Expression Logique et Fonctionnelle . . . Évidemment

Cours n° 3 : Sémantique

Ces notes de cours sont très largement inspirées du chapitre 3 du livre *Approche fonctionnelle de la programmation* de Guy Cousineau et Michel Mauny, Ediscience, 1995.

1 Introduction

But : définir de façon précise la valeur associée à chaque expression (sémantique du langage).

Plusieurs moyens :

– associer à chaque expression du langage un objet mathématique : *sémantique dénotationnelle.*

– réduire les expressions du langage par réécriture : *sémantique par récriture.*

La sémantique par réécriture est une sémantique opérationnelle qui montre le *comment* on obtient la valeur d'une expression. Tandis que la sémantique dénotationnelle est une sémantique déclarative qui montre uniquement *quelle* est la valeur d'une expression.

C'est la sémantique par réécriture qui sera présentée dans ce chapitre sans formaliser excessivement sa présentation.

Le cas des fonctions récursives n'est pas abordé. Il le sera ultérieurement.

2 Évaluation vue comme une réécriture

Évaluer une expression c'est établir une suite de transformations conduisant de cette expression à sa valeur. Chaque transformation est appelée une *réduction.*

La *valeur* d'une expression e est une expression e' obtenue à partir de e par réductions qu'on ne peut plus réduire.

Exemple 1 :
L'évaluation de l'expression `e=(3+4)*(4+1)` peut s'effectuer en trois réductions :

$$\begin{aligned}(3+4)*(4+1) &\Rightarrow (3+4)*5\\ &\Rightarrow 7*5\\ &\Rightarrow 35.\end{aligned}$$

La première réduction a un caractère arbitraire dans le choix de la sous-expression à réduire. On aurait pu envisager de commencer par réduire la sous-expression gauche de `e` :

$$\begin{aligned}(3+4)*(4+1) &\Rightarrow 7*(4+1)\\ &\Rightarrow 7*5\\ &\Rightarrow 35.\end{aligned}$$

Deux propriétés sont requises pour de tels calculs :

– *Finitude* : tous les calculs se terminent.

– *Cohérence* : tous les calculs conduisent au même résultat.

En l'absence de calculs infinis, la cohérence est assurée dès lors que les règles de réécriture forment un système *confluent.* La *confluence* d'un système de réécriture est la propriété que ce système possède si dès qu'une expression e peut se réduire en une étape en deux expressions e_1 et e_2 distinctes, alors il existe une troisième expression e_3 telle que les deux expressions e_2 et e_3 se réduisent (en éventuellement plusieurs étapes) en e_3.

En présence de fonctions récursives la propriété de finitude des réductions n'est plus assurée, et la confluence n'implique pas la cohérence des calculs. En revanche elle implique une propriété affaiblie :

– *Cohérence faible :* tous les calculs qui se terminent conduisent au même résultat.

3 Réécriture des applications

Quelle règles de réécriture adopter pour les expressions ayant la forme de l'application d'une fonction à une expression, i.e. une expression de la forme

`(function x -> e1) e2` ?

Naturellement on est conduit à penser que la réduction d'une telle expression doit donner l'expression `e1` dans laquelle toutes les occurrences libres de la variable `x` ont été remplacées par la `e2` (ou une expression obtenue par réduction de `e2`, cf startégies plus loin).

3.1 Substitutions

On note `e1[x ← e2]` l'expression obtenue en remplaçant toutes les occurrences libres de la variable `x` dans `e1` par l'expression `e2`.

- Si `x` est une variable,
$$x[x \leftarrow e] = e.$$
- Si `y` est une variable autre que `x` :
$$y[x \leftarrow e] = y.$$
- Application :
$$(e1\ e2)[x \leftarrow e] = (e1[x \leftarrow e])\ (e2[x \leftarrow e]).$$
- Abstraction1 :
`(function x -> e1)[x←e]` = `(function x -> e1)`.
- Abstraction2, si `y`≠`x` :
`(function y -> e1)[x←e]` = `(function y -> e1[x←e])`.

Exemple 2 :

1. `(y + x y)[x ← (function x -> x)]`=`(y + (function x -> x) y)`
2. `(x + (function x -> x) y)[x ← 2*y]`=`(2*y + (function x -> x) y)`

Attention aux conflits de noms : Si on ne fait pas attention, des conflits de noms peuvent apparaître. Par exemple, la substitution `(function y -> x*y)[x ← 2*y]` ne doit pas donner `(function y -> (2*y)*y)` car la variable `y` qui est libre dans l'expression `2*y` entre sous la portée de l'abstraction.

Pour obtenir une substitution correcte, il faut préalablement renommer le paramètre `y` de l'expression fonctionnelle, et on a alors

`(function z -> x*z)[x ← 2*y]` = `(function z -> (2*y)*z)`.

3.2 Réduction d'une application

Régle de réduction d'une application :

`(function x -> e1) e2` ⇒ `e1[x ← e2]`.

Exemple 3 :

$$\begin{array}{lcl} \texttt{(function x -> x + x) (3*4)} & \Rightarrow & \texttt{(function x -> x + x) 12} \\ & \Rightarrow & \texttt{12 + 12} \\ & \Rightarrow & \texttt{24}. \end{array}$$

4 Autres règles de réduction

4.1 L'expression conditionnelle

`if true then e1 else e2` ⇒ `e1`

`if false then e1 else e2` ⇒ `e2`

4.2 L'expression `let ... in ...`

Si on se souvient que l'expression **let** x = e1 **in** e2 est équivalente à l'expression (**function** x -> e2) e1, on ne sera pas surpris de la règle de réduction

let x = e1 **in** e2 $\Rightarrow$ e2[x $\leftarrow$ e1].

4.3 Les n-uplets

Si la i-ème composante d'un n-uplet (e1, ...,ei, ...,en) se réduit en ei', alors

(e1, ...,ei, ...,en) $\Rightarrow$ (e1, ...,ei', ...,en)

5 Stratégies

Reprenons l'évaluation de l'expression de l'exemple 3. Au lieu de réduire d'abord la sous-expression (3 * 4), on aurait pu utiliser en priorité la règle de réduction d'une application, et on aurait alors obtenu

(**function** x -> x + x) (3 * 4) $\Rightarrow$ (3 * 4) + (3 * 4),

et la réduction peut se poursuivre ensuite en

$$\begin{array}{ll} \Rightarrow & \texttt{12 + (3 * 4)} \\ \Rightarrow & \texttt{12 + 12} \\ \Rightarrow & \texttt{24}. \end{array}$$

Comme on le voit, il n'y a pas unicité du calcul de la valeur d'une expression.

On appelle *stratégie d'évaluation* toute procédure déterministe fixant pour chaque type d'expression la réduction à appliquer.

Nous allons étudier deux stratégies :

– la stratégie par *valeur*

– et la stratégie par *nécessité* ou stratégie *paresseuse* (*lazy evaluation*).

5.1 Stratégie par valeurs

C'est la stratégie la plus courante en particulier celle suivie par CAML.

Dans cette stratégie, les arguments des applications de fonctions sont évalués avant d'être transmis à la fonction, et la valeur d'une structure de données (n-uplet) est cette structure dans laquelle les composantes sont évaluées.

Avant de présenter cette stratégie, commençons par fixer les notations utilisées. La notation

$$\vdash \texttt{e} \rightarrow v$$

signifie

l'expression e a pour valeur v.

Les valeurs sont des expressions qu'il n'est plus possible de réduire. Intuitivement les données de base (entiers, flottants, booléens, caractères) sont des valeurs. Les règles qui suivent indiquent précise ce que sont les valeurs des n-uplets et des expressions fonctionnelles.

L'évaluation de chacune des constructions du langage sera présentée sous la forme d'une règle notée

$$\frac{\text{Hypothèses}}{\text{Conclusion}}\ (\text{Nom})$$

Évaluation d'une expression conditionnelle :

$$\frac{\vdash \texttt{e1} \Rightarrow \textbf{true} \qquad \vdash \texttt{e2} \Rightarrow v}{\vdash \textbf{if}\ \texttt{e1}\ \textbf{then}\ \texttt{e2}\ \textbf{else}\ \texttt{e3} \Rightarrow v}\ (\text{Cond1}) \qquad \frac{\vdash \texttt{e1} \Rightarrow \textbf{false} \qquad \vdash \texttt{e3} \Rightarrow v}{\vdash \textbf{if}\ \texttt{e1}\ \textbf{then}\ \texttt{e2}\ \textbf{else}\ \texttt{e3} \Rightarrow v}\ (\text{Cond2})$$

Évaluation d'une abstraction :

$$\frac{}{\vdash \textbf{function}\ \texttt{x -> e} \Rightarrow \textbf{function}\ \texttt{x -> e}}\ (\text{Abstr})$$

Évaluation d'un n-uplets :

$$\frac{\vdash \texttt{e1} \Rightarrow v_1 \qquad \ldots \qquad \vdash \texttt{en} \Rightarrow v_n}{\vdash \texttt{(e1,...,en)} \Rightarrow (v_1, \ldots, v_n)}\ (N\text{-uplet})$$

Évaluation d'une application :

$$\frac{\vdash \texttt{e1} \Rightarrow \textbf{function}\ \texttt{x -> e} \qquad \vdash \texttt{e2} \Rightarrow v_2 \qquad \vdash \texttt{e}[\texttt{x} \leftarrow v_2] \Rightarrow v}{\vdash \texttt{e1 e2} \Rightarrow v}\ (\text{Appl1})$$

$$\frac{\vdash \texttt{e1} \Rightarrow \textbf{function}\ \texttt{(x1,...,xn) -> e} \qquad \vdash \texttt{e2} \Rightarrow (v_1, \ldots, v_n) \qquad \vdash \texttt{e}[\texttt{x}_i \leftarrow v_i] \Rightarrow v}{\vdash \texttt{e1 e2} \Rightarrow v}\ (\text{Appln})$$

5.2 Stratégie par nécessité

Évaluation d'un n-uplet :

$$\frac{}{\vdash \texttt{(e1,...,en)} \Rightarrow \texttt{(e1,...,en)}}\ (N\text{-uplet})$$

Évaluation d'une application :

$$\frac{\vdash \texttt{e1} \Rightarrow \textbf{function}\ \texttt{x -> e} \qquad \vdash \texttt{e}[\texttt{x} \leftarrow \texttt{e2}] \Rightarrow v}{\vdash \texttt{e1 e2} \Rightarrow v}\ (\text{Appl1})$$

$$\frac{\vdash \texttt{e1} \Rightarrow \textbf{function}\ \texttt{(x1,...,xn) -> e} \qquad \vdash \texttt{e2} \Rightarrow (v_1, \ldots, v_n) \qquad \vdash \texttt{e}[\texttt{x}_i \leftarrow v_i] \Rightarrow v}{\vdash \texttt{e1 e2} \Rightarrow v}\ (\text{Appln})$$

5.3 Espionner la stratégie suivie par CAML

Espionner les règles d'évaluation suivies par CAML en introduisant des effets de bords par affichages de messages.

Pour chaque expression (ou sous-expression) qu'on cherche à observer, on fait précéder son évaluation par une impression d'un message.

Par exemple, pour faire précéder l'évaluation de `3 + 4` par l'impression du message `elfe`, on écrit la séquence

```
# Printf.printf "%s\n" "elfe" ; 3 + 4 ;;
elfe
- : int = 7
```

ou dans un style plus fonctionnel

```
# (function () -> 3 + 4) (Printf.printf "%s\n" "elfe") ;;
elfe
- : int = 7
```

On peut appliquer cet espionnage à des sous-expressions.

Par exemple, l'espionnage

```
# ((function () -> 3) (Printf.printf "%s\n" "3"))
  +
  ((function () -> 4) (Printf.printf "%s\n" "4")) ;;
4
3
- : int = 7
```

nous apprend que la sous-expression `4` est évaluée avant `3`.