Gamma$ lorsqu'il existe un indice i (avec $1 \leqslant i \leqslant n$) tel que $x = x_i$ et $A = A_i$, et $x_j \neq x$ pour $i < j \leqslant n$ (c'est-à-dire que A est le type associé à l'occurrence la plus à droite de x dans Γ). On va considérer des *jugements de typage* de la forme

$$\Gamma \vdash t : A$$

c'est-à-dire des triplets constitués d'un environnement de typage Γ, d'un terme t et d'un type A qui doivent être lus comme

« en supposant que les variables x_i on le type A_i indiqué dans Γ, le terme t a le type A »

Les jugements de la forme $\vdash t : A$ considérés précédemment sont les cas particulier où Γ est la liste vide. Les jugements de typage valides seront décrits par des règles d'inférence de la forme

$$\frac{\Gamma_1 \vdash t_1 : A_1 \qquad \ldots \qquad \Gamma_n \vdash t_n : A_n}{\Gamma \vdash t : A}$$

qui signifient comme ci-dessus que si tous les $\Gamma_i \vdash t_i : A_i$ en hypothèse (au dessus de la barre) sont dérivables alors on peut aussi dériver $\Gamma \vdash t : A$ (c'est la conclusion, en dessous de la barre). Une telle règle peut aussi se lire du bas vers le haut : pour montrer que t a le type A dans le contexte Γ, il faut montrer que tous les jugements de typage $\Gamma_i \vdash t_i : A_i$ sont dérivables. Un jugement de typage est donc valide quand il est la conclusion d'un *arbre de dérivation* (ou *dérivation de typage*), formé des règles d'inférence.

Les règles d'inférence sont données à la figure 5. Leur signification est la suivante :

$$\frac{}{\Gamma \vdash \underline{n} : \texttt{int}}\ (\textsc{int}) \qquad \frac{}{\Gamma \vdash \underline{b} : \texttt{bool}}\ (\textsc{bool})$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash \texttt{add} : \texttt{int} \times \texttt{int} \Rightarrow \texttt{int}}\ (\textsc{add})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\ x\ \rightarrow\ t : A \Rightarrow B}\ (\textsc{fun}) \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\ (\textsc{app})$$

$$\frac{}{\Gamma \vdash \texttt{fst} : A \times B \Rightarrow A}\ (\textsc{fst})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t,u) : A \times B}\ (\textsc{pair}) \qquad \frac{}{\Gamma \vdash \texttt{snd} : A \times B \Rightarrow B}\ (\textsc{snd})$$

FIGURE 5 – Règles de typage.

— (INT), (BOOL), (ADD), (FST), (SND) : pour les constantes (entiers, booléens, `add`, `fst`, `snd`), la règle indique que la constante a le type attendu quel que soit le contexte Γ, sans hypothèse,
— (VAR) : si on a la supposition, dans le contexte Γ, que la variable x est de type A, on peut en déduire que x est de type A,
— (FUN) : montrer que $\texttt{fun}\ x\ \rightarrow\ t$ est une fonction de type $A \Rightarrow B$, revient à montrer que t a le type B en supposant que x est de type A,
— (APP) : si t est une fonction de type $A \Rightarrow B$ et u est une expression de type A alors $t\ u$ est de type B,
— (PAIR) : si t a le type A et u a le type B alors (t,u) a le type $A \times B$.

Exemple 22. Le type donné pour le programme ci-dessus peut être montré à l'aide de la dérivation de typage suivante :

$$\dfrac{\dfrac{\dfrac{}{x : \texttt{int} \vdash \texttt{add} : \texttt{int} \times \texttt{int} \Rightarrow \texttt{int}}\ (\textsc{add}) \quad \dfrac{\dfrac{}{x : \texttt{int} \vdash x : \texttt{int}}\ (\textsc{var}) \quad \dfrac{}{x : \texttt{int} \vdash 1 : \texttt{int}}\ (\textsc{int})}{x : \texttt{int} \vdash (x,1) : \texttt{int} \times \texttt{int}}\ (\textsc{add})}{x : \texttt{int} \vdash \texttt{add}\ (x,1) : \texttt{int}}\ (\textsc{app}) \quad \dfrac{}{x : \texttt{int} \vdash \texttt{true} : \texttt{bool}}\ (\textsc{bool})}{x : \texttt{int} \vdash (\texttt{add}\ (x,1),\texttt{true}) : (\texttt{int} \times \texttt{bool})}\ (\textsc{pair})}{\vdash \texttt{fun}\ x\ \rightarrow\ (\texttt{add}\ (x,1),\texttt{true}) : \texttt{int} \Rightarrow (\texttt{int} \times \texttt{bool})}\ (\textsc{fun})$$

Exemple 23. Le programme

$$(\texttt{fun}\ f\ \rightarrow\ f\ 0)\ (\texttt{fun}\ x\ \rightarrow\ x\ \texttt{+}\ 1)$$

admet le type `int`. En effet,

$$\dfrac{\dfrac{\dfrac{}{f : \texttt{int} \Rightarrow \texttt{int} \vdash f : \texttt{int} \Rightarrow \texttt{int}}\ (\textsc{var}) \quad \dfrac{}{f : \texttt{int} \Rightarrow \texttt{int} \vdash 0 : \texttt{int}}\ (\textsc{int})}{f : \texttt{int} \Rightarrow \texttt{int} \vdash f\ 0 : \texttt{int}}\ (\textsc{app})}{\vdash \texttt{fun}\ f\ \rightarrow\ f\ 0 : \texttt{int} \Rightarrow \texttt{int} \Rightarrow \texttt{int}}\ (\textsc{fun}) \quad \dfrac{\dfrac{\vdots}{x : \texttt{int} \vdash x\ \texttt{+}\ 1 : \texttt{int}}}{\vdash \texttt{fun}\ x\ \rightarrow\ x\ \texttt{+}\ 1 : \texttt{int} \Rightarrow \texttt{int}}\ (\textsc{fun})}{\vdash (\texttt{fun}\ f\ \rightarrow\ f\ 0)\ (\texttt{fun}\ x\ \rightarrow\ x\ \texttt{+}\ 1) : \texttt{int}}\ (\textsc{app})$$

2.2.3 Premières propriétés du système de type

Étant donné un système de types tel que celui décrit ci-dessus, il est naturel de s'intéresser à ses propriétés.

— *Unicité du typage* : le type d'un terme est-il unique ?
— *Canonicité du typage* : s'il n'y a pas unicité, a-t-on au moins un « meilleur type » pour un terme ?
— *Préservation par réduction* : le typage est-il préservé au cours de l'exécution ?

On peut aussi s'intéresser aux questions suivantes :

— *Typablité* : étant donné un programme t, on cherche à déterminer s'il est typable, c'est-à-dire s'il existe un type A tel que $\vdash t : A$.
— *Inférence de type* : étant donné un programme t, on cherche à déterminer un type A tel que $\vdash t : A$ est dérivable lorsque le terme est typable.
— *Vérification de type* : étant donné un programme t et un type A, on cherche à déterminer si le jugement $\vdash t : A$ est dérivable.

Remarque 24. Notons que si on sait inférer les types, on sait résoudre le problème de la typabilité. De plus, si on a unicité du typage alors la vérification se réduit à l'inférence : pour savoir si un terme t admet un type A, il suffit d'inférer le type B de t et de vérifier si $B = A$.

2.2.4 Programmes non-typables

Dans notre système de type, il existe des programmes qui ne sont pas typables. Par exemple,

```
add (true,false)
```

n'admet pas de type. En effet, on peut observer qu'à chaque construction du langage est associée une unique règle qui permet de typer les programmes de cette forme (par exemple, une fonction sera nécessairement typée à l'aide de la règle (FUN)). Ici, une dérivation de type doit commencer par

$$\dfrac{\dfrac{\pi_1}{\vdash \texttt{add} : A \Rightarrow B} \qquad \dfrac{\pi_2}{\vdash \texttt{(true,false)} : A}}{\vdash \texttt{add (true,false)} : B}\ (\textsc{app})$$

La règle pour la dérivation π_1 est nécessairement (ADD), ce qui implique $A = \texttt{int} \times \texttt{int}$ et $B = \texttt{int}$. La dérivation π_2 est à son tour de la forme

$$\dfrac{\dfrac{\pi_3}{\vdash \texttt{true} : \texttt{int}} \qquad \dfrac{\pi_4}{\vdash \texttt{false} : \texttt{int}}}{\texttt{(true,false)} : \texttt{int} \times \texttt{int}}\ (\textsc{pair})$$

et il n'y a pas de règle qui permette de construire la dérivation π_3 (ou π_4). En effet, comme on cherche à typer `true`, la seule possibilité est d'utiliser la règle (BOOL), mais celle-ci ne permet pas de déduire que le programme a le type `int` (elle permet simplement de déduire que le programme a le type `bool`).

Voici d'autres exemples de programmes non-typables:

```
add 1
fun x → y
fun f → (f 1,f true)
fun f → f f
```

Exercice 25. Montrez que c'est le cas.

2.2.5 Unicité du typage

Il existe des programmes dont le type n'est pas unique. Par exemple, la fonction identité

```
fun x → x
```

admet les types `fun int → int` et `fun bool → bool`, ainsi que plus généralement le type

$$A \Rightarrow A$$

pour tout type A. En effet, on a, pour tout type A,

$$\dfrac{\dfrac{}{x : A \vdash x : A}\ (\textsc{var})}{\vdash \texttt{fun}\ x\ \rightarrow\ x : A \Rightarrow A}\ (\textsc{fun})$$

De même, le programme `fst` admet les types $\texttt{int} \times \texttt{int} \Rightarrow \texttt{int}$ ou $\texttt{bool} \times \texttt{bool} \Rightarrow \texttt{bool}$, ou de façon plus générale

$$A \times B \Rightarrow A$$

pour tout types A et B.

Essayons de modifier le langage afin d'avoir un type unique, que l'on peut facilement inférer. Si l'on essaye de comprendre la raison de la non-unicité dans le premier exemple ci-dessus, on se rend compte que le coupable est la règle

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\ x\ \to\ t : A \Rightarrow B}\ (\textsc{fun})$$

En effet, si l'on veut déterminer le type de $\texttt{fun}\ x\ \to\ t$ dans l'environnement Γ, il faut déterminer le type t dans l'environnement $\Gamma, x : A$ pour un certain type A pour lequel nous n'avons aucune indication. Une façon de contourner le problème est de demander à l'utilisateur d'indiquer ce type en changeant la syntaxe des fonctions de

$$\texttt{fun}\ x\ \to\ t$$

en

$$\texttt{fun}\ (x : A)\ \to\ t$$

qui indique qu'on a un argument x de type A, et la règle associée en

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\ (x : A)\ \to\ t : A \Rightarrow B}\ (\textsc{fun})$$

Pour les mêmes raisons, il est aussi nécessaire de changer la syntaxe de `fst` et `snd` en

$$\texttt{fst}_{A,B} \qquad \text{et} \qquad \texttt{snd}_{A,B}$$

et les règles associées en

$$\frac{}{\Gamma \vdash \texttt{fst}_{A,B} : A \times B \Rightarrow A}\ (\textsc{fst}) \qquad\qquad \frac{}{\Gamma \vdash \texttt{snd}_{A,B} : A \times B \Rightarrow B}\ (\textsc{snd})$$

On a appelle le langage résultat *mini-ML avec indications de type* (cela correspond à la variante *à la Church* du λ-calcul).

Remarque 26. Au lieu d'imposer une annotation de type à `fst` et `snd`, on pourrait aussi imposer qu'il soient toujours appliqués à un terme, c'est-à-dire changer la syntaxe en

$$t \quad ::= \quad \ldots \quad | \quad \texttt{fst}\, t \quad | \quad \texttt{snd}\, t$$

En effet, le type de l'argument permet alors toujours de déterminer le type retourné. Ainsi, l'expression `fst` n'est plus valide dans le langage, mais on peut toujours obtenir la première projection comme $\texttt{fun}\ x\ \to\ \texttt{fst}\, x$, et de même pour la seconde projection. Les règles de typage associées sont

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \texttt{fst}\, t : A}\ (\textsc{fst}) \qquad\qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \texttt{snd}\, t : B}\ (\textsc{snd})$$

Théorème 27. *Dans le langage mini-ML avec indications de type, un programme t admet au plus un type.*

Démonstration. On montre de façon plus générale que pour tout contexte Γ et programme t il y a au plus un type A tel que $\Gamma \vdash t : A$ est dérivable. On raisonne par récurrence sur t.

— Si t est de la forme $\underline{n}$ alors un jugement de typage est nécessairement de la forme

$$\frac{}{\Gamma \vdash \underline{n} : \texttt{int}}\ (\textsc{int})$$

et son type est `int`. Le cas $t = \underline{b}$ est similaire.

— Si t est de la forme x alors un jugement de typage est nécessairement de la forme

$$\frac{}{\Gamma \vdash x : A} \text{ (VAR)}$$

avec $(x : A) \in \Gamma$ et A est donc déterminé de façon unique par Γ lorsqu'il l'est (on utilise ici le fait que A est le type associé à l'occurrence la plus à droite de x dans Γ).

— Si t est de la forme $\texttt{fun}\ (x : A)\ \to\ u$ alors un jugement de typage est nécessairement de la forme

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \texttt{fun}\ (x : A)\ \to\ u : A \Rightarrow B} \text{ (FUN)}$$

et t est typable si et seulement si u admet un type B dans l'environnement $\Gamma, x : A$, auquel cas son type est $A \Rightarrow B$. Par hypothèse de récurrence le type B de u est unique et donc le type de t l'est aussi.

— Si t est de la forme $u\ v$ alors un jugement de typage est nécessairement de la forme

$$\frac{\Gamma \vdash u : A \Rightarrow B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B} \text{ (APP)}$$

et t est typable si le type de u (unique par hypothèse de récurrence) est de la forme $A \Rightarrow B$ et le type de v (unique par hypothèse de récurrence) est A, auquel cas son typé est nécessairement B.

— Les autres cas sont similaires et laissés au lecteur. □

Notons que la preuve ci-dessus est *constructive* : non seulement elle montre l'unicité du programme t, mais elle le construit explicitement par récurrence sur t. Ceci montre que l'on sait inférer les types dans mini-ML avec annotations de type (et donc vérifier les types, par la remarque théorème 24).

Nous verrons en section 2.6 une solution plus satisfaisante pour palier à la non-unicité du typage : au lieu de demander à l'utilisateur de spécifier certains types, nous allons généraliser la notion de type, en introduisant des variables. Dans ce système, un programme n'aura pas un type unique, mais on pourra montrer qu'il existe un type *principal*, qui est plus général que tous les autres.

2.3 Sémantique

Comme dans le cas du langage IMP, on peut définir une sémantique opérationnelle à grands pas et à petits pas pour mini-ML. De par la nature assez différente du langage, il nous est nécessaire de définir à nouveau ces notions.

2.3.1 Évaluation

Commençons par définir l'*évaluation*, c'est-à-dire la sémantique à grands pas. Les *valeurs*, qui sont les termes que l'on considère comme étant un résultat, sont ici ceux engendrés par la grammaire

$$v \quad ::= \quad \underline{n} \ \mid\ \underline{b} \ \mid\ \texttt{add} \ \mid\ \texttt{fun}\ x \to t \ \mid\ (v,v') \ \mid\ \texttt{fst} \ \mid\ \texttt{snd}$$

où t est un terme quelconque.

On considère donc comme une valeur :

— une valeur de base (entier, booléen),

— une opération de base (`add`, `fst`, `snd`),

— une fonction,

— une paire de valeurs.

Notons qu'on ne demande pas à ce que le corps d'une fonction soit une valeur. En effet, on ne va pas chercher à évaluer le corps d'une fonction tant qu'elle n'est pas appliquée à un argument.

Remarque 28. En OCaml, l'évaluation se déroule aussi de cette façon. En effet, on peut observer l'évaluation en affichant des chaînes de caractères. Par exemple,

$$\frac{}{\underline{n} \longrightarrow\!\!\!\!\rightarrow \underline{n}}\ (\textsc{int}) \qquad \frac{}{\underline{b} \longrightarrow\!\!\!\!\rightarrow \underline{b}}\ (\textsc{bool})$$

$$\frac{}{\texttt{add} \longrightarrow\!\!\!\!\rightarrow \texttt{add}}\ (\textsc{add}) \qquad \frac{t \longrightarrow\!\!\!\!\rightarrow \texttt{add} \qquad u \longrightarrow\!\!\!\!\rightarrow (\underline{m},\underline{n})}{t\ u \longrightarrow\!\!\!\!\rightarrow \underline{m+n}}\ (\textsc{sum})$$

$$\frac{}{(\texttt{fun}\ x\ \rightarrow\ t) \longrightarrow\!\!\!\!\rightarrow (\texttt{fun}\ x\ \rightarrow\ t)}\ (\textsc{fun}) \qquad \frac{t \longrightarrow\!\!\!\!\rightarrow (\texttt{fun}\ x\ \rightarrow\ t') \qquad u \longrightarrow\!\!\!\!\rightarrow u' \qquad t'[x \mapsto u'] \longrightarrow\!\!\!\!\rightarrow v}{t\ u \longrightarrow\!\!\!\!\rightarrow v}\ (\textsc{app})$$

$$\frac{t \longrightarrow\!\!\!\!\rightarrow t' \qquad u \longrightarrow\!\!\!\!\rightarrow u'}{(t,u) \longrightarrow\!\!\!\!\rightarrow (t',u')}\ (\textsc{pair})$$

$$\frac{}{\texttt{fst} \longrightarrow\!\!\!\!\rightarrow \texttt{fst}}\ (\textsc{fst}) \qquad \frac{t \longrightarrow\!\!\!\!\rightarrow \texttt{fst} \qquad u \longrightarrow\!\!\!\!\rightarrow (v_1,v_2)}{t\ u \longrightarrow\!\!\!\!\rightarrow v_1}\ (\textsc{fstp})$$

$$\frac{}{\texttt{snd} \longrightarrow\!\!\!\!\rightarrow \texttt{snd}}\ (\textsc{snd}) \qquad \frac{t \longrightarrow\!\!\!\!\rightarrow \texttt{snd} \qquad u \longrightarrow\!\!\!\!\rightarrow (v_1,v_2)}{t\ u \longrightarrow\!\!\!\!\rightarrow v_2}\ (\textsc{sndp})$$

FIGURE 6 – Évaluation en mini-ML.

```
let x = print_string "A"; 3 + 2;;
```

affiche `A`, ce qui montre que OCaml évalue l'expression définissant `x` (il va assigner la valeur `5` à `x`). Mais

```
let f = fun x -> print_string "A"; 3 + 2;;
```

ne provoque pas d'affichage car ce qui montre que le corps de la fonction `f` n'est pas évalué. Il ne le sera que si on appelle la fonction en lui fournissant un argument : par exemple `f 1` va afficher `A` et renvoyer `5`.

Nous pouvons maintenant définir la relation d'*évaluation* $t \longrightarrow\!\!\!\!\rightarrow v$ qui indique qu'un terme t s'évalue en une valeur v : le membre de droite sera toujours une valeur. Notons que, contrairement à IMP, cette relation ne dépend pas d'un état courant. Comme à l'accoutumée, on définit cette relation au moyen de règles d'inférence, données à la figure 10.

Commentons ces règles.

— (INT) et (BOOL) : les deux premières règles indiquent respectivement que les entiers et les booléens s'évaluent en eux-mêmes.

— (ADD) indique que la fonction `add` (sans argument) s'évalue en elle même.

— (SUM) indique que la fonction `add` appliquée à une paire d'arguments s'évalue en la somme de l'évaluation des arguments. Par exemple, on a

$$\texttt{3 + 2} \longrightarrow\!\!\!\!\rightarrow \texttt{5}$$

où on rappelle que `3 + 2` est une notation pour `add (3,2)`.

— (FUN) indique qu'une fonction (non appliquée à des arguments) s'évalue en elle-même.

— (APP) et (LET) sont plus subtiles et détaillées ci-dessous.

— (PAIR) indique que l'évaluation d'une paire est la paire de l'évaluation de ses composantes.

— (FST) et (SND) indiquent que `fst` et `snd`, qui sont déjà des valeurs, s'évaluent en eux-mêmes.

— (FSTP) et (SNDP) indique que `fst` et `snd` appliquées à une paire renvoient respectivement la première et la seconde composante.

Détaillons la règle (APP). Pour évaluer une application $t\ u$, on commence par évaluer t : on s'attend à ce que celui-ci s'évalue en une fonction $\texttt{fun}\ x\ \rightarrow\ t'$ (ou alors une opération de base `add`, `fst` ou `snd`, ces étant traités par les règles spécifiques (SUM), (FSTP) et (SNDP) pour ces opérations). On évalue aussi l'argument u pour obtenir une valeur u'. Et enfin, on évalue l'expression $t'[x \mapsto u']$,

qui est une notation pour l'expression t' où la variable x a été remplacée par u'. Par exemple, considérons le programme t

$$(\texttt{fun}\ x\ \texttt{->}\ (x\ \texttt{+}\ x))\ (\texttt{3}\ \texttt{+}\ \texttt{2})$$

Pour l'évaluer, on va évaluer la fonction

$$\texttt{fun}\ x\ \texttt{->}\ (x\ \texttt{+}\ x)$$

qui s'évalue en elle-même par la règle (FUN), ainsi que l'argument

$$\texttt{3}\ \texttt{+}\ \texttt{2}$$

qui s'évalue en `5`. On en déduit que le résultat de l'évaluation du programme t, va être obtenu en évaluant

$$(x\ \texttt{+}\ x)[x \mapsto \texttt{5}]$$

c'est-à-dire `5 + 5`, dont le résultat est `10`. Ceci peut être résumé par la dérivation suivante :

$$\dfrac{\dfrac{}{(\texttt{fun}\ x\ \texttt{->}\ (x\ \texttt{+}\ x)) \longrightarrow\!\!\!\!\rightarrow (\texttt{fun}\ x\ \texttt{->}\ (x\ \texttt{+}\ x))}\ (\textsc{fun}) \qquad \dfrac{}{\texttt{3}\ \texttt{+}\ \texttt{2} \longrightarrow\!\!\!\!\rightarrow \texttt{5}}\ (\textsc{sum}) \qquad \dfrac{}{\texttt{5}\ \texttt{+}\ \texttt{5} \longrightarrow\!\!\!\!\rightarrow \texttt{10}}\ (\textsc{sum})}{(\texttt{fun}\ x\ \texttt{->}\ (x\ \texttt{+}\ x))\ (\texttt{3}\ \texttt{+}\ \texttt{2}) \longrightarrow\!\!\!\!\rightarrow \texttt{10}}\ (\textsc{app})$$

Parfois, ce qu'on applique à un argument n'est pas directement une fonction mais un terme qui s'évalue en une fonction. En effet, les fonctions peuvent elles-mêmes renvoyer des fonctions ! Afin de donner un exemple, considérons tout d'abord le programme

$$\texttt{fun}\ f\ \texttt{->}\ (\texttt{fun}\ x\ \texttt{->}\ f\ (f\ x))$$

Celui-ci prend en argument une fonction f et renvoie la fonction qui à x associe f appliqué deux fois à x. Par exemple, si on l'applique à la fonction qui incrémente de 1 son argument, on obtient la fonction qui incrémente de 2 son argument :

$$(\texttt{fun}\ f\ \texttt{->}\ (\texttt{fun}\ x\ \texttt{->}\ f\ (f\ x)))\ (\texttt{fun}\ n\ \texttt{->}\ n\ \texttt{+}\ \texttt{1})$$

se réduit en

$$\texttt{fun}\ x\ \texttt{->}\ (\texttt{fun}\ n\ \texttt{->}\ n\ \texttt{+}\ \texttt{1})\ ((\texttt{fun}\ n\ \texttt{->}\ n\ \texttt{+}\ \texttt{1})\ x)$$

On peut alors montrer que le programme

$$((\texttt{fun}\ f\ \texttt{->}\ (\texttt{fun}\ x\ \texttt{->}\ f\ (f\ x)))\ (\texttt{fun}\ n\ \texttt{->}\ n\ \texttt{+}\ \texttt{1}))\ (\texttt{3}\ \texttt{+}\ \texttt{2})$$

s'évalue en `7` par la règle (APP). En effet le terme appliqué s'évalue en la fonction qui incrémente par 2 décrite ci-dessus, l'argument s'évalue en `5` et le résultat de l'évaluation sera celui de l'évaluation de

$$(\texttt{fun}\ n\ \texttt{->}\ n\ \texttt{+}\ \texttt{1})\ ((\texttt{fun}\ n\ \texttt{->}\ n\ \texttt{+}\ \texttt{1})\ \texttt{5})$$

qui est `7`. La règle (LET) peut être expliquée de façon similaire.

On peut montrer que l'évaluation est déterministe :

Théorème 29. *Si $t \longrightarrow\!\!\!\!\rightarrow v$ et $t \longrightarrow\!\!\!\!\rightarrow v'$ alors $v = v'$.*

Démonstration. Par induction sur la dérivation de $t \longrightarrow\!\!\!\!\rightarrow v$. □

On pourrait donc décrire l'évaluation par une fonction partielle qui à un terme associe le résultat de son évaluation. Cette fonction n'est pas totale. Par exemple, les programmes

```
add false
true 5
(fun f -> f f) (fun f -> f f)
```

ne s'évaluent pas en une valeur (montrez-le !).

2.3.2 Substitution et renommage

Dans la section précédente, nous avons utilisé la notation

$$t[x \mapsto u]$$

qui représente le terme t dans lequel toutes les occurrences libres de la variable x ont été remplacée par u, c'est-à-dire la *substitution* de x par u dans t. Par exemple, on aura

$$(\texttt{fun}\ y\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{+}\ y)[x \mapsto u] = \texttt{fun}\ y\ \texttt{->}\ u\ \texttt{+}\ u\ \texttt{+}\ y$$

Celle-ci est en effet utilisée dans la règle (APP) (et la règle (LET)) pour définir la réduction de l'application d'une fonction à un argument : si u est une valeur, on aura

$$(\texttt{fun}\ x\ \texttt{->}\ t)\ u \longrightarrow\!\!\!\!\!\rightarrow t[x \mapsto u]$$

Par exemple,

$$(\texttt{fun}\ x\ \texttt{->}\ (\texttt{fun}\ y\ \texttt{->}\ x\ \texttt{+}\ x))\ u \longrightarrow\!\!\!\!\!\rightarrow (\texttt{fun}\ y\ \texttt{->}\ x\ \texttt{+}\ x)[x \mapsto u]$$

On s'attend à pouvoir définir la substitution par induction sur le terme t sur lequel on l'effectue la substitution, mais cette définition présente quelques difficultés. Par exemple, on s'attend à ce que l'on ait

$$(\texttt{fun}\ y\ \texttt{->}\ x\ \texttt{+}\ x)[x \mapsto u] = \texttt{fun}\ y\ \texttt{->}\ u\ \texttt{+}\ u$$

et c'est « presque » juste, excepté quand u fait lui-même référence à y (ou quand x et y sont la même variable). En effet,

— pour $t = z$ (où z est une variable arbitraire, distincte de y), le résultat de la substitution

$$(\texttt{fun}\ y\ \texttt{->}\ x\ \texttt{+}\ x)[x \mapsto z] = \texttt{fun}\ y\ \texttt{->}\ z\ \texttt{+}\ z$$

est la fonction qui ignore son argument et renvoie le double de z,

— en revanche pour $t = y$, on a

$$(\texttt{fun}\ y\ \texttt{->}\ x\ \texttt{+}\ x)[x \mapsto y] = \texttt{fun}\ y\ \texttt{->}\ y\ \texttt{+}\ y$$

est la fonction qui renvoie le double de son argument, au lieu de la fonction qui ignore son argument renvoie le double de y.

Afin de traiter ce dernier cas, observons que dans une fonction, le nom de la variable utilisée pour l'argument importe peu. Dans le second cas, on effectuera plutôt la substitution en renommant d'abord la variable y de l'argument (en z par exemple) :

$$(\texttt{fun}\ y\ \texttt{->}\ x\ \texttt{+}\ x)[x \mapsto y] = (\texttt{fun}\ z\ \texttt{->}\ x\ \texttt{+}\ x)[x \mapsto y] = \texttt{fun}\ z\ \texttt{->}\ y\ \texttt{+}\ y$$

On dira que deux fonctions qui ne diffère que par un *renommage* des variables utilisées en argument sont *α-convertibles*. C'est par exemple le cas des fonctions :

$$\texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x \qquad\qquad \texttt{fun}\ y\ \texttt{->}\ y\ \texttt{+}\ y$$

On considérera deux telles fonctions comme équivalentes. Dans une expression de la forme $\texttt{fun}\ x\ \texttt{->}\ t$ on dira que la variable x est *liée* dans t, pour indiquer qu'on peut la renommer. Une variable qui n'est pas liée sera dite *libre*. Après ces explications, introduisons ces notions de façons formelles.

Définition 30 (Variables libres)**.** Étant donné un terme t, on définit l'ensemble $\mathrm{VL}(t)$ de ses *variables libres* par

$$\begin{aligned} \mathrm{VL}(\underline{n}) = \mathrm{VL}(\underline{b}) = \mathrm{VL}(\texttt{add}) = \mathrm{VL}(\texttt{fst}) = \mathrm{VL}(\texttt{snd}) &= \emptyset \\ \mathrm{VL}(\texttt{fun}\ x\ \texttt{->}\ t) &= \mathrm{VL}(t) \setminus \{x\} \\ \mathrm{VL}(t\ u) = \mathrm{VL}(\texttt{(}t\texttt{,}u\texttt{)}) &= \mathrm{VL}(t) \cup \mathrm{VL}(u) \end{aligned}$$

Définition 31 (Substitution)**.** Étant donnés une variable x et deux termes t et u, on définit la *substitution*

$$t[x \mapsto u]$$

de x par u dans t par induction sur t par

$$\begin{array}{rll}
t[x \mapsto u] = t & & \text{pour } t \in \{\underline{n}, \underline{b}, \texttt{add}, \texttt{fst}, \texttt{snd}\} \\
x[x \mapsto u] = u & & \\
y[x \mapsto u] = y & & \text{pour } x \neq y \\
(\texttt{fun } y \to t)[x \mapsto u] = \texttt{fun } y \to (t[x \mapsto u]) & & \text{si } x \neq y \text{ et } y \notin \mathrm{VL}(u) \\
(t\ t')[x \mapsto u] = (t[x \mapsto u])\ (t'[x \mapsto u]) & & \\
(t,t')[x \mapsto u] = (t[x \mapsto u], t'[x \mapsto u]) & &
\end{array}$$

Notons que l'opération de substitution n'est que partiellement définie. Par exemple, la valeur de

$$(\texttt{fun } x \to x)[x \mapsto x]$$

n'est pas définie (bien que dans ce cas on s'attende à avoir $\texttt{fun } x \to x$ comme résultat). On dit que deux termes t et u sont α-convertibles lorsqu'ils ne diffèrent que par renommage de variables liées. Formellement, cette relation est définie de la façon suivante :

Définition 32 (α-conversion)**.** On définit la relation d'*α-convertibilité* $\overset{\alpha}{=}$ sur les termes comme la plus petite congruence telle que

$$\texttt{fun } x \to t \quad \overset{\alpha}{=} \quad \texttt{fun } y \to (t[x \mapsto y])$$

lorsque le membre de droite est défini.

Plus explicitement, supposer que la relation $\overset{\alpha}{=}$ est une *congruence*, revient à supposer qu'elle est close par « passage au contexte », c'est-à-dire qu'elle satisfait les règles suivantes :

$$\frac{t \overset{\alpha}{=} t'}{\texttt{fun } x \to t \overset{\alpha}{=} \texttt{fun } x \to t'} \qquad \frac{t \overset{\alpha}{=} t' \qquad u \overset{\alpha}{=} u'}{t\ u \overset{\alpha}{=} t'\ u'} \qquad \frac{t \overset{\alpha}{=} t' \qquad u \overset{\alpha}{=} u'}{(t,u) \overset{\alpha}{=} (t',u')}$$

À partir de maintenant, nous allons considérer les termes du langages modulo α-conversion, c'est-à-dire qu'on identifie deux programmes qui ne diffèrent que par renommage de variables liées. On peut alors montrer que l'opération de substitution est toujours définie sur ces termes :

Proposition 33. *Étant donnés deux termes t et u et une variable x, il existe toujours un terme t' avec $t \overset{\alpha}{=} t'$ tel que $t'[x \mapsto u]$ est bien définie.*

Démonstration. Par induction sur t (voir le cours de λ-calcul). □

2.3.3 Réduction

La sémantique à grands pas, l'évaluation n'est pas très modulaire : elle calcule directement la valeur associée à un terme, on ne peut donc pas raisonner progressivement sur la réduction. De plus, elle ne permet pas de raisonner sur les termes qui ont des évaluations infinies, on ne peut pas distinguer entre en programme qui boucle (ce qui est acceptable) et un programme qui plante (ce qui n'est pas acceptable).

On définit la *réduction* comme la relation $\longrightarrow$ entre termes définie par les règles de la figure 7. Notez que dans ces règles, les termes v, v_1 et v_2 sont toujours supposées être des valeurs.

Face à une application $t\ u$, on réduit d'abord l'argument u par la règle (APPR) jusqu'à obtenir une valeur (on ne peut pas réduire t par (APPL) car celle-ci suppose que son argument est une valeur), puis on réduit t en une valeur par (APPL). Ensuite on poursuit la réduction suivant la forme de la valeur obtenue pour t :

— (APP) : si c'est une fonction on procède à la substitution de l'argument par sa valeur,
— (ADD) : si c'est `add` on calcule le résultat de l'addition,

Pour v, v_1 et v_2 des valeurs :

$$\frac{t \longrightarrow t'}{t\ v \longrightarrow t'\ v}\ (\textsc{appl}) \qquad \frac{u \longrightarrow u'}{t\ u \longrightarrow t\ u'}\ (\textsc{appr})$$

$$\frac{}{(\texttt{fun}\ x\ \to\ t)\ v \longrightarrow t[x \mapsto v]}\ (\textsc{app}) \qquad \frac{}{\texttt{add}\ (\underline{m}\texttt{,}\underline{n}) \longrightarrow \underline{m+n}}\ (\textsc{add})$$

$$\frac{t \longrightarrow t'}{(t\texttt{,}v) \longrightarrow (t'\texttt{,}v)}\ (\textsc{pairl}) \qquad \frac{u \longrightarrow u'}{(t\texttt{,}u) \longrightarrow (t\texttt{,}u')}\ (\textsc{pairr})$$

$$\frac{}{\texttt{fst}\ (v_1\texttt{,}v_2) \longrightarrow v_1}\ (\textsc{fst}) \qquad \frac{}{\texttt{snd}\ (v_1\texttt{,}v_2) \longrightarrow v_2}\ (\textsc{snd})$$

FIGURE 7 – Réduction en mini-ML.

— (FST) / (SND) : si c'est `fst` ou `snd`, on effectue la projection.

Enfin, pour les paires $(t\texttt{,}u)$, on évalue u par (PAIRR) puis t par (PAIRL).

On peut observer que c'est la stratégie implémentée en OCaml. Par exemple,

```
let p = print_string
let _ = (p "C"; (fun x y -> p "D"; x + y)) (p "B"; 2) (p "A"; 3)
```

va afficher `ABCD`.

Remarque 34 (Stratégies de réduction). La *stratégie de réduction* que nous avons présentée, c'est-à-dire la façon dont nous évaluons les programmes n'est pas la seule raisonnable possible. Présentons brièvement les différentes options qui s'offrent à nous.

— La stratégie que nous avons présentée est dite *en appel par valeur* : on commence par évaluer les arguments avant de les passer aux fonctions. Ainsi, on n'aura pas à plusieurs fois calculer sa valeur même si la fonction l'utilise plusieurs fois. Une évaluation sera par exemple

$$(\texttt{fun}\ x\ \to\ x+x)\ (3+2) \longrightarrow (\texttt{fun}\ x\ \to\ x+x)\ 5 \longrightarrow 5+5 \longrightarrow 10$$

— La réduction *en appel par nom*, applique les fonction aux argument sans les évaluer d'abord : il ne seront évalués qu'au moment où ils sont utilisés. Si on reprend l'exemple précédent, on aura

$$(\texttt{fun}\ x\ \to\ x+x)\ (3+2) \longrightarrow (3+2)+(3+2) \longrightarrow 5+(3+2) \longrightarrow 5+5 \longrightarrow 10$$

on notera que l'on a fait cette fois deux fois le travail d'évaluer $3+2$. Cette stratégie est donc généralement moins efficace que celle en appel par nom, excepté dans une situation (importante !) : si on a un argument dont l'évaluation ne termine pas mais qu'on n'utilise pas, cette stratégie ne bouclera pas, contrairement à l'appel par valeur.

— Une stratégie est dite *faible* si elle ne réduit pas sous les abstractions. C'est le cas de la stratégie que nous avons présentée. Dans une stratégie qui n'est pas faible, on pourrait avoir des réductions de la forme

$$(\texttt{fun}\ x\ \to\ 3+2) \longrightarrow (\texttt{fun}\ x\ \to\ 5)$$

La plupart des stratégies sont faibles, car on s'attend à ce que ça soit la présence d'argument qui lance le calcul du corps de la fonction.

— Une stratégie est dite *de tête* lorsque l'on ne réduit pas les variables appliquées à des termes, i.e. un terme de la forme $x\ t\ u$ ne sera pas réduit. Ça n'est pas le cas dans notre stratégie.

2.3.4 Formes normales

Une *forme normale* est un terme t qui ne se réduit pas, c'est-à-dire qu'il n'existe pas de terme t' tel que $t \longrightarrow t'$. Par définition des valeurs, on fait l'observation suivante :

Lemme 35. *Toute valeur est une forme normale.*

En revanche, la réciproque n'est pas vraie. Par exemple, le terme

$$\texttt{add true}$$

est une forme normale mais n'est pas une valeur. Nous verrons que le typage empêche précisément ce genre de situations.

Théorème 36. *La réduction est déterministe: si $t \longrightarrow u$ et $t \longrightarrow u'$ alors $u = u'$.*

Démonstration. La preuve se fait par induction sur le terme t, en utilisant le théorème 35 pour assurer que deux règles ne s'appliquent pas simultanément. Par exemple, dans le cas où le terme est de la forme (t,u), les deux règles qui peuvent a priori s'appliquer sont (PAIRL) et (PAIRR). Mais elles ne peuvent pas s'appliquer simultanément : en effet, si (PAIRL) s'applique alors u est une forme normale et (PAIRR) ne peut donc s'appliquer. □

2.3.5 Équivalence des sémantiques

On cherche maintenant à montrer que les deux sémantiques (à grands et à petits pas) coïncident. On note $t \xrightarrow{*} u$ pour indiquer qu'il existe une suite de réductions de t à u.

Proposition 37. *Si $t \longrightarrow\!\!\!\!\rightarrow v$ alors $t \xrightarrow{*} v$.*

Démonstration. Par induction sur la dérivation de $t \longrightarrow\!\!\!\!\rightarrow v$. Par exemple, supposons que la dernière règle utilisée est

$$\frac{t \longrightarrow\!\!\!\!\rightarrow (\texttt{fun}\ x\ \texttt{->}\ t') \qquad u \longrightarrow\!\!\!\!\rightarrow u' \qquad t'[x \mapsto u'] \longrightarrow\!\!\!\!\rightarrow v}{t\ u \longrightarrow\!\!\!\!\rightarrow v}\ (\textsc{app})$$

Par hypothèse de récurrence, on a

$$t \xrightarrow{*} (\texttt{fun}\ x\ \texttt{->}\ t') \qquad\qquad u \xrightarrow{*} u' \qquad\qquad t'[x \mapsto u'] \xrightarrow{*} v$$

et on en déduit la suite de réductions

$$t\ u \xrightarrow{*} t\ u' \xrightarrow{*} (\texttt{fun}\ x\ \texttt{->}\ t')\ u' \longrightarrow t'[x \mapsto u'] \xrightarrow{*} v$$

Pour les deux premières, il faut d'abord montrer les lemmes auxiliaires suivants :
— si $u \xrightarrow{*} u'$ alors $t\ u \xrightarrow{*} t\ u'$,
— si $t \xrightarrow{*} t'$ alors $t\ v \xrightarrow{*} t'\ v$ (pour v une valeur),
qui se montrent par récurrence sur la longueur de la dérivation en hypothèse. Les autres cas se traitent de façon similaire. □

Réciproquement, on commence par montrer les deux lemmes suivants.

Lemme 38. *Pour toute valeur v, on a $v \longrightarrow\!\!\!\!\rightarrow v$.*

Démonstration. Par induction sur v. □

Lemme 39. *Si $t \longrightarrow t'$ et $t' \longrightarrow\!\!\!\!\rightarrow v$ alors $t \longrightarrow\!\!\!\!\rightarrow v$.*

Démonstration. Par induction sur la dérivation de $t \longrightarrow t'$. □

Proposition 40. *Si $t \xrightarrow{*} v$ pour une valeur v alors $t \longrightarrow\!\!\!\!\rightarrow v$.*

Démonstration. Par récurrence sur la longueur de la réduction $t \xrightarrow{*} v$ en utilisant théorème 38 pour le cas de base et théorème 39 pour le cas de récurrence. □

On en conclut, que les deux sémantiques coïncident.

Théorème 41. *Pour v une valeur, on a $t \xrightarrow{*} v$ si et seulement si $t \longrightarrow\!\!\!\!\rightarrow v$.*

2.3.6 Extensions du langage

On peut facilement ajouter d'autres constructions simples au langage.

Déclarations locales. On peut ajouter des *déclarations locales* (comme le `let` en OCaml) par la syntaxe suivante :

$$\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u \qquad = \qquad (\texttt{fun}\ x\ \to\ u)\, t$$

Si on tient à les ajouter comme de véritables termes (et non comme un sucre syntaxique), on peut ajouter à la syntaxe de termes

$$t \quad ::= \quad \ldots \quad | \quad \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ t'$$

Notons que dans une expression de la forme $\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u$ la variable x est liée dans u (mais pas dans t), on s'autorisera donc à la renommer de façon similaire aux abstractions. On ajoute la règle de typage

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash u : B}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u : B}\ (\textsc{let})$$

qui indique que montrer que $\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u$ a le type B revient à montrer que t est typable avec un certain type A et u a le type B en supposant que x a le type A. Les deux règles de réduction sont

$$\frac{t \longrightarrow t'}{\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u \longrightarrow \texttt{let}\ x\ \texttt{=}\ t'\ \texttt{in}\ u}\ (\textsc{letl}) \qquad \frac{}{\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ t \longrightarrow t[x \mapsto v]}\ (\textsc{let})$$

et la règle d'évaluation est

$$\frac{t \longrightarrow\!\!\!\!\!\rightarrow\ t' \qquad u[x \mapsto t'] \longrightarrow\!\!\!\!\!\rightarrow\ v}{(\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u) \longrightarrow\!\!\!\!\!\rightarrow\ v}\ (\textsc{let})$$

Branchements conditionnels. Pour ajouter les *branchements conditionnels*, on étend la syntaxe des termes par

$$t \quad ::= \quad \ldots \quad | \quad \texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t''$$

on ajoute la règle de typage

$$\frac{\Gamma \vdash t : \texttt{bool} \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A}{\Gamma \vdash \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ v : A}\ (\textsc{if})$$

on ajoute les règles d'évaluation

$$\frac{t \longrightarrow\!\!\!\!\!\rightarrow\ \texttt{true} \qquad u \longrightarrow\!\!\!\!\!\rightarrow\ u'}{\texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ v \longrightarrow\!\!\!\!\!\rightarrow\ u'} \qquad \frac{t \longrightarrow\!\!\!\!\!\rightarrow\ \texttt{false} \qquad v \longrightarrow\!\!\!\!\!\rightarrow\ v'}{\texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ v \longrightarrow\!\!\!\!\!\rightarrow\ v'}$$

et enfin les règles de réduction

$$\frac{t \longrightarrow t'}{\texttt{if}\ t\ \texttt{then}\ u_1\ \texttt{else}\ u_2 \longrightarrow \texttt{if}\ t'\ \texttt{then}\ u_1\ \texttt{else}\ u_2}$$

$$\frac{}{\texttt{if true then}\ u_1\ \texttt{else}\ u_2 \longrightarrow u_1}$$

$$\frac{}{\texttt{if false then}\ u_1\ \texttt{else}\ u_2 \longrightarrow u_2}$$

On peut alors montrer que les propriétés montrées au dessus sont préservées dans cette extension.

Un langage comme OCaml incorpore de nombreuses extensions qui ne seront pas traitées ici, à l'exception des définitions récursives (section 2.5). Citons entre autres

— les *références* (des variables que l'on peut modifier),
— les *exceptions* (pour traiter les cas exceptionnels),
— les *enregistrements* (qui sont des n-uplets dont les champs sont nommés),

— les *types inductifs*, les *variants polymorphes* (qui sont utilisés pour définir les types de données classiques comme les listes, les arbres, etc.),
— les *modules* (qui sont utilisés pour regrouper des fonctions),
— les *objets*,
— etc.

2.4 Sûreté du typage

Le typage sert à montrer l'absence d'erreurs à l'exécution : *well typed programs cannot go wrong* disait Milner. On cherche maintenant à formaliser cette intuition. Nous avons observé ci-dessus que toute valeur est une forme normale (théorème 35), mais la réciproque n'est pas vérifiée. Par exemple, le programme

```
add true
```

est une forme normale (il ne peut pas se réduire) mais n'est pas une valeur (au sens défini à la section 2.3.1). On peut considérer qu'un tel programme est indésirable : son calcul ne s'arrête pas parce qu'il est terminé, mais parce qu'on ne sait pas traiter ce cas. Nous allons voir que le typage permet d'éviter que de telles situations se produisent. Autrement dit, lorsqu'on exécute un programme typé, soit l'exécution est infinie, soit on aboutit à une valeur (théorème 45).

Nous commençons par montrer le lemme suivant, qui prouve que le typage est compatible avec la substitution :

Lemme 42. *Si* $\Gamma, x : A, \Gamma' \vdash t : B$ *et* $\Gamma, \Gamma' \vdash u : A$ *alors* $\Gamma, \Gamma' \vdash t[x \mapsto u] : B$.

Démonstration. Par induction sur la dérivation de $\Gamma, x : A, \Gamma' \vdash t : B$.
— Si la dernière règle est

$$\frac{}{\Gamma, x : A, \Gamma' \vdash \underline{n} : \texttt{int}}\ (\textsc{int})$$

alors on a immédiatement $\underline{n}[x \mapsto u] = \underline{n}$ et

$$\frac{}{\Gamma, \Gamma' \vdash \underline{n} : \texttt{int}}\ (\textsc{int})$$

Les autres constantes se traitent de façon similaire.
— Si la dernière est

$$\frac{}{\Gamma, x : A, \Gamma' \vdash y : A}\ (\textsc{var})$$

avec $x \neq y$ alors on conclut comme dans le cas précédent. Si la dernière règle est

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ (\textsc{var})$$

alors on conclut immédiatement avec la dérivation de $\Gamma, \Gamma' \vdash u : A$ donnée en hypothèse.
— Si la dernière règle est

$$\frac{\Gamma, x : A, \Gamma' \vdash t : A \Rightarrow B \qquad \Gamma, x : A, \Gamma' \vdash t' : A}{\Gamma, x : A, \Gamma' \vdash t\ t' : B}\ (\textsc{app})$$

on conclut par

$$\frac{\dfrac{\vdots}{\Gamma, \Gamma' \vdash t[x \mapsto u] : A \Rightarrow B} \qquad \dfrac{\vdots}{\Gamma, \Gamma' \vdash t'[x \mapsto u] : A}}{\Gamma, x : A, \Gamma' \vdash (t[x \mapsto u])\ (t'[x \mapsto u]) : B}\ (\textsc{app})$$

où les dérivations $\vdots$ sont données par hypothèse d'induction.
— Si la dernière règle est

$$\frac{\Gamma, x : A, \Gamma', y : B \vdash t : C}{\Gamma, x : A, \Gamma' \vdash \texttt{fun}\ y\ \texttt{->}\ t : B \Rightarrow C}\ (\textsc{fun})$$

quitte à renommer y, on peut toujours supposer $x \neq y$. On a donc

$$(\texttt{fun}\ \ y\ \ \to\ \ t)[x \mapsto u] = \texttt{fun}\ \ y\ \ \to\ \ (t[x \mapsto u])$$

et on conclut

$$\dfrac{\dfrac{\vdots}{\Gamma, \Gamma', y : B \vdash t[x \mapsto u] : C}}{\Gamma, \Gamma' \vdash \texttt{fun}\ \ y\ \ \to\ \ t[x \mapsto u] : B \Rightarrow C}\ (\textsc{fun})$$

où la dérivation $\vdots$ est donnée par hypothèse d'induction. [Notons que, à cause de ce cas, on ne pourrait pas montrer simplement le cas particulier où Γ' est vide.]

— Les autres cas sont laissés en exercice au lecteur. □

Montrons maintenant que le typage est préservé par la réduction (ceci est parfois appelé la *réduction du sujet*) :

Proposition 43. *Si $\Gamma \vdash t : A$ est dérivable et $t \longrightarrow t'$ alors $\Gamma \vdash t' : A$ est aussi dérivable.*

Démonstration. On raisonne par induction sur la dérivation de $t \longrightarrow t'$. Par exemple :

— Si la dernière règle est

$$\dfrac{t \longrightarrow t'}{t\ v \longrightarrow t'\ v}\ (\textsc{appl})$$

alors la dérivation de typage de $t\ v$ est nécessairement de la forme

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t : A \Rightarrow B} \qquad \dfrac{\vdots}{\Gamma \vdash v : A}}{\Gamma \vdash t\ v : B}\ (\textsc{app})$$

On a donc une dérivation de $\Gamma \vdash t : A \Rightarrow B$ et donc, par hypothèse d'induction, de $\Gamma \vdash t' : A \Rightarrow B$. On en déduit ainsi la dérivation de typage

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t' : A \Rightarrow B} \qquad \dfrac{\vdots}{\Gamma \vdash v : A}}{\Gamma \vdash t'\ v : B}\ (\textsc{app})$$

— Si la dernière règle est

$$\dfrac{}{(\texttt{fun}\ \ x\ \ \to\ \ t)\ v \longrightarrow t[x \mapsto v]}\ (\textsc{app})$$

alors la dérivation de typage de $(\texttt{fun}\ \ x\ \ \to\ \ t)\ v$ est nécessairement de la forme

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : A \vdash t : B}}{\Gamma \vdash \texttt{fun}\ \ x\ \ \to\ \ t : A \Rightarrow B}\ (\textsc{fun}) \qquad \dfrac{\vdots}{\Gamma \vdash v : A}}{\Gamma \vdash (\texttt{fun}\ \ x\ \ \to\ \ t)\ v : B}\ (\textsc{app})$$

On a donc une dérivation de $\Gamma, x : A \vdash t : B$ et de $\Gamma \vdash v : A$, et le théorème 42 nous assure que l'on a aussi une dérivation de

$$\Gamma \vdash t[x \mapsto v] : B$$

— Les autres cas sont laissés en exercice. □

On montre ensuite qu'un programme typé ne peut jamais être « bloqué » (ceci est parfois appelé le *progrès*) :

Proposition 44. *Si $\vdash t : A$ et t est une forme normale alors t est une valeur.*

Démonstration. Par récurrence sur la dérivation de $\vdash t : A$. Notons que la règle (VAR) ne peut être appliquée car l'environnement Γ est supposé vide, et t n'est donc pas une variable.

— Si la dernière règle est

$$\frac{x : A \vdash t : B}{\vdash \texttt{fun}\ x\ \to\ t : A \Rightarrow B}\ (\textsc{fun})$$

alors on a immédiatement que $\texttt{fun}\ x\ \to\ t$ est une valeur.

— Si la dernière règle est

$$\frac{\dfrac{\vdots}{\vdash t : A \Rightarrow B} \qquad \dfrac{\vdots}{\vdash u : A}}{\vdash t\ u : B}\ (\textsc{app})$$

Une réduction $t \longrightarrow t'$ induirait, par (APPL), une réduction $t\ u \longrightarrow t'u$, ce qui est exclu car $t\ u$ est supposé être en forme normale : t est donc aussi une forme normale. De même, u est une forme normale. Par hypothèse d'induction t et u sont donc des valeurs, et on en déduit que $t\ u$ est aussi une valeur par inspection de cas sur t (par exemple, t ne peut pas être de la forme $\texttt{fun}\ x\ \to\ t'$ sinon $t\ u$ ne serait pas normal).

— Les autres cas sont laissés en exercice au lecteur. □

On peut finalement montrer la propriété de *sûreté* des programmes bien typés :

Théorème 45. *Si t est un programme typé et $t \xrightarrow{*} u$, pour u une forme normale, alors u est une valeur.*

Démonstration. Supposons $\vdash t : A$. En utilisant la théorème 43, on montre aisément que l'on a $\vdash u : A$ et par la théorème 44 on en déduit que u est une valeur. □

Le théorème ci-dessus montre qu'un programme bien typé n'aboutira jamais à une erreur, dans le sens où il ne se réduit jamais dans une forme normale qui n'est pas une valeur. La maxime « les programmes bien typés ne font pas d'erreur » doit être comprise dans ce sens : certains programmes typés font des erreurs, mais celles-ci sont d'une autre sorte. On peut par exemple avoir des divisions par zéro au cours du calcul ou des programmes qui sont erronés parce qu'il ne font pas ce qu'on attend d'eux...

On peut se demander si la réciproque est vraie, la réponse est non : il existe des programmes qui ne se réduisent pas vers une forme normale qui n'est pas une valeur (autrement dit qui soit bouclent, soit se réduisent vers une valeur) mais qui ne sont pas bien typés. Par exemple, le programme

$$\texttt{fun}\ x\ \to\ \texttt{add true}$$

est une valeur (et donc en forme normale), mais n'est pas bien typé. Pour un exemple peut-être un peu plus parlant en OCaml, le programme

```
if true then 5 else false
```

se réduit vers la valeur `5`, mais n'est pas bien typé.

2.5 Points fixes

On a vu au théorème 45, que tout programme bien typé soit se réduit de façon infinie, soit atteint une valeur. Cependant, on sera bien en peine de trouver un exemple de programme qui boucle dans ce langage : on peut en fait montrer (et c'est non-trivial) que tous les programmes du langage terminent (voir le cours sur le λ-*calcul simplement typé*). La raison pour laquelle nous avons formulé les choses de cette façon est que les mêmes théorèmes vont être conservés si on considère des extensions du langage, y compris des extensions qui permettent d'écrire des programmes qui ne terminent pas. De telles extensions sont nécessaires pour rendre le langage raisonnablement expressif, comme ajouter la possibilité de définir des fonctions de façon récursive (on pourrait aussi penser ajouter la possibilité de faire des boucles `for` ou `while`, mais celles-ci sont moins naturelles dans le cadre d'un langage fonctionnel pur). Par exemple, la fonction factorielle peut être définie à l'aide de la fonction récursive suivante :

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1)
```

2.5.1 Opérateurs de points fixes en OCaml

Plutôt que d'ajouter directement la possibilité de définir des fonctions récursives, il s'avère plus simple d'ajouter un opérateur de point fixe, qui permettra de définir des fonctions récursives. En mathématiques, un point fixe d'une fonction $f : A \to A$ est un élément $a \in A$ tel que $f(a) = a$. De même, dans un langage de programmation, un *point fixe* d'un programme f est un terme a tel que

$$f\ a \overset{*}{\longleftrightarrow} a$$

où $\overset{*}{\longleftrightarrow}$ est la relation d'équivalence engendrée par la réduction $\longrightarrow$. En pratique, on cherchera souvent un terme a tel que

$$a \overset{*}{\longrightarrow} f\ a$$

Un *opérateur de point fixe* est une fonction `fix` qui, à toute fonction f, associe un point fixe `fix` f.

En OCaml, on peut par exemple définir un opérateur de point fixe par

```
let rec fix f = f (fix f)
```

On peut alors se convaincre que, une fois cette fonction définie, on peut implémenter n'importe quelle fonction récursive sans utiliser le mot clé `rec`. Par exemple, montrons comment implémenter la fonction factorielle définie ci-dessus. Partant de la définition récursive, nous appliquons la transformation suivante :

— on enlève le mot clé `rec`,
— on ajoute un nouvel argument `f` à la fonction,
— on remplace chaque appel récursif par un appel à `f`.

On obtient alors la fonction suivante, appelée une *fonctionnelle* :

```
let fact_fun f n =
  if n = 0 then 1 else n * f (n-1)
```

On s'attend alors à pouvoir définir la factorielle comme le point fixe de cette fonctionnelle:

```
let fact = fix fact_fun
```

En effet, cela revient à appliquer la fonction `fact_fun` à l'argument `fix fact_fun`, c'est-à-dire précisément à la fonction `fact` que nous sommes en train de définir, et les appels « récusifs » à `f` seront donc bien des appels à `fact`. Cependant, si on essaye d'exécuter la définition de `fact` ci-dessus, on obtient le message d'erreur

```
Stack overflow during evaluation (looping recursion?).
```

qui est dû au fait que OCaml essaye de procéder à l'évaluation infinie suivante :

$$\begin{aligned}
\texttt{fix fact_fun} &\longrightarrow \texttt{fact_fun (fix fact_fun)} \\
&\longrightarrow \texttt{fact_fun (fact_fun (fix fact_fun))} \\
&\longrightarrow \texttt{fact_fun (fact_fun (fact_fun (fix fact_fun)))} \\
&\longrightarrow \ldots
\end{aligned}$$

Cependant, on peut régler cela en ajoutant un argument supplémentaire à notre définition de l'opérateur de point fixe :

```
let rec fix f x = f (fix f) x
```

Cette transformation ne change pas fondamentalement la définition de `fix`, mais elle change la façon dont OCaml l'évalue : dans la définition

```
let fact_fun f n =
  if n = 0 then 1 else n * f (n-1)
```

l'évaluation est bloquée car la fonction attend maintenant un second argument, et on peut vérifier que notre programme a bien le comportement attendu :

```
let () = assert (fact 5 = 120)
```

2.5.2 Opérateur de point fixe en mini-ML

Nous allons montrer comment ajouter un opérateur de point fixe en mini-ML. On étend la syntaxe des termes par

$$t \quad ::= \quad \ldots \quad | \quad \texttt{fix}$$

on étend la définition des valeurs par

$$v \quad ::= \quad \ldots \quad | \quad \texttt{fix}$$

on ajoute la règle d'évaluation

$$\frac{t \longrightarrow\!\!\!\!\!\rightarrow \texttt{fun}\ x\ \rightarrow\ t' \qquad t'[x \mapsto \texttt{fix}\, t'] \longrightarrow\!\!\!\!\!\rightarrow v}{\texttt{fix}\, t \longrightarrow\!\!\!\!\!\rightarrow v}$$

on ajoute les règles de réduction

$$\frac{t \longrightarrow t'}{\texttt{fix}\, t \longrightarrow \texttt{fix}\, t'} \qquad\qquad \frac{}{\texttt{fix}\, (\texttt{fun}\ x\ \rightarrow\ t) \longrightarrow t[x \mapsto \texttt{fix}\, (\texttt{fun}\ x\ \rightarrow\ t)]}$$

et on ajoute la règle de typage

$$\frac{}{\Gamma \vdash \texttt{fix} : (A \Rightarrow A) \Rightarrow A}\ (\textsc{fix})$$

L'évaluation et la réduction sont encore déterministes (théorème 29, théorème 36) et coïncident sur les valeurs (théorème 41).

Notons que dans ce langage il existe maintenant des termes typables qui ne terminent pas, c'est-à-dire qui donnent lieu à des suites de réductions infinies. Par exemple, le terme

$$\texttt{fix}\, (\texttt{fun}\ x\ \rightarrow\ x)$$

a le type A pour tout type A :

$$\cfrac{\cfrac{\cfrac{}{x : A \vdash x : A}\ (\textsc{var})}{\vdash \texttt{fun}\ x\ \rightarrow\ x : A \Rightarrow A}\ (\textsc{fun})}{\vdash \texttt{fix}\, (\texttt{fun}\ x\ \rightarrow\ x) : A}\ (\textsc{fix})$$

et donne lieu à une suite de réductions infinie :

$$\texttt{fix}\, (\texttt{fun}\ x\ \rightarrow\ x) \longrightarrow (\texttt{fun}\ x\ \rightarrow\ x)\, (\texttt{fix}\, (\texttt{fun}\ x\ \rightarrow\ x)) \longrightarrow \texttt{fix}\, (\texttt{fun}\ x\ \rightarrow\ x) \longrightarrow \cdots$$

Cependant, les théorèmes énoncés pour le système de types, en particulier la sûreté (théorème 45) sont encore valides dans cette extension, ce qui est la raison pour laquelle nous avons fait attention à ne jamais utiliser le fait que tous les termes terminent dans la version du langage sans `fix`.

Comme expliqué ci-dessus, on peut maintenant définir la factorielle dans notre langage :

$$\texttt{fix}\, (\texttt{fun}\ f\ \rightarrow\ \texttt{fun}\ n\ \rightarrow\ \texttt{if}\ n = \texttt{0}\ \texttt{then 1 else}\ n \times (f\ (n - \texttt{1})))$$

2.6 Inférence de type

Nous avons vu à la section 2.2.5 que, pour la variante du langage avec annotations de type, il y a unicité du type (théorème 27), et il n'est pas très difficile de voir que ce type (lorsqu'il existe) peut être calculé par induction sur le terme en utilisant les règles de la figure 5, c.f. section B.

Nous allons maintenant nous intéresser à la variante sans annotations de types. Dans ce langage, le type n'est clairement plus unique. Par exemple, le terme correspondant à la fonction identité

$$\texttt{fun}\ x\ \rightarrow\ x$$

admet les type $\texttt{int} \Rightarrow \texttt{int}$, $\texttt{bool} \Rightarrow \texttt{bool}$ et plus généralement $A \Rightarrow A$ pour tout type A. Afin de prendre ceci en compte, nous allons généraliser la syntaxe des types en y ajoutant des variables et généraliser l'inférence de type à ce cadre.

2.6.1 Types principaux

On suppose fixé un ensemble dénombrable de *variables de types* $X, Y, \ldots$ et on définit maintenant les types par la syntaxe suivante :

$$A \quad ::= \quad X \quad | \quad \texttt{int} \quad | \quad \texttt{bool} \quad | \quad A \Rightarrow A' \quad | \quad A \times A'$$

où X désigne une variable de type arbitraire. Les règles de typage sont inchangées et restent celles données à la figure 5.

Une *substitution* σ est une fonction des variables de types dans les types telle que $\sigma(X) = X$ pour tout variable X sauf un nombre fini d'entre elles. Notons que l'identité est une substitution et que la composée de substitutions est encore une substitution. On note parfois

$$\sigma \quad = \quad [X_1 \mapsto A_1, \ldots, X_n \mapsto A_n]$$

la substitution σ telle que $\sigma(X_i) = A_i$ (et $\sigma(X) = X$ pour X n'étant pas l'un des X_i). Une substitution est un *renommage* lorsqu'elle est injective et que l'image de toute variable est une variable. Étant donné un type A, on note $A[\sigma]$ le type A où toute variable X a été remplacée par $\sigma(X)$ (et de même on note $\Gamma[\sigma]$ un contexte où on a appliqué la substitution σ à tous les types). On note aussi parfois $A[B/X]$ pour le type A où l'on a replacé toutes les occurrence de la variable X par B.

On note $A \sqsubseteq B$ lorsqu'il existe une substitution σ telle que $A[\sigma] = B$. Par exemple,

$$X \Rightarrow X \qquad \sqsubseteq \qquad (\texttt{int} \Rightarrow Y) \Rightarrow (\texttt{int} \Rightarrow Y)$$

Lemme 46. *La relation $\sqsubseteq$ est un pré-ordre sur les types (c'est-à-dire que cette relation est réflexive et transitive). Deux types A et B sont équivalents (c'est-à-dire que l'on a $A \sqsubseteq B$ et $B \sqsubseteq A$) si et seulement si il existe un renommage bijectif σ tel que $A[\sigma] = B$.*

On peut remarquer que le typage est stable par substitution :

Proposition 47. *Si $\Gamma \vdash t : A$ est dérivable alors $\Gamma[\sigma] \vdash t : A[\sigma]$ est aussi dérivable, et ce pour toute substitution σ.*

Démonstration. Par induction sur la dérivation de $\Gamma \vdash t : A$.

— Si la dernière règle est

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{var})$$

alors on a immédiatement

$$\frac{(x : A[\sigma]) \in \Gamma[\sigma]}{\Gamma[\sigma] \vdash x : A[\sigma]}\ (\textsc{var})$$

— Si la dernière règle est

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\ x\ \to\ t : A \Rightarrow B}\ (\textsc{fun})$$

par hypothèse d'induction on a une dérivation de $\Gamma[\sigma], x : A[\sigma] \vdash t : B[\sigma]$ et on conclut par

$$\frac{\Gamma[\sigma], x : A[\sigma] \vdash t : B[\sigma]}{\Gamma[\sigma] \vdash \texttt{fun}\ x\ \to\ t : A[\sigma] \Rightarrow B[\sigma]}\ (\textsc{fun})$$

— Si la dernière règle est

$$\frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\ (\textsc{app})$$

par hypothèse d'induction on a une dérivation de $\Gamma[\sigma] \vdash t : A[\sigma] \Rightarrow B[\sigma]$ et de $\Gamma[\sigma] \vdash u : A[\sigma]$ et on conclut par

$$\frac{\Gamma[\sigma] \vdash t : A[\sigma] \Rightarrow B[\sigma] \qquad \Gamma[\sigma] \vdash u : A[\sigma]}{\Gamma[\sigma] \vdash t\ u : B[\sigma]}\ (\textsc{app})$$

— Les autres cas sont similaires. □

Un type A pour un terme t est dit *principal* lorsque $\vdash t : A$ est dérivable $\vdash t : B$ est dérivable si et seulement si $A \sqsubseteq B$, c'est-à-dire qu'il engendre tous les types de t par substitution (en particulier, A est nécessairement un type pour t car $A \sqsubseteq A$). Par exemple, $X \Rightarrow X$ est le type principal de `fun` $x \to x$. Comme dans la phrase précédente, on parle parfois abusivement *du* type principal d'un terme au lieu *d'un* type principal : ceci est justifié par le théorème 46 qui nous assure que deux types principaux ne diffèrent que par un renommage des variables.

Il sera utile de généraliser la définition de type principal à un contexte quelconque. Étant donné un contexte Γ et un type t, un *type principal pour t dans le contexte* Γ est une paire consistant en une substitution σ et un type A tels que

— $\Gamma[\sigma] \vdash t : A$ est dérivable,

— pour toute substitution τ telle que $\Gamma[\tau] \vdash t : B$ est dérivable, il existe une substitution τ' telle que $\tau = \tau' \circ \sigma$ et $B = A[\tau']$.

De façon informelle, il s'agit d'une spécialisation du contexte Γ qui rende t typable, la plus générale possible. On pourra vérifier qu'on retrouve la définition donnée à la section précédente en considérant le cas où Γ est le contexte vide.

Exemple 48. Dans un contexte Γ état $f : X \Rightarrow (X \Rightarrow X), a : Y \times Y$, le terme $f\ a$ admet le type principal formé de

— la substitution $\sigma = [X \mapsto Y \times Y]$

— le type $A = (Y \times Y) \Rightarrow (Y \times Y)$

La substitution σ exprime le fait que, pour que la fonction f puisse être appliquée à a il faut au moins que le type X de son argument soit le même que celui de a, c'est-à-dire $Y \times Y$. Dans ce cas, le type du résultat est le type A donné ci-dessus.

2.6.2 Typage comme résolution d'un système d'équations

Nous allons voir que dans notre système, un terme typable admet toujours un type principal et donner un algorithme qui détermine ce type principal. Cet algorithme va procéder en deux étapes :

1. nous allons d'abord associer à un terme un système d'équations sur les types dont les solutions vont fournir les types possible du terme,
2. nous allons ensuite montrer que lorsque ce système admet une solution il en admet toujours une qui est la « plus générale », que l'on peut calculer, et qui correspond au type le plus général du terme.

Un *système d'équations de types* E est un ensemble fini de paires (A, B) de types, que l'on notera $A \stackrel{?}{=} B$:

$$E \quad = \quad \{A_1 \stackrel{?}{=} B_1, \ldots, A_n \stackrel{?}{=} B_n\}$$

Une substitution σ est une *solution* de E (ou encore un *unificateur* de E) lorsque $A_i[\sigma] = B_i[\sigma]$ pour tout indice i. On note $\bot$ un système d'équations arbitraire qui n'admet pas de solution, par exemple $\bot = \{(X \Rightarrow Y) \stackrel{?}{=} (X \times Y)\}$.

Algorithme 49. À un un contexte Γ et un terme t, nous associons un ensemble E_t et un type A_t, par induction sur le terme t (sauf mention explicite contraire, l'environnement est toujours Γ).

— Si le terme est un entier $\underline{n}$, on définit

$$E_{\underline{n}} = \emptyset \qquad\qquad A_{\underline{n}} = \texttt{int}$$

— Si le terme est un booléen $\underline{b}$, on définit

$$E_{\underline{b}} = \emptyset \qquad\qquad A_{\underline{b}} = \texttt{bool}$$

— Si le terme est une variable x avec $x \in \mathrm{dom}(\Gamma)$, on définit $E_x = \emptyset$ et $A_x = \Gamma(x)$. Si $x \notin \mathrm{dom}(\Gamma)$ alors on définit $E_x = \bot$ A_x un type arbitraire (par exemple $A_x = X$).

— Si le terme est une abstraction `fun` $x \to t$. On note E_t et A_t la paire produite pour t dans le contexte $\Gamma, x : X$, où X est une variable fraîche et on définit

$$E_{\texttt{fun } x \to t} = E_t \qquad\qquad A_{\texttt{fun } x \to t} = X \Rightarrow A_t$$

$$\frac{}{\Gamma \vdash \underline{n} : \texttt{int} \mid} \text{(INT)} \qquad \frac{}{\Gamma \vdash \underline{b} : \texttt{bool} \mid} \text{(BOOL)}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid} \text{(VAR)} \qquad \frac{}{\Gamma \vdash \texttt{add} : \texttt{int} \times \texttt{int} \Rightarrow \texttt{int} \mid} \text{(ADD)}$$

$$\frac{\Gamma, x : X \vdash t : A_t \mid E_t}{\Gamma \vdash \texttt{fun}\ x\ \rightarrow\ t : X \Rightarrow A_t \mid E_t} \text{(FUN)} \qquad \frac{\Gamma \vdash t : A_t \mid E_t \qquad \Gamma \vdash u : A_u \mid E_u}{\Gamma \vdash t\ u : X \mid E_t, E_u, A_t \stackrel{?}{=} (A_u \Rightarrow X)} \text{(APP)}$$

$$\frac{\Gamma \vdash t : A_t \mid E_t \qquad \Gamma \vdash u : A_u \mid E_u}{\Gamma \vdash (t,u) : A_t \times A_u \mid E_t, E_u} \text{(PAIR)} \qquad \frac{}{\Gamma \vdash \texttt{fst} : X \times Y \Rightarrow X \mid} \text{(FST)}$$

$$\frac{}{\Gamma \vdash \texttt{snd} : X \times Y \Rightarrow Y \mid} \text{(SND)}$$

FIGURE 8 – Génération d'un système d'équations.

— Si le terme est une application $t\ u$, on définit

$$E_{t\ u} = E_t \cup E_u \cup \{A_t \stackrel{?}{=} A_u \Rightarrow X\} \qquad A_{t\ u} = X$$

où X est une variable fraîche.

— Si le terme est une paire (t,u), on définit

$$E_{(t,u)} = E_t \cup E_u \qquad A_{(t,u)} = A_t \times A_u$$

— Si le terme est `fst`, on définit

$$E_{\texttt{fst}} = \emptyset \qquad A_{\texttt{fst}} = (X \times Y) \Rightarrow X$$

où X et Y sont des variables fraîches (le cas `snd` est similaire).

— Si le terme est `fix`, on définit

$$E_{\texttt{fix}} = \emptyset \qquad A_{\texttt{fix}} = (X \Rightarrow X) \Rightarrow X$$

Ci-dessus, par une variable *fraîche*, on entend une variable qui n'apparaît pas ailleurs (dans les types, contextes et systèmes d'équations considérés).

L'algorithme ci-dessus peut être reformulé sous forme de règles d'inférences qui manipulent des séquents de la forme $\Gamma \vdash t : A_t \mid E_t$ qui signifient que le terme t dans le contexte Γ produit le système d'équations E_t et le type A_t. Les règles sont données à la figure 8, et correspondent immédiatement aux différents cas de l'algorithme décrit ci-dessus.

On peut alors formuler les deux propriétés suivantes de l'algorithme qui montre que le système d'équations produit décrit exactement les types possibles pour le terme original. On suppose fixés un contexte Γ et un terme t et on note E_t et A_t le système d'équations et le type produits par l'algorithme (autrement dit, on suppose $\Gamma \vdash t : A_t \mid E_t$ dérivable par les règles de la figure 8).

Proposition 50 (Correction)**.** *Si σ est une solution de E_t alors $A_t[\sigma]$ est un type pour t dans le contexte $\Gamma[\sigma]$ (i.e. $\Gamma[\sigma] \vdash t : A_t[\sigma]$ est dérivable par les règles de la figure 5).*

Démonstration. Par induction sur t. □

Proposition 51 (Complétude)**.** *Pour toute substitution σ et type A tels que $\Gamma[\sigma] \vdash t : A$ est dérivable, σ est une solution de E_t et on a $A = A_t[\sigma']$ pour une substitution σ' qui coïncide avec σ sur les variables apparaissant dans Γ.*

Démonstration. Par induction sur la dérivation de $\Gamma[\sigma] \vdash t : A$. □

2.6.3 Résolution des systèmes d'équations

Nous allons maintenant présenter un algorithme permettant de résoudre un système d'équations E. Plus exactement, notre algorithme va soit renvoyer la solution la plus générale de E, soit échouer lorsque E n'admet pas de solution. Ici, par une *solution la plus générale* de E on entend une substitution σ qui est une solution de E et telle que, pour toute solution τ de E, il existe une substitution τ' telle que $\tau = \tau' \circ \sigma$. Comme au théorème 46, on peut montrer que deux solutions les plus générales ne diffèrent que par un renommage.

Algorithme 52. Étant donné un système d'équations E, nous définissons (partiellement) la substitution $\mathrm{unify}(E)$ par

$$\mathrm{unify}(\mathtt{int} \stackrel{?}{=} \mathtt{int}, E) = \mathrm{unify}(E) \tag{1}$$

$$\mathrm{unify}(\mathtt{bool} \stackrel{?}{=} \mathtt{bool}, E) = \mathrm{unify}(E) \tag{2}$$

$$\mathrm{unify}(X \stackrel{?}{=} X, E) = \mathrm{unify}(E) \tag{3}$$

$$\mathrm{unify}((A \Rightarrow B) \stackrel{?}{=} (A' \Rightarrow B'), E) = \mathrm{unify}(A \stackrel{?}{=} A', B \stackrel{?}{=} B', E) \tag{4}$$

$$\mathrm{unify}((A \times B) \stackrel{?}{=} (A' \times B'), E) = \mathrm{unify}(A \stackrel{?}{=} A', B \stackrel{?}{=} B', E) \tag{5}$$

$$\mathrm{unify}(X \stackrel{?}{=} A, E) = \mathrm{unify}(E[X \mapsto A]) \circ [X \mapsto A] \qquad \text{si } X \notin \mathrm{VL}(A) \tag{6}$$

$$\mathrm{unify}(A \stackrel{?}{=} X, E) = \mathrm{unify}(E[X \mapsto A]) \circ [X \mapsto A] \qquad \text{si } X \notin \mathrm{VL}(A) \tag{7}$$

$$\mathrm{unify}(\emptyset) = \mathrm{id} \tag{8}$$

Dans tous les autres cas unify échoue.

Dans (6) et (7), on s'autorise la notation $E[\sigma]$ pour l'application d'une substitution σ à tous les types apparaissant dans un système d'équations E. On note aussi $\mathrm{VL}(A)$ l'ensemble des variables apparaissant dans le type A. La condition de bord de (6) et (7) est importante : sans celle-ci, on aurait

$$\mathrm{unify}(X \stackrel{?}{=} (X \Rightarrow X)) = [X \mapsto (X \Rightarrow X)]$$

mais cette substitution n'est pas une solution du système d'équation (celui-ci n'admet pas de solution, voir la théorème 54).

Montrons d'abord que c'est bien un algorithme, dans le sens où on obtient toujours en temps fini soit une substitution, soit un échec. Ceci n'est pas évident : on peut voir unify comme étant une fonction définie en faisant des appels récursif à elle-même et il n'est pas exclu qu'on puisse avoir une suite infinie d'appels récursifs. En particulier, les appels récursifs ne font pas toujours décroître le nombre d'équations dans E, car (4) et (5) les font augmenter. En réalité, il faut prendre une notion plus subtile de « taille » pour les équations. Nous allons prendre en compte les deux mesures suivantes.

— On note $\mathrm{VL}(E)$ l'ensemble des variables apparaissant dans les types de E et $|E|_{\mathrm{var}} = |\mathrm{VL}(E)|$ le nombre de variables qui apparaissent dans E.

— Étant donné un type A, on note $|A|_{\mathrm{op}}$ le nombre d'opérateurs qu'il utilise, ce qui est défini inductivement par

$$|\mathtt{int}|_{\mathrm{op}} = |\mathtt{bool}|_{\mathrm{op}} = |X|_{\mathrm{op}} = 1 \qquad |A \Rightarrow B|_{\mathrm{op}} = |A \times B|_{\mathrm{op}} = 1 + |A|_{\mathrm{op}} + |B|_{\mathrm{op}}$$

et par extension, on note

$$|E|_{\mathrm{op}} = \sum_{A \stackrel{?}{=} B \in E} |A|_{\mathrm{op}} + |B|_{\mathrm{op}}$$

la somme des tailles des types apparaissant dans E.

Proposition 53. *L'algorithme 52 termine : quelle que soit l'entrée il produit en temps fini une substitution ou un échec.*

Démonstration. On définit la taille d'un système d'équations de E comme la paire d'entiers

$$|E| = (|E|_{\text{var}}, |E|_{\text{op}})$$

On compare une paire d'entiers par l'ordre lexicographique défini par $(m, n) \leqslant_{\text{lex}} (m', n')$ si et seulement si $m < m'$, ou $m = m'$ et $n \leqslant n'$. Il n'existe pas de suite strictement décroissante dans $\mathbb{N} \times \mathbb{N}$ muni de cet ordre (sinon, il existerait une suite strictement décroissante dans l'une des composantes $\mathbb{N}$ avec l'ordre usuel). Comparons dans chacun des cas la taille du système d'équations de l'appel récursif avec la taille du système d'équations originel :

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
$\lvert E\rvert_{\text{var}}$	$=$	$=$	$\leqslant$	$=$	$=$	$<$	$<$
$\lvert E\rvert_{\text{op}}$	$<$	$<$	$<$	$<$	$<$		
$\lvert E\rvert$	$<$	$<$	$<$	$<$	$<$	$<$	$<$

Dans tous les cas la taille $|E|$ du système d'équations E diminue strictement : si on avait une suite infinie d'appels récursifs, on aurait une suite infinie strictement décroissante dans $\mathbb{N} \times \mathbb{N}$ (avec l'ordre $\leqslant_{\text{lex}}$) ce qui est exclu d'après ce qui précède. □

Montrons maintenant qu'il produit les bonnes solutions. On commence par montrer qu'il refuse dans les bons cas :

Proposition 54. *Si l'algorithme 52 échoue alors le système n'admet pas de solution.*

Démonstration. Si l'algorithme échoue alors le système donné en entrée est nécessairement de la forme $A \stackrel{?}{=} B, E$ et on a deux cas.

- Soit A et B sont de forme différente (par exemple $A = \texttt{int}$ et $B = \texttt{bool}$, ou $A = \texttt{int}$ et $B = B_1 \Rightarrow B_2$, ou $A = A_1 \Rightarrow A_2$ et $B = B_1 \times B_2$) auquel cas le système n'admet clairement pas de solution, puisqu'une solution σ devrait vérifier $A[\sigma] = B[\sigma]$, ce qui est impossible.
- Soit A est une variable X et B est un type distinct de X dans lequel la variable apparaît (ou la situation similaire où les rôles de A et B sont échangés), par exemple $B = X \Rightarrow Y$. Une solution σ doit vérifier $X[\sigma] = B[\sigma]$. Comme B n'est pas une variable et que X y apparaît, on a nécessairement $|X[\sigma]|_{\text{op}} < |B[\sigma]|_{\text{op}}$, contradiction. Par exemple, avec $B = X \Rightarrow Y$,

$$|(X \Rightarrow Y)[\sigma]| = |X[\sigma] \Rightarrow Y[\sigma]| = 1 + |X[\sigma]| + |Y[\sigma]| > |X[\sigma]|$$ □

Les premières transformations, sont valides dans le sens où elles préservent l'ensemble des solutions.

Lemme 55. *Pour les équations (1) à (5), de la forme* $\text{unify}(E) = \text{unify}(E')$*, les systèmes d'équations E et E' admettent les mêmes solutions.*

Démonstration. Immédiat par inspection des différents cas. □

Le cas de l'équation (6) (ainsi que le cas de (7) qui est similaire) est plus subtil et est traité par le lemme suivant.

Lemme 56. *Une substitution σ est une solution du système $(X \stackrel{?}{=} A, E)$ avec $X \notin \text{VL}(A)$ si et seulement si on a $\sigma = \tau \circ [X \mapsto A]$, pour τ une solution de $E[X \mapsto A]$.*

Démonstration. On a la suite d'équivalences suivantes :

(i) σ est une solution $(X \stackrel{?}{=} A, E)$,

(ii) $\sigma(X) = A[\sigma]$ et σ est une solution de E,

(iii) $\sigma(X) = A[\sigma]$ et σ est une solution de $E[X \mapsto A]$,

(iv) σ est de la forme $\tau \circ [X \mapsto A]$ pour τ une solution de $E[X \mapsto A]$.

Pour l'équivalence entre (ii) et (iii), on utilise le lemme suivant, simple à montrer : étant donnés deux types A et B et une substitution σ, si $\sigma(X) = A[\sigma]$ alors $B[\sigma] = [X \mapsto A][\sigma]$. Détaillons l'équivalence entre (iii) et (iv).

(iii) ⇒ (iv) Si $\sigma(X) = A[\sigma]$ et σ est une solution de $E[X \mapsto A]$ alors on définit τ par $\tau(X) = X$ et $\tau(Y) = \sigma(Y)$ pour $Y \neq X$. On a bien $\sigma = \tau \circ [X \mapsto A]$ car

$$\begin{aligned}(\tau \circ [X \mapsto A])(X) &= A[\tau] = A[\sigma] && \text{car } X \notin \mathrm{VL}(A)\\ (\tau \circ [X \mapsto A])(Y) &= \tau(Y) = \sigma(Y) && \text{pour } Y \neq X\end{aligned}$$

et τ est une solution de $E[X \mapsto A]$ car σ est en est une solution et X n'y apparaît pas.

(iv) ⇒ (iii) Si $\sigma = \tau \circ [X \mapsto A]$ pour τ une solution de $E[X \mapsto A]$. On a d'une part $\sigma(X) = A[\tau] = A[\sigma]$, car X n'apparaît pas dans A. Et d'autre part σ est une solution de $E[X \mapsto A]$ car τ en est une et que X n'apparaît pas dans le système d'équations. □

Enfin la correction du cas de base, qui correspond à l'équation (8) est donné par le lemme suivant :

Lemme 57. *La solution la plus générale du système vide* $\emptyset$ *est* id.

Démonstration. Toute substitution σ est solution du système vide, et on a bien $\sigma = \sigma \circ \mathrm{id}$. □

On en déduit alors la correction de l'algorithme 52 (la complétude étant donnée par la théorème 54) :

Théorème 58. *Étant donné un système d'équation* E*, si* unify(E) *n'échoue pas alors il produit une substitution la plus générale de* E*, qui est telle que* $\sigma(X) = X$ *pour toute variable* X *n'apparaissant pas dans* E*.*

Démonstration. On montre la propriété par récurrence sur les appels récursifs, ce qui est justifié par la théorème 53. C'est vérifié pour le cas de base par le théorème 57, les cas (1) à (5) sont traités par le théorème 55 et les cas (6) et (7) sont conséquence du théorème 56 (qui utilise l'hypothèse $\sigma(X) = X$ pour X n'apparaissant pas dans E). □

Notons que l'algorithme d'*unification* n'est pas vraiment spécifique aux cas des équations de types. De fait, il se généralise à des équations arbitraires entre termes.

2.6.4 Polymorphisme

Dans le système précédent, on voit qu'on peut donner plusieurs types à un terme donné. Par exemple, le programme

$$(\texttt{(fun } x \to x\texttt{) 5,(fun } x \to x\texttt{) true)}$$

est typable car on peut donner à la fois les types `int` ⇒ `int` et `bool` ⇒ `bool` aux deux sous-programmes `fun` $x \to x$. Cependant, on ne peut pas utiliser un même sous-programme avec deux types différents. Par exemple, le programme

$$\texttt{let } f \texttt{ = (fun } x \to x\texttt{) in (}f\texttt{ 5,}f\texttt{ true)}$$

c'est-à-dire (voir la section 2.3.6)

$$\texttt{(fun } f \to \texttt{(}f\texttt{ 5,}f\texttt{ true))}\,\texttt{(fun } x \to x\texttt{)}$$

n'est *pas* typable, car il faudrait donner *à la fois* le type `int` ⇒ `int` et le type `bool` ⇒ `bool` à la variable f.

L'idée, employée dans les langages tels que OCaml, et due à Milner est d'utiliser non pas des types mais des *schémas de types*, c'est-à-dire des types dont certaines variables sont quantifiées universellement de façon prénexe, autrement dit des "types" de la forme

$$\forall X_1 \ldots \forall X_n . A$$

où A est un type au sens précédent, dans lequel les variables X_i peuvent apparaître dans A et sont liée par les quantifications universelles (on s'autorisera à les renommer). Par exemple, l'identité aura le type

$$\forall X.(X \Rightarrow X)$$

$$\frac{(x : B) \in \Gamma \qquad A \preceq B}{\Gamma \vdash x : A}\ (\textsc{var}) \qquad \frac{\Gamma \vdash t : A \qquad \Gamma, x : \forall_\Gamma A \vdash u : B}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u : B}\ (\textsc{let})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\ x\ \rightarrow\ t : A \Rightarrow B}\ (\textsc{fun}) \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\ (\textsc{app})$$

FIGURE 9 – Règles de typage polymorphe.

(la notation en OCaml est `'a -> 'a`) qui signifie qu'il peut être utilisé avec n'importe quelle variable de type X.

Les règles de typage avec polymorphisme sont données à la figure 9 (du moins celles qui changent par rapport à la figure 5). Les contextes Γ sont maintenant constitués de paires $x_i : A_i$ où les A_i sont des schémas de types (et non plus des types). Étant donné un type A et un schéma de type $\forall X_1 \ldots \forall X_n . B$, on écrit

$$A \quad \preceq \quad \forall X_1 \ldots \forall X_n . B$$

lorsqu'il existe des types A_1, …, A_n tels que

$$A \quad = \quad B[X_1 \mapsto A_1, \ldots, X_n \mapsto A_n]$$

On dit alors que A est une *instance* du schéma de type. Dans la règle (LET), le schéma de type $\forall_\Gamma A$ est la *généralisation* du type A vis-à-vis de Γ, défini par

$$\forall_\Gamma A \quad = \quad \forall X_1 \ldots \forall X_n . B$$

où $\mathrm{VL}(A) \setminus \mathrm{VL}(\Gamma) = \{X_1, \ldots, X_n\}$. On a par exemple

$$\dfrac{\dfrac{}{x : X \vdash x : X}\ (\textsc{var}) \qquad \dfrac{\dfrac{\dfrac{}{f : \forall X.X \Rightarrow X \vdash f : \texttt{bool} \Rightarrow \texttt{bool}}\ (\textsc{var}) \qquad \dfrac{}{\ldots \vdash \texttt{true} : \texttt{bool}}\ (\textsc{bool})}{f : \forall X.X \Rightarrow X \vdash f\ \texttt{true}}\ (\textsc{app}) \qquad \dfrac{\vdots}{f : \forall X.X \Rightarrow X \vdash f\ 1 : \texttt{int}}}{f : \forall X.X \Rightarrow X \vdash (f\ \texttt{true}, f\ 1) : \texttt{bool} \times \texttt{int}}\ (\textsc{pair})}{\vdash \texttt{fun}\ x\ \rightarrow\ x : X \Rightarrow X}\ (\textsc{fun})}{\vdash \texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \rightarrow\ x\ \texttt{in}\ (f\ \texttt{true}, f\ 1)}\ (\textsc{let})$$

Lors de la généralisation, on ne quantifie pas sur les variables du contexte Γ. Si l'on s'autorisait à quantifier sur toutes les variables, on aurait des dérivations incorrectes telles que

$$\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X}\ (\textsc{var}) \qquad \dfrac{}{x : X, y : \forall X.X \vdash y : Y}\ (\textsc{var})}{x : X \vdash \texttt{let}\ y\ \texttt{=}\ x\ \texttt{in}\ y : Y}\ (\textsc{let})}{\vdash \texttt{fun}\ x\ \rightarrow\ \texttt{let}\ y\ \texttt{=}\ x\ \texttt{in}\ y : X \Rightarrow Y}\ (\textsc{fun})$$

où on donne le type $X \Rightarrow Y$ à une fonction qui est essentiellement l'identité.

3 Bibliographie

Voici les principales références sur le sujet, qui ont bien sûr été utilisées pour préparer ces notes.

Sémantique opérationnelle :

— Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*, The MIT Press, 1993.
`https://www.cin.ufpe.br/~if721/intranet/TheFormalSemanticsofProgrammingLanguages.pdf`
— Hanne Riis Nielson, Flemming Nielson, *Semantics with Applications*, Springer, 2006.
`http://www.cs.ru.nl/~herman/onderwijs/semantics2019/wiley.pdf`

Typage :

— Xavier Leroy, cours *Typage et Programmation*.
`https://xavierleroy.org/dea/typage/`
— Benjamin Pierce, *Types and Programming Languages*, MIT Press, 2002.
`https://github.com/MPRI/M2-4-2/raw/master/Types%20and%20Programming%20Languages.pdf`

A Évaluation de IMP

On implémente les expressions de IMP par les types

```
type var = string

type aexpr =
  | Int of int
  | Var of var
  | Add of aexpr * aexpr
  | Sub of aexpr * aexpr
  | Mul of aexpr * aexpr

type bexpr =
  | Bool of bool
  | Eq   of aexpr * aexpr
  | Lt   of aexpr * aexpr
  | Not  of bexpr

type cexpr =
  | Skip
  | Set   of var * aexpr
  | Seq   of cexpr * cexpr
  | If    of bexpr * cexpr * cexpr
  | While of bexpr * cexpr
```

On implémente un état comme une fonction qui à toute variable associe sa valeur (qui est un entier). On peut donc décrire leur type, ainsi que la fonction qui modifie la valeur d'une variable dans un état par

```
type state = var -> int

let state_set s x n y = if y = x then n else s y
```

Enfin on implémente l'évaluation des différentes catégories syntaxiques par

```
let rec aeval s = function
  | Int n -> n
  | Var x -> s x
  | Add (a, a') -> aeval s a + aeval s a'
  | Sub (a, a') -> aeval s a - aeval s a'
  | Mul (a, a') -> aeval s a * aeval s a'

let rec beval s = function
  | Bool b -> b
  | Eq (a, a') -> aeval s a = aeval s a'
  | Lt (a, a') -> aeval s a < aeval s a'
  | Not b -> not (beval s b)

let rec eval s = function
  | Skip -> s
  | Set (x, a) -> state_set s x (aeval s a)
  | Seq (c, c') -> eval (eval s c) c'
  | If (b, c, c') -> if beval s b then eval s c else eval s c'
  | While (b, c) -> if beval s b then eval (eval s c) (While (b, c)) else s
```

B Typage en mini-ML

On considère ici la variante de mini-ML avec annotations de type. On peut décrire les types du langage mini-ML avec annotations de type par

```
type typ =
  | TInt | TBool
  | TArr of typ * typ
  | TProd of typ * typ
```

et les termes par

```
type var = string

type prog =
  | Int of int
  | Bool of bool
  | Var of string
  | Add
  | Fun of var * typ * prog
  | App of prog * prog
  | Pair of prog * prog
  | Fst of prog
  | Snd of prog
  | Fix of prog
```

La fonction d'inférence suivante prend un environnement `env` correspondant à Γ (codé comme une liste de paires variable et type) et un programme et renvoie son type, ou lève l'exception `Not_typable` si le programme n'admet pas de type dans l'environnement :

```
exception Not_typable

let rec infer env = function
  | Int _ -> TInt
  | Bool _ -> TBool
  | Var x ->
    (
      try List.assoc x env
      with Not_found -> raise Not_typable
    )
  | Add -> TArr (TProd (TInt, TInt), TInt)
  | Fun (x, a, t) -> infer ((x,a)::env) t
  | App (t, u) ->
    (
      match infer env t with
      | TArr (a, b) ->
        if infer env u <> a then raise Not_typable;
        b
      | _ -> raise Not_typable
    )
  | Pair (t, u) ->
    let a = infer env t in
    let b = infer env u in
    TProd (a, b)
  | Fst t ->
    (
      match infer env t with
      | TProd (a, b) -> a
```

```
      | _ -> raise Not_typable
    )
  | Snd t ->
    (
      match infer env t with
      | TProd (a, b) -> b
      | _ -> raise Not_typable
    )
  | Fix t ->
    (
      match infer env t with
      | TArr (a, a') when a = a' -> a
      | _ -> raise Not_typable
    )
```

Enfin, comme indiqué ci-dessus, on peut vérifier si un programme admet un type donné à l'aide de la fonction suivante, qui prend un environnement, un programme et un type et renvoie un booléen :

```
let check env t a =
  try infer env t = a
  with Not_typable -> false
```

Pour l'évaluation, on n'implémente généralement pas directement les règles de la figure 10, car la substitution est coûteuse à implémenter du fait de la nécessité de gérer le renommage des variables (voir section 2.3.2). En pratique, on utilise un *environnement* Σ qui est une liste de paires, notées $x \mapsto v$, consistant en une variable x et une valeur v, qui stocke l'information que la valeur de la variable x est v. On note $(x \mapsto v) \in \Sigma$ pour indiquer que la dernière valeur assignée à x est v, c'est-à-dire que le couple de la forme $x \mapsto v'$ le plus à droite dans Σ est tel que $v' = v$. Les règles de l'évaluation sont alors celles données à la figure 10. La fonction d'évaluation correspondante est la suivante (l'environnement est stocké dans la liste `env`) :

```
let rec eval env = function
  | Int _ | Bool _ | Add | Fun _ as v -> v
  | Var x -> List.assoc x env
  | App (t, u) ->
    (
      match eval env t with
      | Fun (x, _, t) ->
        let u = eval env u in
        eval ((x,u)::env) t
      | Add ->
        (
          match eval env u with
          | Pair (Int m, Int n) -> Int (m + n)
          | _ -> assert false
        )
      | _ -> assert false
    )
  | Pair (t, u) ->
    let t = eval env t in
    let u = eval env u in
    Pair (t, u)
  | Fst t ->
    (
      match eval env t with
      | Pair (t, u) -> t
      | _ -> assert false
    )
```

$$\frac{t \in \{\underline{n}, \underline{b}, \texttt{add}, \texttt{fst}, \texttt{snd}, \texttt{fun}\ x\ \to\ t'\}}{\Sigma \vdash t \longrightarrow\!\!\!\!\!\rightarrow t}\ (\textsc{val})$$

$$\frac{(x \mapsto v) \in \Sigma}{\Sigma \vdash x \longrightarrow\!\!\!\!\!\rightarrow v}\ (\textsc{var})$$

$$\frac{\Sigma \vdash t \longrightarrow\!\!\!\!\!\rightarrow (\texttt{fun}\ x\ \to\ t') \qquad \Sigma \vdash u \longrightarrow\!\!\!\!\!\rightarrow u' \qquad \Sigma, x \mapsto u' \vdash t' \longrightarrow\!\!\!\!\!\rightarrow v}{\Sigma \vdash t\ u \longrightarrow\!\!\!\!\!\rightarrow v}\ (\textsc{app})$$

$$\frac{\Sigma \vdash t \longrightarrow\!\!\!\!\!\rightarrow \texttt{add} \qquad \Sigma \vdash u \longrightarrow\!\!\!\!\!\rightarrow (\underline{m}, \underline{n})}{\Sigma \vdash t\ u \longrightarrow\!\!\!\!\!\rightarrow \underline{m+n}}\ (\textsc{add})$$

$$\frac{\Sigma \vdash t \longrightarrow\!\!\!\!\!\rightarrow \texttt{fst} \qquad \Sigma \vdash u \longrightarrow\!\!\!\!\!\rightarrow (v_1, v_2)}{\Sigma \vdash t\ u \longrightarrow\!\!\!\!\!\rightarrow v_1}\ (\textsc{fst})$$

$$\frac{\Sigma \vdash t \longrightarrow\!\!\!\!\!\rightarrow \texttt{fst} \qquad \Sigma \vdash u \longrightarrow\!\!\!\!\!\rightarrow (v_1, v_2)}{\Sigma \vdash t\ u \longrightarrow\!\!\!\!\!\rightarrow v_2}\ (\textsc{snd})$$

$$\frac{\Sigma \vdash t \longrightarrow\!\!\!\!\!\rightarrow t' \qquad \Sigma, x \mapsto t' \vdash u \longrightarrow\!\!\!\!\!\rightarrow v}{\Sigma \vdash (\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u) \longrightarrow\!\!\!\!\!\rightarrow v}\ (\textsc{let})$$

$$\frac{\Sigma \vdash t \longrightarrow\!\!\!\!\!\rightarrow t' \qquad \Sigma \vdash u \longrightarrow\!\!\!\!\!\rightarrow u'}{\Sigma \vdash (t, u) \longrightarrow\!\!\!\!\!\rightarrow (t', u')}\ (\textsc{pair})$$

FIGURE 10 – Évaluation par marchine abstraite en mini-ML.

```
| Snd t ->
  (
    match eval env t with
    | Pair (t, u) -> u
    | _ -> assert false
  )
| Fix t ->
  (
    match eval env t with
    | Fun (x, _, u) as t -> eval ((x,t)::env) u
    | _ -> assert false
  )
```

Index

α-convertibilité, 24

application, 16

branchement conditionnel, 4, 28

commande, 3
congruence, 25
contexte, 17

déclaration, 28
dérivation, 17
domaine, 17
déterminisme, 7, 15, 23

effet de bord, 4
environnement, 44
environnement de typage, 17
équation de types, 35
équivalence observationnelle, 9
état, 5, 42
evaluation, 44
évaluation, 3, 5, 21, 42
expression, 16, 42
    arithmétique, 2, 3
    booléenne, 3

fonction, 16
fonctionnelle, 32
forme normale, 26

IMP, 3
induction, 2
    sur les dérivations, 7
inférence de type, 43

jugement de typage, 17

mini-ML, 16

paire, 16
point fixe, 32
principal, 35
programme, 16
progrès, 30
projection, 16
pureté, 16

renommage, 24, 34
réduction, 9, 25
    stratégie, 26
réduction du sujet, 30

stratégie de réduction, 26
substitution, 24, 34
système d'équations, 35
sémantique
    à grands pas, 5, 21
    à petits pas, 9, 25
sûreté, 31

terme, 16, 43
typage, 17, 43
    dérivation, 17
    environnement, 17
    jugement, 17
    unicité, 19
type, 17, 43
    principal, 35

unificateur, 35

valeur, 21
variable, 3, 34
    fraîche, 36