Chapitre 7

Les références et les affectations

Considérons deux nombres : le nombre π et la température qu'il fait à Paris. Aujourd'hui, le nombre π est compris entre `3.14` et `3.15` et la température qu'il fait à Paris est comprise entre `16` et `17` degrés. Demain, le nombre π aura conservé sa valeur, mais la température à Paris aura sans doute changé. En mathématiques, on réserve le nom de *nombre* aux nombres qui ne varient pas au cours du temps : la température à Paris n'est pas un nombre qui change de valeur, mais une fonction du temps.

Cependant, la formalisation de la température d'un système comme une fonction du temps est peut-être trop générale, car elle ne prend pas en compte le fait que la variation de la température d'un système à un certain instant dépend, en général, de sa température à cet instant, mais pas de sa température dix secondes plus tôt ou dix secondes plus tard. Un système n'a donc, en général, pas accès à sa fonction température dans son intégralité, mais uniquement à la valeur actuelle de cette fonction. C'est pour cela que les équations de la physique sont, en général, des équations aux dérivées partielles et non des équations fonctionnelles quelconques.

Les programmes informatiques utilisent également des objets qui varient au cours du temps. Par exemple, dans un programme qui gère la billetterie d'une salle de concert, le nombre de places disponibles pour un concert varie au cours du temps : il diminue d'une unité à chaque fois qu'une place est vendue. Du point de vue mathématique, c'est donc une fonction du temps. Cependant pour savoir s'il est encore possible de vendre une place ou si la réservation est close, le programme a uniquement besoin d'avoir accès à la valeur actuelle de cette fonction, et non à cette fonction dans son intégralité : à un instant `t`, il n'interroge cette fonction que sur sa valeur en `t`.

De ce fait, quand on écrit un tel programme, on ne souhaite pas exprimer le nombre de places encore disponibles pour un concert par une fonction du temps, c'est-à-dire par un terme de type `nat -> nat` — en supposant une hor-

loge discrète —, ce qui signifierait que l'on veut connaître à chaque instant `t` le nombre de places encore disponibles pour ce concert à chaque instant `t'`. Cela n'est d'ailleurs pas possible, car cela supposerait que l'on connaisse le nombre de places disponibles à chaque instant `t'` du futur. On ne peut pas non plus exprimer ce nombre par un terme de type `nat` car, comme un nombre mathématique, la valeur d'un terme de type `nat` en PCF ne peut pas varier au cours du temps. On doit donc introduire une nouvelle sorte de termes pour ces nombres qui varient au cours du temps : les *références*, aussi parfois appelées *variables*, bien qu'il vaille mieux éviter cette terminologie, car la notion de référence est très différente de la notion de variable du langage mathématique et des langages fonctionnels.

Si `x` est une référence, on peut faire deux choses avec `x`, l'interroger sur la valeur actuelle `!x` et modifier sa valeur `x := t`, c'est-à-dire contribuer à la construction de cette fonction du temps, en décrétant que la valeur de cette fonction est désormais, et jusqu'à nouvelle modification, la valeur actuelle du terme `t`.

La question de l'égalité de deux « nombres qui varient au cours du temps » est délicate. On peut comparer un nombre qui varie au cours du temps, comme la température qu'il fait à Paris, à une feuille sur un arbre, qui petite, souple et verte au printemps devient grande, cassante et jaune à l'automne. Bien que la feuille se transforme, on ferait une erreur si on pensait que la petite feuille verte a été désintégrée et que la grande feuille jaune est apparue *ex nihilo*. Bien qu'elle se transforme, c'est bien la même feuille qui est sur l'arbre en mars et en octobre. C'est un vieux paradoxe qu'une chose puisse changer tout en restant la même chose. De même, la température à Paris est une seule et même chose, même si elle varie au cours du temps. En revanche, la température à Paris et la température à Rome sont deux choses différentes, même s'il peut arriver qu'elles soient accidentellement égales à certains moments.

Considérer la température à Paris et à Rome comme des fonctions du temps permet de résoudre ce paradoxe : le fait qu'une fonction prenne des valeurs différentes en deux points ne l'empêche pas d'être une seule et même fonction, et le fait que deux fonctions prennent accidentellement la même valeur en un point ne les empêche pas d'être distinctes.

Dans les langages de programmation, si `x` et `y` sont deux références, il faut soigneusement distinguer la question de savoir si les références `x` et `y` sont égales, c'est-à-dire si `x` et `y` sont une même chose — en termes mathématiques : la même fonction du temps — et la question de savoir si les nombres `!x` et `!y` sont égaux, c'est-à-dire si `x` et `y` ont accidentellement la même valeur à cet instant précis. En particulier, dans le premier cas, la valeur de `y` sera modifiée si on modifie la valeur de `x`, mais, dans le second, elle ne le sera pas nécessairement.

7.1 Une extension de PCF

Nous allons construire une extension de PCF en ajoutant les constructions `!` et `:=`.

Le terme `x := 4`, comme le terme `fact 3`, que nous connaissons, ordonne une action. Le terme `x := 4` ordonne de modifier la valeur de la référence `x`, de même que le terme `fact 3` ordonne de calculer la factorielle du nombre `3`. Il y a cependant une différence entre ces deux actions : l'effet du calcul de la factorielle de `3` est une valeur, l'effet de l'action `x := 4` est de modifier « l'état global du monde ». Avant cette action, la référence `x` avait la valeur `0`, par exemple, après cette action elle a la valeur `4`. Quand on ajoute des références à PCF, l'interprétation d'un terme ne produit plus uniquement une valeur, mais également un nouvel état du monde. Cette modification de l'état du monde est appelée l'*effet secondaire* de l'interprétation d'un terme — on emploie aussi la terminologie *effet de bord*, qui est une traduction imprécise de l'anglais *side effect*, qui signifie *effet secondaire*.

Dans la sémantique de PCF avec des références, l'état du monde à un instant donné se formalise comme une fonction d'un ensemble fini `R` vers les valeurs de PCF. Les éléments de `R` sont appelés des *références*. Dans le langage de programmation primitif d'un ordinateur, son *langage machine*, l'ensemble des références est fixe et c'est l'ensemble des adresses mémoires de l'ordinateur. Dans tous les autres langages, l'ensemble `R` est un ensemble quelconque. On ne distingue pas, dans la sémantique du langage, un ensemble `R` d'un autre ensemble `R'` qui a le même nombre d'éléments. Cela traduit le fait que les programmeurs ne savent pas précisément où sont stockées leurs données dans l'ordinateur.

En PCF avec des références, comme dans la plupart des langages de programmation, les valeurs associées aux références peuvent, bien entendu, varier au cours du temps, mais, de plus, l'ensemble `R` lui-même peut également varier au cours du temps : au cours de l'exécution d'un programme, on peut ajouter une nouvelle référence. Pour cela, le langage contient une construction `ref`. L'effet secondaire de l'interprétation du terme `ref t`, est de créer une nouvelle référence qui débute avec une valeur qui est la valeur actuelle du terme `t`. La valeur calculée par cette interprétation est la référence elle-même.

Puisque l'interprétation du terme `ref t` produit une valeur qui est une référence, les références doivent être des valeurs.

7.2 La sémantique de PCF avec des références

Dans la sémantique opérationnelle à grands pas de PCF avec des références, la relation est de la forme `e, m ⊢ t ↪ V, m'` où `t` est le terme interprété, `e` l'environnement dans lequel on l'interprète, `m` l'état du monde dans lequel cette interprétation se déroule, `V` la valeur produite par cette interprétation, et `m'` le nouvel état du monde produit par cette interprétation.

$$\frac{}{\texttt{e, m} \vdash \texttt{x} \hookrightarrow \texttt{V, m}} \text{ si e contient } \texttt{x = V}$$

$$\frac{\texttt{e', m} \vdash \texttt{fix y t} \hookrightarrow \texttt{V, m'}}{\texttt{e, m} \vdash \texttt{x} \hookrightarrow \texttt{V, m'}} \quad \text{si e contient } \texttt{x} = \langle \texttt{fix y t, e'} \rangle$$

$$\frac{\texttt{e, m} \vdash \texttt{u} \hookrightarrow \texttt{W, m'} \quad \texttt{e, m'} \vdash \texttt{t} \hookrightarrow \langle \texttt{x, t', e'} \rangle, \texttt{m''} \quad \texttt{(e', x = W), m''} \vdash \texttt{t'} \hookrightarrow \texttt{V, m'''}}{\texttt{e, m} \vdash \texttt{t u} \hookrightarrow \texttt{V, m'''}}$$

$$\frac{}{\texttt{e, m} \vdash \texttt{fun x -> t} \hookrightarrow \langle \texttt{x, t, e} \rangle, \texttt{m}}$$

$$\frac{}{\texttt{e, m} \vdash \texttt{n} \hookrightarrow \texttt{n, m}}$$

$$\frac{\texttt{e, m} \vdash \texttt{u} \hookrightarrow \texttt{q, m'} \quad \texttt{e, m'} \vdash \texttt{t} \hookrightarrow \texttt{p, m''}}{\texttt{e, m} \vdash \texttt{t} \otimes \texttt{u} \hookrightarrow \texttt{n, m''}} \text{ si } \texttt{p} \otimes \texttt{q} = \texttt{n}$$

$$\frac{\texttt{e, m} \vdash \texttt{t} \hookrightarrow \texttt{0, m'} \quad \texttt{e, m'} \vdash \texttt{u} \hookrightarrow \texttt{V, m''}}{\texttt{e, m} \vdash \texttt{ifz t then u else v} \hookrightarrow \texttt{V, m''}}$$

$$\frac{\texttt{e, m} \vdash \texttt{t} \hookrightarrow \texttt{n, m'} \quad \texttt{e, m'} \vdash \texttt{v} \hookrightarrow \texttt{V, m''}}{\texttt{e, m} \vdash \texttt{ifz t then u else v} \hookrightarrow \texttt{V, m''}} \quad \begin{array}{c}\text{si n constante} \\ \text{entière} \neq 0\end{array}$$

$$\frac{\texttt{(e, x =} \langle \texttt{fix x t, e} \rangle \texttt{), m} \vdash \texttt{t} \hookrightarrow \texttt{V, m'}}{\texttt{e, m} \vdash \texttt{fix x t} \hookrightarrow \texttt{V, m'}}$$

$$\frac{\texttt{e, m} \vdash \texttt{t} \hookrightarrow \texttt{W, m'} \quad \texttt{(e, x = W), m'} \vdash \texttt{u} \hookrightarrow \texttt{V, m''}}{\texttt{e, m} \vdash \texttt{let x = t in u} \hookrightarrow \texttt{V, m''}}$$

On peut alors exprimer les règles des trois nouvelles constructions, `ref`, `!` et `:=`

$$\frac{\texttt{e, m} \vdash \texttt{t} \hookrightarrow \texttt{V, m'}}{\texttt{e, m} \vdash \texttt{ref t} \hookrightarrow \texttt{r, (m', r = V)}}$$

si `r` est une référence quelconque qui n'apparaît pas dans `m'`

$$\frac{\texttt{e, m} \vdash \texttt{t} \hookrightarrow \texttt{r, m'}}{\texttt{e, m} \vdash \texttt{!t} \hookrightarrow \texttt{V, m'}} \text{ si m' contient } \texttt{r = V}$$

$$\frac{\texttt{e, m} \vdash \texttt{t} \hookrightarrow \texttt{r, m'} \quad \texttt{e, m'} \vdash \texttt{u} \hookrightarrow \texttt{V, m''}}{\texttt{e, m} \vdash \texttt{t := u} \hookrightarrow \texttt{0, (m'', r = V)}}$$

La construction `t ; u` dont l'interprétation consiste à interpréter `t`, jeter la valeur obtenue, puis interpréter `u`, n'avait pas d'intérêt en l'absence d'effets secondaires, puisque la valeur du terme `t ; u` était toujours la même que celle du terme `u`, pourvu que `t` termine. On peut maintenant l'ajouter à PCF

$$\frac{\texttt{e, m} \vdash \texttt{t} \hookrightarrow \texttt{V, m'} \quad \texttt{e, m'} \vdash \texttt{u} \hookrightarrow \texttt{W, m''}}{\texttt{e, m} \vdash \texttt{t ; u} \hookrightarrow \texttt{W, m''}}$$

On peut également ajouter les constructions `whilez`, `for`, ... qui n'avaient pas non plus d'intérêt en l'absence d'effets secondaires.

Exercice 7.1 *Écrire un interpréteur pour PCF avec des références.*

L'incertitude dans laquelle l'introduction de ce livre nous avait plongés peut enfin être dissipée.

Exercice 7.2 *Quelle est la valeur du terme*

```
let n = ref 0
in let f = fun x -> fun y -> x
in let g = fun z -> (n := !n + z; !n)
in f (g 2) (g 7)
```

? Dans quel ordre sont interprétés les arguments en PCF ? Pourquoi ? Comment pourrait-on modifier les règles ci-dessus afin que la valeur de ce terme soit `2` *et non* `9` *? À la section 2.5 nous avons fait une remarque : « Dans le cas d'une application... ». Que penser de cette remarque ?*

Quelle est la valeur de ce terme en Caml ? En Java, quelle est la valeur du terme

```
class Reference {
 static int n;
 static int f (int x, int y) {return x;}
 static int g (int z) {n = n + z; return n;}
 static public void main (String[ ] args) {
  n = 0; System.out.println(f(g(2),g(7)));}}
```

? Dans quel ordre Caml interprète-t-il ses arguments ? et Java ?

Exercice 7.3 *La valeur du terme*

```
let x = ref 4 in let f = fun y -> y + !x in (x := 5; f 6)
```

est-elle `10` *ou bien* `11` *? Comparer ce résultat avec celui de l'exercice 2.8.*

Exercice 7.4 *Donner la sémantique opérationnelle à grands pas de la construction* `whilez`*. Quelle est la valeur du terme*

```
let f = fun n ->
(let k = ref 1
 in let i = ref 1
 in (whilez (!i - n) do k := !k * !i; i := !i + 1 done; !k))
in f 3
```

?

Exercice 7.5 *(Les bizarreries des références en appel par nom) Les règles de sémantique opérationnelle à grands pas que nous avons données sont-elles en appel par nom ou en appel par valeur ? Donner une règle similaire pour l'application en appel par nom, en gardant le* `let` *en appel par valeur. Quelle est la valeur du terme* `let n = ref 0 in ((fun x -> x + x) (n := !n + 1 ; 4)) ; !n` *en appel par valeur ? et en appel par nom ? Quelle est la valeur du terme* `let n = ref 0 in ((fun x -> 2 * x) (n := !n + 1 ; 4)) ; !n` *en appel par valeur ? et en appel par nom ?*

Exercice 7.6 *(Le typage des références) Pour typer les termes écrits en PCF avec des références, il faut ajouter un symbole* `ref` *au langage des types, de manière à ce que* `nat ref`*, par exemple, soit le type des références sur un entier. Ainsi, si* `t` *est un terme de type* `A ref` *alors* `!t` *est un terme de type* `A`*.*

Étendre les règles de type de la section 5.1 afin de typer les termes de PCF avec des références.

Écrire un vérificateur de types pour PCF avec des références.

Associer références et polymorphisme dans le même langage est délicat. On ne tentera pas de le faire dans cet exercice.

Exercice 7.7 *(La fonctionnalisation des programmes impératifs) On considère un terme* `t` *qui exprime une fonction entière à* `p` *arguments entiers et qui contient une variable libre* `n` *de type* `nat ref`*. On associe à ce terme une fonction qui prend en arguments* `p + 1` *entiers et qui retourne un couple d'entiers — voir l'exercice 3.13 — qui à* $\mathtt{a}_1, \ldots, \mathtt{a}_p$, `m` *associe le couple d'entiers formé de la valeur du terme* `let n = ref m in (t` $\mathtt{a}_1$ `...` $\mathtt{a}_p$`)` *et de la valeur du terme* `!n` *à la fin de cette interprétation. Quelle est la fonction associée au terme*

- `fun z -> (n := !n + z ; !n)`

? et au terme

- `(fun z -> (n := !n + z ; !n)) 7` *?*

? et au terme

- `(fun x -> fun y -> x) ((fun z -> (n := !n + z ; !n)) 2) ((fun z -> (n := !n + z ; !n)) 7)` *?*

Peut-on programmer ces fonctions en PCF sans références ?

Plus généralement,

- *quelle est la fonction associée au terme* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `2` *?*
- *Et au terme* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` $\mathtt{y}_1$ *?*
- *Et au terme* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `!n` *?*
- *Si* `t` *est de type* `nat` *et* `f` *est la fonction associée au terme* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `t`*, quelle est la fonction associée à* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `n := t` *?*
- *Si* `t` *et* `u` *sont de type* `nat` *et* `f` *et* `g` *les fonctions associées aux termes* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `t` *et* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `u`*, quelle est la fonction associée à* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `(t + u)` *?*
- *Si* `t` *et* `u` *sont de type* `nat` *et* `f` *et* `g` *les fonctions associées aux termes* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `t` *et* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `u`*, quelle est la fonction associée à* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `(t ; u)` *?*
- *Si* `t` *est un terme de type* `nat -> ... -> nat -> nat` *— avec* `q` *arguments de type* `nat` *—* $\mathtt{u}_1$*, ...,* $\mathtt{u}_q$ *sont des termes de type* `nat`*, et* `f`*,* $\mathtt{g}_1$*, ...,* $\mathtt{g}_q$ *les fonctions associées aux termes* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `t` *et* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` $\mathtt{u}_1$*, ...,* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` $\mathtt{u}_q$*, quelle est la fonction associée à* `fun` $\mathtt{y}_1$ `->` ... `->` `fun` $\mathtt{y}_p$ `->` `(t` $\mathtt{u}_1$ ... $\mathtt{u}_q$`)` *?*

Peut-on programmer ces fonctions en PCF sans références ?

Écrire un programme qui transforme un terme de PCF construit avec ces symboles et qui contient une variable libre de type `nat ref` *en un programme qui n'en contient pas et qui a la même sémantique.*

Exercice 7.8 *(Pour ceux qui préfèrent écrire* `x := x + 1` *que* `x :=!x + 1`*) On se donne un ensemble fini de références et on définit une extension de PCF avec une constante pour chacune de ces références. Ces constantes sont appelées* variables affectables. *Le symbole* `:=` *s'applique désormais à une variable affectable et un terme :* `X := t`.

Si `X` *est une variable affectable, la valeur attribuée par la sémantique opérationnelle au terme* `X` *est la valeur associée à la référence* `X` *dans l'état du monde dans lequel cette interprétation se déroule.*

Écrire la sémantique opérationnelle à grands pas de cette extension de PCF.

Écrire un interpréteur pour cette extension de PCF.

Exercice 7.9 *(Un langage impératif minimal) On considère un langage qui comprend des constantes entières, les quatre opérations, des variables affectables — voir exercice 7.8 —, l'affectation* `:=`*, la séquence* `;`*, le test* `ifz` *et la boucle* `whilez`*, mais pas de variables ordinaires, de* `fun`*, de* `fix`*, de* `let` *ou d'application.*

Écrire la sémantique opérationnelle de ce langage. Écrire un interpréteur pour ce langage. Écrire la factorielle dans ce langage. Que peut-on programmer dans ce langage ?

Pour conclure ce chapitre, on peut remarquer que dans quasiment tous les langages de programmation, on a deux manières assez différentes de programmer la factorielle. Par exemple, en Java, on peut programmer la factorielle récursivement

```
static int fact (int x) {
 if (x == 0) return 1; return x * (fact (x - 1));}
```

ou avec une boucle

```
static int fact (int x) {
 int k = 1;
 for (int i = 1; i <= x; i = i + 1) k = k * i;
 return k;}
```

Doit-on préférer un programme ou l'autre ?

Bien entendu, la théorie des langages de programmation ne permet pas de répondre aux questions « morales » de la forme « Doit-on... ? » On peut cependant dire quelques mots sur la manière dont cette question a évolué au cours du temps.

Dans les premiers langages de programmation — langages machines, assembleurs, Fortran, Basic, ... — seul le second programme est possible. Il est, en effet, plus facile d'exécuter un programme avec des boucles et des références sur un ordinateur qui, *in fine*, est lui-même un système physique, avec un état qui

évolue dans le temps selon des règles locales, que d'exécuter une fonction définie par un point fixe.

Lisp est un des premiers langages à avoir proposé des définitions récursives. Avec Lisp, on a pu se passer, pour la première fois, de références et d'effets secondaires, ce qui simplifie la sémantique du langage, le rapproche du langage mathématique, permet de raisonner plus facilement sur les programmes et permet d'écrire des programmes complexes plus simplement. Par exemple, il est plus facile d'écrire un programme qui dérive une expression algébrique en utilisant la récursivité que de gérer explicitement une pile d'expressions en attente d'être dérivées. Il était alors naturel d'opposer le style fonctionnel pur au style impératif « impur ».

Restait que les premières implémentations des langages fonctionnels étaient beaucoup plus lentes que celles des langages impératifs, précisément parce que, comme nous l'avons dit, exécuter un programme fonctionnel sur un ordinateur, qui est un système physique, est plus compliqué qu'exécuter un programme impératif. Au cours des années quatre vingt dix, la compilation des langages fonctionnels a fait des progrès qui ont beaucoup affaibli la pertinence des arguments d'efficacité, sauf dans le domaine du calcul intensif.

De plus, tous les langages modernes proposent à la fois des traits fonctionnels et des traits impératifs, ce qui fait que le seul critère de choix doit aujourd'hui être la simplicité de la programmation.

Il est raisonnable de penser que, du point de vue de la simplicité de la programmation, tous les problèmes ne sont pas identiques. Un programme qui dérive des expressions algébriques s'exprime plus simplement dans un style fonctionnel. En revanche, quand on programme une tortue Logo, il est assez naturel de parler de la position de la tortue sur la table, de son orientation, ... — c'est-à-dire de son état à un instant donné. Il est également naturel de parler des actions de la tortue : se déplacer, tracer un trait, ... et il n'est pas si pratique de parler de tout cela de manière fonctionnelle : il répugne en effet aux tortues de concevoir leurs actions comme des fonctions de l'espace des dessins dans lui-même.

Un point reste encore aujourd'hui mystérieux : les programmes, qu'ils soient fonctionnels ou impératifs, sont toujours des définitions de fonctions des valeurs d'entrées vers les valeurs de sorties. Si la programmation impérative a amené de nouvelles manières de définir des fonctions, qui sont dans certains cas plus pratiques pour les informaticiens que les définitions mathématiques traditionnelles des langages fonctionnels, on ne voit pas pourquoi elles ne seraient pas également plus pratiques pour les mathématiciens. Pourtant, il ne semble pas que le langage mathématique utilise de notion analogue à celle de référence.