Chapitre 2

Le langage PCF

Nous allons illustrer ces différentes manières d'exprimer la sémantique d'un langage de programmation par un exemple : le langage PCF — *Programming language for computable functions* —, aussi appelé Mini-ML.

2.1 Un langage fonctionnel : PCF

2.1.1 Les programmes sont des fonctions

Nous avons remarqué au chapitre précédent qu'un programme déterministe calcule une fonction et nous en avons tiré les principes d'une sémantique : la sémantique dénotationnelle. Cette remarque sert aussi de base à certains langages de programmation : les langages fonctionnels comme Caml, Haskell ou Lisp, qui sont traditionnellement ceux par lesquels on commence l'étude des langages de programmation.

Dans ces langages, on cherche à réduire la distance qui sépare la notion informatique de programme de la notion mathématique de fonction. Autrement dit, on cherche à réduire la distance entre un programme et sa sémantique dénotationnelle. Les deux constructions de base du langage PCF sont : *la construction explicite* d'une fonction que l'on note `fun x -> t`, et l'*application* d'une fonction à son argument que l'on note `t u`.

Le langage PCF contient également une constante pour chaque entier naturel, les quatre opérations `+, -, *, /` et le test à zéro `ifz t then u else v`. Sur les entiers naturels, l'addition et la multiplication sont toujours définies, c'est également le cas de la soustraction, avec la convention `n - m = 0` quand `n < m`. La division est la division euclidienne, mais la division par `0` produit une erreur.

2.1.2 Les fonctions sont des objets de première classe

Dans beaucoup de langages de programmation, il est possible de définir une fonction qui prend une autre fonction en argument, ou qui retourne une autre

fonction, même si cette construction a une syntaxe parfois différente du passage d'un argument ordinaire, comme un entier ou une chaîne de caractères. Dans un langage fonctionnel, une fonction se définit de la même manière, qu'elle prenne un entier ou une fonction en argument.

Par exemple, la composition d'une fonction avec elle-même se définit ainsi `fun f -> fun x -> f (f x)`.

Pour exprimer le fait que les fonctions ne sont pas distinguées, et donc qu'elles peuvent être prises en argument et retournées par d'autres fonctions, on dit parfois que les fonctions sont *des objets de première classe*.

2.1.3 Les fonctions de plusieurs arguments

Il n'y a pas en PCF de symbole permettant de construire une fonction de plusieurs arguments. On construit ces fonctions comme des fonctions d'un argument unique en utilisant l'isomorphisme `(A × B) -> C = A -> (B -> C)`. Par exemple, la fonction qui à `x` et `y` associe le nombre `x * x + y * y` est définie comme la fonction qui à `x` associe la fonction qui à `y` associe le nombre `x * x + y * y` : `fun x -> fun y -> x * x + y * y`.

Appliquer cette fonction `f` aux nombres `3` et `4` demande de l'appliquer d'abord à `3`, ce qui donne le terme `f 3`, qui est la fonction qui à `y` associe `3 * 3 + y * y`, puis à `4`, ce qui donne le terme `(f 3) 4`. On convient que l'application associe à gauche et on note ce terme `f 3 4`.

2.1.4 L'absence d'affectations

Par rapport à un langage comme Caml ou Java, la principale originalité de PCF est l'absence d'*affectations*, il n'y a pas de construction de la forme `x := t` ou `x = t` permettant d'affecter une valeur à une « variable ». Nous verrons, au chapitre 7, une extension de PCF avec des affectations.

2.1.5 Les définitions récursives

En mathématiques, certaines fonctions ne peuvent pas être construites explicitement. Par exemple, dans un manuel pour le collège ou le lycée, on définit la fonction puissance en utilisant des points de suspension

$$\texttt{x, n} \mapsto \underbrace{\texttt{x} \times \cdots \times \texttt{x}}_{\texttt{n fois}}$$

ou encore par une définition par récurrence.

Dans les langages de programmation, on utilise des constructions similaires : les boucles et les définitions récursives. Le langage PCF contient une construction qui permet les définitions récursives.

On dit souvent qu'une fonction est définie récursivement si on peut l'utiliser elle-même dans sa propre définition. Ce principe est absurde : dans les langages de programmation, comme ailleurs, les définitions circulaires sont incorrectes.

On ne peut donc pas « définir » la fonction `fact` comme la fonction `fun n -> ifz n then 1 else n * (fact (n - 1))`. De manière générale, on ne peut pas définir une fonction `f` par un terme `G` dans lequel cette fonction `f` a une occurrence. En revanche, on peut définir la fonction `f` comme le point fixe de la fonction `fun f -> G`. Par exemple, on peut définir la fonction `fact` comme le point fixe de la fonction `fun f -> fun n -> ifz n then 1 else n * (f (n - 1))`.

Cette fonction a-t-elle un point fixe ? et si elle en a un, ce point fixe est-il unique ? et sinon de quel point fixe parle-t-on ? Laissons ces questions pour le moment, et souvenons-nous simplement qu'une définition récursive est la définition d'un point fixe.

Le langage PCF contient un symbole `fix` qui lie une variable dans son argument, tel que le terme `fix f G` exprime le point fixe de la fonction `fun f -> G`. La fonction `fact` se définit alors ainsi : `fix f fun n -> ifz n then 1 else n * (f (n - 1))`.

Ici encore, on peut remarquer que cette notation `fix` permet de construire la factorielle sans nécessairement lui donner un nom.

2.1.6 Les définitions

On peut, en théorie, se passer de définitions et remplacer partout les symboles définis par le corps de leur définition. Cependant, l'introduction d'un mécanisme de définition permet de rendre les programmes plus simples et plus clairs.

On ajoute donc une dernière construction à PCF, la construction `let x = t in u`. Les occurrences de la variable `x` dans `u` sont liées par cette construction, mais pas celles de `x` dans `t`. Le symbole `let` est donc un symbole à deux arguments et il lie une variable dans son second argument.

2.1.7 Le langage PCF

Le langage PCF contient
- un symbole `fun` à un argument, liant une variable dans son argument,
- un symbole α à deux arguments, ne liant pas de variables dans ses arguments,
- une infinité de constantes pour les entiers,
- quatre symboles `+`, `-`, `*` et `/` à deux arguments, ne liant pas de variables dans leurs arguments,
- un symbole `ifz` à trois arguments, ne liant pas de variables dans ses arguments,
- un symbole `fix` à un argument, liant une variable dans son argument,
- un symbole `let` à deux arguments, liant une variable dans son second argument.

Autrement dit, la syntaxe de PCF est inductivement définie par

```
t = x
  | fun x -> t
```

```
| t t
| n
| t + t | t - t | t * t | t / t
| ifz t then t else t
| fix x t
| let x = t in t
```

Malgré sa petite taille, le langage PCF est complet au sens de Turing, c'est-à-dire que toutes les fonctions calculables des entiers dans les entiers sont programmables en PCF.

Exercice 2.1 *Écrire un programme en PCF qui prend deux nombres entiers* `n` *et* `p` *en arguments et retourne* $\mathtt{n}^{\mathtt{p}}$.

Exercice 2.2 *Écrire un programme en PCF qui prend un nombre entier* `n` *en argument et retourne l'entier* `1` *ou* `0` *selon que ce nombre est premier ou non.*

Exercice 2.3 *(Les polynômes en PCF) Écrire un programme en PCF qui prend un nombre entier* `q` *en argument et retourne le plus grand entier* `u` *tel que* `u (u + 1) / 2` $\leq$ `q`.

La fonction `K` *de Cantor est la fonction de* $\mathbb{N}^2$ *dans* $\mathbb{N}$ `fun n -> fun p -> (n + p) (n + p + 1) / 2 + n`. *Et la fonction* `K'` *est la fonction de* $\mathbb{N}$ *dans* $\mathbb{N}^2$ `fun q -> (q - (u (u + 1) / 2), u - q + u (u + 1) / 2)` *où* `u` *est le plus grand entier tel que* `u (u + 1) / 2` $\leq$ `q`.

Montrer que `K` $\circ$ `K' = id`. *Soit* `n` *et* `p` *deux entiers, montrer que le plus grand entier* `u` *tel que* `u (u + 1) / 2` $\leq$ `(n + p) (n + p + 1) / 2 + n` *est* `n + p`. *En déduire que* `K'` $\circ$ `K = id`. *En déduire que* `K` *est une bijection de* $\mathbb{N}^2$ *dans* $\mathbb{N}$.

Soit `L` *la fonction* `fun n -> fun p -> (K n p) + 1`. *Un polynôme à coefficients entiers* $\mathtt{a_0} + \mathtt{a_1}\ \mathtt{X} + \cdots + \mathtt{a_i}\ \mathtt{X}^{\mathtt{i}} + \cdots + \mathtt{a_n}\ \mathtt{X}^{\mathtt{n}}$ *peut se représenter par l'entier* $\mathtt{L}\ \mathtt{a_0}\ (\mathtt{L}\ \mathtt{a_1}\ (\mathtt{L}\ \mathtt{a_2}\ \ldots\ (\mathtt{L}\ \mathtt{a_n}\ 0)\ldots))$.

Écrire en PCF un programme qui prend deux entiers en arguments et retourne la valeur du polynôme représenté par le premier entier appliqué au second.

2.2 La sémantique opérationnelle à petits pas de PCF

2.2.1 Les règles

Considérons le programme `fun x -> 2 * x` et appliquons-le à la constante `3`. Nous obtenons le terme `(fun x -> 2 * x) 3`. Conformément à l'esprit de la sémantique opérationnelle à petits pas, tentons de calculer peu à peu ce terme jusqu'à obtenir un résultat : `6` si tout se passe bien. La première étape de cette simplification est celle du *passage d'arguments*, le remplacement de l'argument

formel `x` par l'argument réel `3`. Le terme initial se transforme donc, lors d'un premier petit pas, en le terme `2 * 3`. Dans un second petit pas, le terme `2 * 3` se calcule en `6`. Le premier petit pas, le passage d'arguments, se généralise à tous les termes de la forme `(fun x -> t) u` où une fonction de la forme `fun x -> t` est appliquée à un argument `u`. Cela mène à la règle suivante appelée *règle de β-réduction*

$$\texttt{(fun x -> t) u} \longrightarrow \texttt{(u/x)t}$$

La relation $\texttt{t} \longrightarrow \texttt{u}$ se lit « `t` se réduit — ou se réécrit — en `u` ». La seconde règle se généralise en la règle

$$\texttt{p} \otimes \texttt{q} \longrightarrow \texttt{n} \text{ (si } \texttt{p} \otimes \texttt{q} = \texttt{n})$$

où $\otimes$ est l'un des quatre opérateurs arithmétiques. On ajoute des règles similaires pour la conditionnelle

$$\texttt{ifz 0 then t else u} \longrightarrow \texttt{t}$$
$$\texttt{ifz n then t else u} \longrightarrow \texttt{u} \text{ (si } \texttt{n} \text{ est une constante entière différente de } \texttt{0})$$

une règle pour le point fixe

$$\texttt{fix x t} \longrightarrow \texttt{(fix x t/x)t}$$

et une règle pour le `let`

$$\texttt{let x = t in u} \longrightarrow \texttt{(t/x)u}$$

On appelle *radical* un terme `t` qui peut être réduit par la relation $\longrightarrow$, c'est-à-dire un terme tel qu'il existe un terme `u` tel que $\texttt{t} \longrightarrow \texttt{u}$.

2.2.2 Les nombres

On peut objecter, à juste titre, que la règle $\texttt{p} \otimes \texttt{q} \longrightarrow \texttt{n}$ (si $\texttt{p} \otimes \texttt{q} = \texttt{n}$) dont une instance est $\texttt{2 * 3} \longrightarrow \texttt{6}$ n'explique pas réellement la sémantique des opérateurs arithmétiques, puisqu'elle ne fait que ramener la multiplication de PCF à celle des mathématiques. Ce choix est motivé par le fait que ce n'est pas tant la sémantique des opérateurs arithmétiques que nous voulons mettre en lumière que celle des autres constructions du langage.

Si on veut détailler la sémantique des opérateurs arithmétiques sans se reposer sur leurs homologues mathématiques, il faut considérer une variante de PCF dans laquelle on supprime les constantes entières et on introduit une constante pour le nombre `0` et un symbole `S` — pour « successeur » — à un argument. Le nombre `3`, par exemple, se représente désormais par le terme `S(S(S(0)))`. On ajoute alors les règles de sémantique opérationnelle à petits pas

$\texttt{0 + u} \longrightarrow \texttt{u}$
$\texttt{S(t) + u} \longrightarrow \texttt{S(t + u)}$
$\texttt{0 - u} \longrightarrow \texttt{0}$
$\texttt{t - 0} \longrightarrow \texttt{t}$
$\texttt{S(t) - S(u)} \longrightarrow \texttt{t - u}$

`0 * u` $\longrightarrow$ `0`
`S(t) * u` $\longrightarrow$ `t * u + u`
`t / S(u)` $\longrightarrow$ `ifz t - u then 0 else S((t - S(u)) / S(u))`

Remarquons que, en toute rigueur, nous devrions encore ajouter une règle pour la division par `0` qui produirait un terme exceptionnel : `erreur`.

Exercice 2.4 *(Les entiers de Church) Au lieu d'introduire des symboles* `0` *et* `S`*, on peut exprimer l'entier* `n`*, non par le terme* `S(S(...(0)...))`*, mais par le terme* `fun z -> fun s -> s (s (...(s z)...))`*. Montrer que l'on peut programmer l'addition et la multiplication sur ces entiers. Montrer que l'on peut programmer le test à* `0`*.*

Exercice 2.5 *(La numération à position) On peut objecter, à juste titre, que la représentation des entiers avec les symboles* `0` *et* `S` *ou comme des entiers de Church est une régression historique puisque l'on représente alors un entier par un terme dont la taille est linéaire en l'entier — comme quand on représente le nombre* `n` *par* `n` *bâtons — et non logarithmique comme avec la numération à position.*

Une alternative consiste à se donner un symbole `z` *pour le nombre* `0` *et deux fonctions* `0` *et* `I` *pour les fonctions* `n` $\mapsto$ `2 * n` *et* `n` $\mapsto$ `2 * n + 1`*. Le nombre* `26` *se représente alors par le terme* `0(I(0(I(I(z)))))`*, en retournant ce terme, on obtient* `II0I0` *qui est la représentation binaire de ce nombre.*

Écrire la sémantique opérationnelle à petits pas des opérateurs arithmétiques dans ce langage.

2.2.3 La congruence

En utilisant les règles de la sémantique opérationnelle à petits pas de PCF, on obtient que

`(fun x -> 2 * x) 3` $\longrightarrow$ `2 * 3` $\longrightarrow$ `6`

Si on note $\longrightarrow^*$ la fermeture réflexive-transitive de la relation $\longrightarrow$, on en déduit `(fun x -> 2 * x) 3` $\longrightarrow^*$ `6`.

Cependant, avec cette définition, le terme `(2 + 3) + 4` ne se réduit pas sur le terme `9` par la relation $\longrightarrow^*$. En effet, pour réduire un terme de la forme `t + u` il est nécessaire que les termes `t` et `u` soient des constantes entières. Or, le premier terme `2 + 3` n'est pas une constante entière mais une somme. Il faut donc, dans un premier temps, calculer `2 + 3` en `5`, puis, dans un second temps, calculer `5 + 4` en `9`. Le problème est que, avec notre définition, si le terme `2 + 3` se réduit sur `5`, le terme `(2 + 3) + 4`, en revanche, ne se réduit pas sur `5 + 4`.

Il est donc nécessaire de définir une autre relation dans laquelle on peut appliquer une règle à n'importe quel sous-terme du terme que l'on cherche à réduire. On définit donc inductivement une relation $\triangleright$ ainsi

$$\frac{}{\texttt{t} \triangleright \texttt{u}} \text{ si } \texttt{t} \longrightarrow \texttt{u}$$

$$\frac{t \rhd u}{t\ v \rhd u\ v}$$

$$\frac{t \rhd u}{v\ t \rhd v\ u}$$

$$\frac{t \rhd u}{\texttt{fun x -> t} \rhd \texttt{fun x -> u}}$$

$$\frac{t \rhd u}{t + v \rhd u + v}$$

...

On démontre qu'un terme est un radical pour la relation $\rhd$ si et seulement si l'un de ses sous-termes est un radical pour la relation $\longrightarrow$.

2.2.4 Un exemple

Pour illustrer les règles de la sémantique opérationnelle à petits pas de PCF, calculons la factorielle du nombre 3.

```
(fix f fun n -> ifz n then 1 else n * (f (n - 1))) 3
▷ (fun n -> ifz n then 1 else n * ((fix f fun n -> ifz n then 1
else n * (f (n - 1))) (n - 1))) 3
▷ ifz 3 then 1 else 3 * ((fix f fun n -> ifz n then 1 else n
* (f (n - 1))) (3 - 1))
▷ 3 * ((fix f fun n -> ifz n then 1 else n * (f (n - 1)))
(3 - 1))
▷ 3 * ((fix f fun n -> ifz n then 1 else n * (f (n - 1))) 2)
▷ 3 * ((fun n -> ifz n then 1 else n * ((fix f fun n -> ifz n
then 1 else n * (f (n - 1))) (n - 1))) 2)
▷ 3 * (ifz 2 then 1 else 2 * ((fix f fun n -> ifz n then 1
else n * (f (n - 1))) (2 - 1)))
▷ 3 * (2 * ((fix f fun n -> ifz n then 1 else n * (f (n - 1)))
(2 - 1)))
▷ 3 * (2 * ((fix f fun n -> ifz n then 1 else n * (f (n - 1))) 1))
▷ 3 * (2 * ((fun n -> ifz n then 1 else n * ((fix f fun n ->
ifz n then 1 else n * (f (n - 1))) (n - 1))) 1))
▷ 3 * (2 * (ifz 1 then 1 else 1 * ((fix f fun n -> ifz n then 1
else n * (f (n - 1))) (1 - 1))))
▷ 3 * (2 * (1 * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) (1 - 1))))
▷ 3 * (2 * (1 * ((fix f fun n -> ifz n then 1
else n * (f (n - 1))) 0)))
▷ 3 * (2 * (1 * ((fun n -> ifz n then 1 else n * ((fix f fun n ->
ifz n then 1 else n * (f (n - 1))) (n - 1))) 0)))
▷ 3 * (2 * (1 * ((ifz 0 then 1 else 0 * ((fix f fun n ->
ifz n then 1 else n * (f (n - 1))) (0 - 1))))))
▷ 3 * (2 * (1 * 1)) ▷ 3 * (2 * 1) ▷ 3 * 2 ▷ 6
```

2.2.5 Les termes irréductibles et clos

Un terme `t` est dit *irréductible* s'il ne peut pas être réduit par la relation $\triangleright$, c'est-à-dire s'il n'existe pas de terme `u` tel que `t` $\triangleright$ `u`.

On peut alors définir la relation « le terme `u` est le résultat du calcul du terme `t` », où `t` est un terme clos par : `t` $\hookrightarrow$ `u` si et seulement si `t` $\triangleright^*$ `u` et `u` est irréductible. Dans ce cas, le terme `u` est nécessairement clos. Enfin, la relation « le programme `p` sur les entrées e_1, ..., e_n donne la sortie `s` » est simplement la relation `p` e_1 `...` $e_n \hookrightarrow$ `s`.

Exercice 2.6 *(La classification des termes irréductibles et clos) Montrer qu'un terme est irréductible et clos si et seulement s'il est de l'une des formes suivantes*

- `fun x -> t` *où* `t` *est irréductible et ne contient pas d'autres variables libres que* `x`,
- `n` *où* `n` *est une constante entière,*
- V_1 V_2, *où* V_1 *et* V_2 *sont des termes irréductibles et clos et* V_1 *n'est pas de la forme* `fun x -> t`,
- $V_1 \otimes V_2$, *où* V_1 *et* V_2 *sont des termes irréductibles et clos qui ne sont pas tous les deux des constantes entières,*
- `ifz` V_1 `then` V_2 `else` V_3 *où* V_1, V_2 *et* V_3 *sont des termes irréductibles et clos et* V_1 *n'est pas une constante entière.*

Les constantes entières et les termes irréductibles et clos de la forme `fun x -> t` sont appelés des *valeurs*. Quand le résultat du calcul d'un terme est une valeur, on dit aussi que c'est la valeur de ce terme. Calculer la valeur d'un terme s'appelle l'*évaluer*.

Les valeurs ne sont malheureusement pas les seuls résultats possibles pour le calcul d'un terme. Par exemple, le terme `(fun x -> x) 1 2` se réduit sur le terme `1 2` qui est irréductible et clos, et qui est donc le résultat du calcul du terme `(fun x -> x) 1 2`. Ce résultat est indésirable car il n'y a pas grand sens à appliquer un objet, comme `1`, qui n'est pas une fonction. Un tel terme irréductible et clos qui n'est pas une valeur s'appelle un terme *bloqué*. Les termes bloqués sont de la forme V_1 V_2, où V_1 et V_2 sont des termes irréductibles et clos et V_1 n'est pas de la forme `fun x -> t`, par exemple `1 2`, $V_1 \otimes V_2$, où V_1 et V_2 sont des termes irréductibles et clos qui ne sont pas tous les deux des constantes entières, par exemple `1 + (fun x -> x)`, et `ifz` V_1 `then` V_2 `else` V_3 où V_1, V_2 et V_3 sont des termes irréductibles et clos et V_1 n'est pas une constante entière, par exemple `ifz (fun x -> x) then 1 else 2`.

Exercice 2.7 *Quelle valeur la sémantique opérationnelle à petits pas donne-t-elle aux termes*

```
(fun x -> fun x -> x) 2 3
```

et

```
(fun x -> fun y -> ((fun x -> (x + y)) x)) 5 4
```

?

Exercice 2.8 *(La liaison statique) La sémantique opérationnelle à petits pas donne-t-elle la valeur* `10` *ou* `11` *au terme*

```
let x = 4 in let f = fun y -> y + x in let x = 5 in f 6
```

?

La sémantique des premiers dialectes de Lisp donnait `11` et non `10` comme valeur de ce terme. On parle, dans ce cas, de *liaison dynamique*.

2.2.6 La non-terminaison

Il est facile de remarquer que la relation $\hookrightarrow$ n'est pas totale, c'est-à-dire qu'il y a des termes `t` tels qu'il n'existe pas de terme `u` tels que `t` $\hookrightarrow$ `u`. Par exemple, le terme `b = fix x x` se réduit sur lui-même, et c'est le seul terme sur lequel il se réduise. De ce fait, il n'existe pas de terme irréductible sur lequel `b` se réduise.

Exercice 2.9 *Soit* $\texttt{b}_1$ `= (fix f (fun x -> (f x))) 0`. *Sur quels termes ce terme se réduit-il ? Le calcul de ce terme donne-t-il un résultat ?*

Exercice 2.10 *(Le point fixe de Curry) Soit* `t` *un terme et* `u` *le terme* `(fun y -> (t (y y)))(fun y -> (t (y y)))`. *Montrer que* `u` *se réduit sur* `t u`.

Soit `t` *un terme et* `v` *le terme* `(fun y -> ((fun x -> t) (y y)))(fun y -> ((fun x -> t) (y y)))`. *Montrer que* `v` *se réduit sur* `(v/x)t`.

En déduire que le symbole `fix` *est superflu en PCF. Il cessera cependant de l'être quand on ajoutera des types à PCF.*

Quel est le terme `u` *ne contenant pas le symbole* `fix` *et exprimant le terme* `b = fix x x` *? Sur quels termes se réduit-il ? Le calcul de ce terme donne-t-il un résultat ?*

2.2.7 La confluence

Le calcul d'un terme clos peut-il donner plusieurs résultats ? Et, plus généralement, un terme peut-il se réduire sur plusieurs termes irréductibles ? On peut montrer que ce n'est pas le cas, c'est-à-dire que tous les programmes écrits en PCF sont déterministes. Cette propriété n'est pas absolument triviale. Essayons de comprendre pourquoi.

Le terme `(3 + 4) + (5 + 6)` a deux sous-termes qui sont des radicaux. On peut donc commencer par réduire le terme `3 + 4` en `7` ou alors le terme `5 + 6` en `11`. De ce fait, le terme `(3 + 4) + (5 + 6)` se réduit à la fois en `7 + (5 + 6)` et en `(3 + 4) + 11`. Heureusement, aucun de ces deux termes n'est irréductible, et si on poursuit le calcul, on aboutit, dans un cas comme dans l'autre, au terme `18`.

Démontrer qu'un terme quelconque se réduit sur un terme irréductible au plus consiste donc à démontrer que si deux calculs issus du même terme partent dans des directions différentes pour aboutir à deux termes irréductibles, alors ces deux termes sont identiques.

Cette propriété est une conséquence d'un autre propriété de la relation $\triangleright$: la *confluence*. Une relation `R` est dite confluente si chaque fois que `a` $\mathtt{R}^*$ $\mathtt{b}_1$ et `a` $\mathtt{R}^*$ $\mathtt{b}_2$, alors il existe `c` tel que $\mathtt{b}_1$ $\mathtt{R}^*$ `c` et $\mathtt{b}_2$ $\mathtt{R}^*$ `c`.

Il n'est pas difficile de démontrer que la confluence implique qu'un terme se réduise sur un terme irréductible au plus. Si le terme `t` se réduit sur deux termes irréductibles $\mathtt{u}_1$ et $\mathtt{u}_2$, alors on a `t` $\triangleright^*$ $\mathtt{u}_1$ et `t` $\triangleright^*$ $\mathtt{u}_2$. La relation $\triangleright$ étant confluente, il existe un terme `v` tel que $\mathtt{u}_1$ $\triangleright^*$ `v` et $\mathtt{u}_2$ $\triangleright^*$ `v`. Comme $\mathtt{u}_1$ est irréductible, le seul terme `v` tel que $\mathtt{u}_1$ $\triangleright^*$ `v` est $\mathtt{u}_1$ lui-même. On en déduit $\mathtt{u}_1$ `=` `v` et de même $\mathtt{u}_2$ `=` `v`. On en conclut que $\mathtt{u}_1$ `=` $\mathtt{u}_2$. Autrement dit, le terme `t` se réduit sur un terme irréductible au plus.

Nous ne donnerons pas ici la démonstration de la confluence de la relation $\triangleright$. L'idée est que quand un terme `t` contient deux radicaux $\mathtt{r}_1$ et $\mathtt{r}_2$, et que $\mathtt{t}_1$ est le terme obtenu en réduisant $\mathtt{r}_1$ et $\mathtt{t}_2$ le terme obtenu en réduisant $\mathtt{r}_2$, alors on peut retrouver les résidus de $\mathtt{r}_2$ dans $\mathtt{t}_1$ et les réduire. De même, on peut réduire les résidus de $\mathtt{r}_1$ dans $\mathtt{t}_2$, et on obtient alors le même terme. Par exemple, en réduisant `5 + 6` dans `7 + (5 + 6)` et en réduisant `3 + 4` dans `(3 + 4) + 11`, on obtient le même terme : `7 + 11`.

2.3 Les stratégies de réduction

2.3.1 La notion de stratégie

D'après la propriété d'unicité des résultats, quel que soit l'ordre dans lequel on réduit les radicaux d'un terme, si on aboutit à un terme irréductible, alors on aboutit toujours au même. En revanche, il se peut qu'une certaine manière de réduire les radicaux mène à un terme irréductible et une autre non. Par exemple, si on note `C` le terme `fun x -> 0` et $\mathtt{b}_1$ le terme `(fix f (fun x -> (f x)))` `0`. Le terme $\mathtt{b}_1$ se réduit sur le terme $\mathtt{b}_2$ `= (fun x -> (fix f (fun x -> (f x)) x))` `0` puis sur $\mathtt{b}_1$ à nouveau. Le terme `C` $\mathtt{b}_1$ contient plusieurs radicaux et il se réduit sur `0` et sur `C` $\mathtt{b}_2$ qui contient, à son tour, plusieurs radicaux et se réduit, parmi d'autres termes, sur `0` et sur `C` $\mathtt{b}_1$. En réduisant toujours le radical interne, on construit une suite de réductions infinie `C` $\mathtt{b}_1$ $\triangleright$ `C` $\mathtt{b}_2$ $\triangleright$ `C` $\mathtt{b}_1$ $\triangleright \ldots$, alors qu'en réduisant le radical externe on aboutit au résultat `0`.

Ce contre-exemple peut paraître exceptionnel puisqu'il contient une fonction `C` qui n'utilise pas son argument. Mais on peut remarquer que la construction `ifz` est similaire et que dans l'exemple de la factorielle de `3`, le même phénomène se produit : le terme `ifz 0 then 1 else 0 * ((fix f fun n -> ifz n then 1 else n * (f (n - 1))) (0 - 1))` contient plusieurs radicaux, réduire le radical le plus externe donne le résultat `1` en faisant disparaître les autres radicaux, alors que réduire le radical `fix f fun n -> ifz n then 1 else n * (f (n - 1))` mène à une réduction à l'infini. Autrement dit, le terme `fact 3` peut se réduire en `6`, mais aussi à l'infini.

Les calculs des termes `C` $\mathtt{b}_1$ et `fact 3` ont donc un résultat unique, mais tous les chemins de réduction ne mènent pas à ce résultat.

Comme, avec notre définition de la sémantique de PCF, le résultat du calcul

du terme `C` b_1 est `0`, quand on écrit un *évaluateur*, c'est-à-dire un programme qui prend en argument un terme de PCF et retourne sa valeur, si on veut suivre cette sémantique de PCF, il faut que le calcul du terme `C` b_1 termine et produise le résultat `0`. Essayons avec quelques compilateurs du marché. En Caml, le programme

```
let rec f x = f x in let g x = 0 in g (f 0)
```

boucle. En Java, c'est également le cas du programme

```
class Omega {
 static int f (int x) {return f(x);}
 static int g (int x) {return 0;}
 static public void main (String [ ] args) {
  System.out.println(g(f(0)));}}
```

Seul un petit nombre de compilateurs, *en appel par nom* ou *paresseux*, comme Haskell, Lazy-ML ou Gaml terminent sur ce terme.

La sémantique opérationnelle à petits pas de PCF ne correspond donc pas à la sémantique de Caml ou de Java, car elle est trop générale. Face à un terme qui contient plusieurs radicaux, elle n'indique pas quel radical doit être réduit, et, par défaut, elle impose la terminaison de tous les programmes qui terminent d'une manière ou d'une autre. Il manque un ingrédient à cette définition de la sémantique d'un langage : la notion de stratégie qui permet de préciser l'ordre dans lequel on doit réduire les radicaux d'un terme.

Une *stratégie* est une fonction partielle qui, à chaque terme de son domaine de définition, associe une occurrence de ce terme qui est un radical. Une fois donnée une stratégie `s` on peut définir une autre sémantique. Au lieu de définir la relation $\triangleright$, on définit une relation $\triangleright_s$ telle que `t` $\triangleright_s$ `u` si `s t` est défini et `u` est obtenu en réduisant le radical `s t` dans `t`. Puis on définit la relation $\triangleright_s^*$ comme la fermeture réflexive-transitive de $\triangleright_s$ et la relation $\hookrightarrow_s$ comme précédemment.

Une alternative à la définition d'une stratégie consiste à affaiblir les règles de réduction, en particulier les règles de congruence, afin de n'autoriser que la réduction de certains radicaux.

2.3.2 La réduction faible

Avant de définir les stratégies qui, dans le terme `C` b_1 réduisent d'abord le radical extérieur ou d'abord le radical intérieur, donnons un autre exemple qui montre que la sémantique opérationnelle à petits pas est trop libérale, et qu'il est nécessaire de la préciser, par une stratégie ou un affaiblissement des règles de réduction. Appliquons le programme `fun x -> x + (4 + 5)` à la constante `3`. Nous obtenons le terme `(fun x -> x + (4 + 5)) 3` qui contient deux radicaux. Nous pouvons ou bien le réduire en `3 + (4 + 5)` ou alors en `(fun x -> x + 9) 3`. La première réduction décrit ce qui se passe quand on exécute un programme, mais pas la seconde. Commencer à exécuter une fonction avant d'avoir reçu ses arguments ne s'appelle pas *exécuter* un programme, mais l'*optimiser* ou le *spécialiser*.

Une stratégie est une *stratégie de réduction faible* si elle ne réduit jamais un radical qui se trouve sous un `fun`. Ainsi, la réduction faible ne spécialise pas les programmes, elle les exécute.

De ce fait, pour la réduction faible, les termes de la forme `fun x -> t` sont toujours irréductibles.

De manière équivalente, on peut définir la réduction faible en affaiblissant les règles de réduction. On supprime alors la règle de congruence

$$\frac{\texttt{t} \rhd \texttt{u}}{\texttt{fun x -> t} \rhd \texttt{fun x -> u}}$$

Exercice 2.11 *(La classification des termes irréductibles pour réduction faible et clos) Montrer que, pour la réduction faible, un terme est irréductible et clos si et seulement s'il est de l'une des formes suivantes*

- `fun x -> t`*, où* `t` *ne contient pas d'autres variables libres que* `x`*,*
- `n` *où* `n` *est une constante entière,*
- $\texttt{V}_1\ \texttt{V}_2$*, où* $\texttt{V}_1$ *et* $\texttt{V}_2$ *sont des termes irréductibles et clos et* $\texttt{V}_1$ *n'est pas de la forme* `fun x -> t`*,*
- $\texttt{V}_1 \otimes \texttt{V}_2$*, où* $\texttt{V}_1$ *et* $\texttt{V}_2$ *sont des termes irréductibles et clos qui ne sont pas tous les deux des constantes entières,*
- `ifz` $\texttt{V}_1$ `then` $\texttt{V}_2$ `else` $\texttt{V}_3$ *où* $\texttt{V}_1$*,* $\texttt{V}_2$ *et* $\texttt{V}_3$ *sont des termes irréductibles et clos et* $\texttt{V}_1$ *n'est pas une constante entière.*

Quelle est la différence avec l'exercice 2.6 ?

Les constantes entières et les termes clos de la forme `fun x -> t` sont appelés *valeurs*.

2.3.3 L'appel par nom

Revenons maintenant à la question de l'ordre dans lequel on réduit les radicaux dans le terme $\texttt{C}\ \texttt{b}_1$. Cette question se ramène à celle de savoir si on doit évaluer les arguments de la fonction `C` avant de les passer à la fonction ou si on peut passer à la fonction des arguments non encore évalués.

La stratégie de réduction en *appel par nom* consiste à toujours réduire le radical le plus à gauche dans le terme et la stratégie de réduction faible en appel par nom consiste à toujours réduire le radical le plus à gauche dans le terme parmi ceux qui ne sont pas sous un `fun`. Ainsi, le terme $\texttt{C}\ \texttt{b}_1$ se réduit sur `0`. La stratégie en appel par nom doit son intérêt à la propriété suivante — le théorème de standardisation — : si un terme se réduit sur un terme irréductible, alors la réduction en appel par nom termine. Autrement dit, $\hookrightarrow_{\texttt{n}} = \hookrightarrow$.

De plus, quand on évalue le terme `(fun x -> 0) (fact 10)` en appel par nom, on n'a pas besoin de calculer la factorielle de `10`. En revanche, quand on évalue le terme `(fun x -> x + x) (fact 10)`, on calcule cette factorielle deux fois car ce terme se réduit en `(fact 10) + (fact 10)`. La plupart des évaluateurs en appel par nom utilisent une forme de partage pour éviter cette redondance des calculs. On parle alors de réduction *paresseuse*.

2.3.4 L'appel par valeur

La réduction en *appel par valeur*, en revanche, consiste à toujours évaluer les arguments d'une fonction avant de les passer à la fonction. Elle repose sur la convention suivante : on peut réduire un terme de la forme `(fun x -> t) u` uniquement quand `u` est une valeur. Ainsi, quand on évalue le terme `(fun x -> x + x) (fact 10)`, on commence par réduire l'argument jusqu'à obtenir `(fun x -> x + x) 3628800` avant de réduire le radical de gauche. De ce fait, on ne calcule la factorielle de `10` qu'une seule fois.

Toutes les stratégies qui évaluent les arguments d'une fonction avant de les passer à la fonction sont en appel par valeur. Par exemple, la stratégie qui réduit systématiquement le radical le plus à gauche parmi ceux qui sont autorisés. Ainsi, l'appel par valeur n'est pas une stratégie unique, mais une famille de stratégies.

Cette convention peut également s'exprimer comme un affaiblissement de la règle β : le terme `(fun x -> t) u` n'est un radical que si le terme `u` est une valeur.

Une stratégie de réduction faible est en appel par valeur si elle réduit un terme de la forme `(fun x -> t) u` uniquement quand ce terme n'est pas sous un `fun` et que, de plus, le terme `u` est une valeur.

2.3.5 Un peu de paresse est nécessaire

Cependant, même en appel par valeur, la construction `ifz` doit toujours rester en appel par nom : pour évaluer un terme de la forme `ifz t then u else v`, il ne faut surtout pas évaluer les trois arguments, mais il faut évaluer `t` puis, en fonction du résultat de cette évaluation, évaluer ou bien `u` ou bien `v`.

Il est facile de montrer que si on évalue systématiquement les trois arguments d'un `ifz`, alors l'évaluation du terme `fact 3` ne termine pas.

Exercice 2.12 *Pour la réduction faible en appel par nom, quels sont les termes irréductibles et clos ? Et pour la réduction faible en appel par valeur ?*

2.4 La sémantique opérationnelle à grands pas de PCF

Au lieu de définir une stratégie, ou d'affaiblir les règles de réduction, une alternative pour contrôler l'ordre dans lequel les radicaux sont réduits est de définir une sémantique opérationnelle à grands pas.

La sémantique opérationnelle à grands pas d'un langage de programmation consiste à définir inductivement la relation $\hookrightarrow$ sans passer par les relations $\longrightarrow$ et $\rhd$.

2.4.1 En appel par nom

Commençons par la sémantique de PCF en appel par nom. Imaginons un terme de la forme `t u` que l'on réduit en appel par nom jusqu'à obtenir un terme irréductible `V`. On commence par réduire les radicaux qui se trouvent dans `t` jusqu'à obtenir un terme irréductible. Si ce terme a la forme `fun x -> t'`, alors le terme complet a été réduit en `(fun x -> t') u` et le radical le plus à gauche est désormais le terme lui-même. On le réduit sur le terme `(u/x)t'`, que l'on réduit ensuite sur le terme irréductible `V`. On peut donc dire que le terme `t u` se réduit en appel par nom sur le terme irréductible `V` si `t` se réduit en `fun x -> t'` et `(u/x)t'` se réduit en `V`.

Cela s'exprime par une règle

$$\frac{\texttt{t} \hookrightarrow \texttt{fun x -> t'} \quad \texttt{(u/x)t'} \hookrightarrow \texttt{V}}{\texttt{t u} \hookrightarrow \texttt{V}}$$

qui participe à une définition inductive de la relation $\hookrightarrow$ sans passer par les relations $\longrightarrow$ et $\rhd$.

D'autres règles expriment que le résultat du calcul d'un terme de la forme `fun` est ce terme lui-même, c'est-à-dire que l'on ne fait que de la réduction faible

$$\frac{}{\texttt{fun x -> t} \hookrightarrow \texttt{fun x -> t}}$$

et que le résultat du calcul d'un terme de la forme `n` est ce terme lui-même

$$\frac{}{\texttt{n} \hookrightarrow \texttt{n}}$$

une règle exprime la sémantique des opérateurs arithmétiques

$$\frac{\texttt{u} \hookrightarrow \texttt{q} \qquad \texttt{t} \hookrightarrow \texttt{p}}{\texttt{t} \otimes \texttt{u} \hookrightarrow \texttt{n}} \text{ si } \texttt{p} \otimes \texttt{q} = \texttt{n}$$

deux règles expriment la sémantique de la construction `ifz`

$$\frac{\texttt{t} \hookrightarrow \texttt{0} \qquad \texttt{u} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{n} \qquad \texttt{v} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}} \quad \begin{array}{l}\text{si n constante} \\ \text{entière} \neq \texttt{0}\end{array}$$

une règle exprime la sémantique du point fixe

$$\frac{\texttt{(fix x t/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

et une dernière règle exprime la sémantique du `let`

$$\frac{\texttt{(t/x)u} \hookrightarrow \texttt{V}}{\texttt{let x = t in u} \hookrightarrow \texttt{V}}$$

On peut montrer par récurrence structurelle sur la relation d'évaluation que le résultat du calcul d'un terme est toujours une valeur, c'est-à-dire une constante entière ou un terme clos de la forme `fun`. On s'est donc débarrassé des termes bloqués. Le calcul du terme `((fun x -> x) 1) 2` qui donnait un terme bloqué `1 2` comme un résultat, en sémantique opérationnelle à petits pas, ne donne pas de résultat en sémantique opérationnelle à grands pas, car aucune règle ne s'applique à ce terme. En effet, il n'y a pas, en sémantique opérationnelle à grands pas, de règle qui indique comment évaluer une application dont la partie gauche s'évalue sur une constante entière.

2.4.2 En appel par valeur

Les règles de l'appel par valeur sont les mêmes que celles de l'appel par nom à l'exception de la règle de l'application : on calcule la valeur de l'argument avant de la passer à la fonction

$$\frac{\texttt{u} \hookrightarrow \texttt{W} \qquad \texttt{t} \hookrightarrow \texttt{fun x -> t'} \qquad \texttt{(W/x)t'} \hookrightarrow \texttt{V}}{\texttt{t u} \hookrightarrow \texttt{V}}$$

et de la règle du `let`

$$\frac{\texttt{t} \hookrightarrow \texttt{W} \quad \texttt{(W/x)u} \hookrightarrow \texttt{V}}{\texttt{let x = t in u} \hookrightarrow \texttt{V}}$$

Cela donne les règles suivantes

$$\frac{\texttt{u} \hookrightarrow \texttt{W} \qquad \texttt{t} \hookrightarrow \texttt{fun x -> t'} \qquad \texttt{(W/x)t'} \hookrightarrow \texttt{V}}{\texttt{t u} \hookrightarrow \texttt{V}}$$

$$\frac{}{\texttt{fun x -> t} \hookrightarrow \texttt{fun x -> t}}$$

$$\frac{}{\texttt{n} \hookrightarrow \texttt{n}}$$

$$\frac{\texttt{u} \hookrightarrow \texttt{q} \qquad \texttt{t} \hookrightarrow \texttt{p}}{\texttt{t} \otimes \texttt{u} \hookrightarrow \texttt{n}} \text{ si } \texttt{p} \otimes \texttt{q} = \texttt{n}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{0} \qquad \texttt{u} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{n} \qquad \texttt{v} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}} \quad \begin{array}{l}\text{si n constante} \\ \text{entière} \neq \texttt{0}\end{array}$$

$$\frac{\texttt{(fix x t/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{W} \quad \texttt{(W/x)u} \hookrightarrow \texttt{V}}{\texttt{let x = t in u} \hookrightarrow \texttt{V}}$$

Remarquons que, même en appel par valeur, on garde les règles qui expriment la sémantique de la construction `ifz`

$$\frac{\texttt{t} \hookrightarrow \texttt{0} \qquad \texttt{u} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{n} \qquad \texttt{v} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}} \quad \begin{array}{l}\text{si n constante}\\ \text{entière} \neq \texttt{0}\end{array}$$

c'est-à-dire que l'on évalue le deuxième ou le troisième argument du `ifz` que quand cela est nécessaire.

Remarquons également que, même en appel par valeur, on garde la règle

$$\frac{\texttt{(fix x t/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

Il faut, en effet, résister à la tentation d'évaluer le terme `fix x t` en une valeur `W` avant de le substituer dans `t`, car la règle

$$\frac{\texttt{fix x t} \hookrightarrow \texttt{W} \qquad \texttt{(W/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

demanderait, pour évaluer `fix x t`, de commencer par évaluer `fix x t` ce qui est circulaire et le terme `fact 3` n'aurait pas de valeur — son évaluation mènerait à des calculs infinis.

Remarquons enfin que d'autres combinaisons de règles sont possibles. Par exemple, certaines variantes de la sémantique en appel par nom utilisent la règle du `let` en appel par valeur.

Exercice 2.13 *Quelle valeur la sémantique opérationnelle à grands pas donne-t-elle aux termes*

```
(fun x -> fun x -> x) 2 3
```

et

```
(fun x -> fun y -> ((fun x -> (x + y)) x)) 5 4
```

? Comparer avec l'exercice 2.7.

Exercice 2.14 *La sémantique opérationnelle à grands pas donne-t-elle la valeur* `10` *ou* `11` *au terme*

```
let x = 4 in let f = fun y -> y + x in let x = 5 in f 6
```

? Comparer avec l'exercice 2.8.

2.5 L'évaluation de PCF

Un évaluateur de PCF est un programme qui prend en argument un terme clos de PCF et retourne sa valeur. Quand on les lit de bas en haut, les règles de la sémantique opérationnelle à grands pas donnent le squelette d'un tel évaluateur : pour évaluer une application `t u` il faut commencer par évaluer `u` et `t`, ... cela se programme simplement dans un langage comme Caml

```
let rec eval p = match p with
| App(t,u) -> let w = eval u
              in let v = eval t
              in ...
| ...
```

Dans le cas d'une application, les règles de la sémantique opérationnelle de PCF laissent la liberté d'évaluer `u` en premier ou `t` en premier — ce qui traduit le fait que l'appel par valeur n'est pas une stratégie, mais une famille de stratégies —, mais le terme `(W/x)t'` doit être évalué en troisième car il est construit à partir du résultat des deux premières évaluations.

Exercice 2.15 *Écrire un évaluateur en appel par nom pour PCF, c'est-à-dire un programme qui prend en argument un terme clos et calcule sa valeur. Écrire un évaluateur en appel par valeur. Évaluer le terme* `fact 6` *et le terme* `C` b_1 *dans les deux cas.*

La sémantique dénotationnelle de PCF est beaucoup plus difficile à définir. On peut y voir un paradoxe car, PCF étant un langage fonctionnel, interpréter les programmes comme des fonctions ne devrait pas être trop difficile. Cependant, en PCF, n'importe quel objet peut s'appliquer à n'importe quel objet, rien n'empêche, par exemple de construire le terme `fun x -> (x x)`. Contrairement aux fonctions mathématiques, les fonctions de PCF n'ont donc pas de domaine de définition. Nous ne donnerons donc une sémantique dénotationnelle à PCF qu'après avoir ajouté des types, au chapitre 5.