Chapitre 4

Évaluation, environnement, variables et références

4.1 Évaluation, appel par nom et appel par valeur

L'exécution d'un programme Caml est basé sur l'évaluation d'**expressions** et leur **réduction** vers une **valeur**. Nous présentons maintenant les différentes stratégies d'évaluation qui peuvent être mises en œuvre[1].

Supposons que nous ayons défini une fonction calculant le carré de deux entiers comme suit :

```
let carre x=x*x;;
```

et que nous cherchions maintenant à évaluer l'expression :

```
carre(carre(carre 3));;
```

Le résultat sera toujours le même (6561) mais il existe plusieurs moyens d'y arriver. Commençons par voir comment procède Caml. Pour cela traçons[2] les appels à la fonction `carre` lors de l'évaluation :

```
#trace "carre";;
The function carre is now traced.
- : unit = ()
#carre(carre(carre 3));;
carre <-- 3
carre --> 9
carre <-- 9
carre --> 81
carre <-- 81
carre --> 6561
- : int = 6561
```

On constate que comme nous l'avons implicitement supposé au chapitre précédent, les arguments de la fonction sont évalués en premier. (ici `3`, puis carre `3`, puis `carre 9`, puis `carre 81`). (Remarquez que c'est aussi comme cela que l'on procède généralement lorsque l'on calcule à la main.)

D'une manière générale, pour évaluer $f(E_1\ \ldots\ E_i)$ on commence par évaluer tous les E_I puis on évalue le corps de f en remplacant les paramètres par les valeurs calculées.

---

[1] Une présentation plus détaillée de leurs avantages respectifs est donnée dans [?] ou dans [?]

[2] D'une façon générale, utilisez `trace` dès que vous voulez comprendre le comportement d'une fonction récursive

Cette forme de réduction se nomme **appel par valeur**.

Elle a cependant quelques désavantages. comme le montre l'exemple suivant :

Exemple 1. Considérons la fonction constante suivante

```
let zero x =0;;
```

elle renvoie `0` quelque soit son argument. Il est donc trivial de savoir sans aucun calcul que

```
zero (carre(carre(carre 3)));;
```

se reduit (ou s'evalue, ou vaut...) à 0. Pourtant pratiquant l'évaluation par valeur, caml doit d'abord calculer la valeur de l'argument même si cela est inutile par la suite. On peut le vérifier en traçant l'évaluation :

```
#zero (carre(carre(carre 3)));;
carre <-- 3
carre --> 9
carre <-- 9
carre --> 81
carre <-- 81
carre --> 6561
- : int = 0
```

Pour éviter ces évaluation inutiles, d'autres statégies d'évaluation sont possibles (attention pas en Caml). On aurait pu commencer par évaluer le corps de la fonction `zero` et constater qu'il n'utilise pas l'argument qui par conséquent n'a donc nul besoin d'être evalué.

Cela nous amène à deux autres types d'évaluations : l'**évaluation par nom** et l'**évaluation par néssécité** aussi appelée **évaluation paresseuse**.

L'idée de l'évaluation par nom et que pour évaluer $f(E_1\ \dots\ E_i)$ dans le corps de f les parametres formels non plus par F par les valeurs des E_i mais par les expressions elles-même. Dans l'exemple précédent comme le paramètre formel n'apparaît pas dans corps le calcul se termine en une étape mais la vie n'est pas toujours aussi rose comme le montre l'exemple suivant :

Exemple 2. Simulons l'évaluation par nom de `(carre(carre(carre 3)))` :

```
(carre(carre(carre 3)))
=(carre(carre 3))*(carre(carre 3))
=((carre 3)*(carre 3))*(carre(carre 3))
=((3*3)*(carre 3))*((carre 3)*(carre 3))
=(9*(carre 3))*((carre 3)*(carre 3))
=(9*(3*3))*((carre 3)*(carre 3))
=...

=81*81
=6561
```

On voit que cela va être très inéficace car le calcul de `(carre 3)` sera fait quatre fois au lieu d'une.

Certains langages fonctionels notamment Haskell (`http://haskell.org/`), proposent une variante de l'évaluation par nom dite évaluation paresseuse (ou appel par nécéssité) qui va permètre de n'évaluer chaque expression qu'une seule fois. Ainsi dans notre exemple, lorsqu'une occurence de `(carre 3)` sera évaluée, toutes les autres seront immédiatement remplacées par la valeur trouvée sans avoir besoin de les recalculer. Nous ne rentrerons pas dans les détails de fonctionnement et renvoyons le lecteur interessé à la bibliographie.

Remarque 1. Caml pratique l'évaluation par valeur, mais il possède cependant quelques constructions pratiquant l'appel par nécéssité : Le `or`, le `&` et le `if c then a else b` , ou les flux.

(`e1 or e2` n'evalue `e2` que si `e1` est faux)

Exercice 1. Comparer :

```
2=3 or (1/0)=4;;
2=2 or   (1/0)=4;;
(1/0)=4 or  2=2
if 1=2 then 1/0 else 1;;
if 1=2 then 1 else 1/0;;
 2=3 & (1/0)=4;;
 2=2 & (1/0)=4;;
```

4.2 Environnement

Au démarrage du système, vous disposez d'un certain nombre de fonctions prédéfinies (celles de la bibliothèque de base (`core library`) et de la bibliothèque standard (`standard library`)). Cela constitue l'environnement initial $\lfloor E_{init} \rfloor$.

Un environnement est une liste de couples dont le premier membre est un identificateur et le second la valeur associée à cet identificateur. À chaque nouvelle définition (par un `let`) l'environnement est étendu, c'est-à- dire que l'on rajoute une paire représentant la nouvelle définition en tête de la liste représentant l'environnement.

Exemple 3. Après la session suivante :

```
#let a=2;;
a : int = 2
#let b=3;;
b : int = 3
#let c=4;;
c : int = 4
```

l'environnement est le suivant : $\lfloor$(c,4);(b,3)(a,2)$\triangleleft E_{init} \rfloor$

Remarque 2. Ces identificateurs qui dans le monde des langages fonctionnels sont parfois appelés (en référence au λcalcul) des variables libres correspondent à ce que l'on nomme constantes dans les langages comme Pascal ou C.

Lorsque l'on demande l'évaluation d'un identificateur, sa valeur est cherchée dans l'environnement. Si elle ne s'y trouve pas, le système produit un message signalant que cet identificateur est inconnu.

Exemple 4. Avec les déclarations précédentes :

```
#b;;
- : int = 3
#d;;
Toplevel input:
>d;;
>^
The value identifier d is unbound.
#
```

On peut redéfinir un identificateur.

Exemple 5. On redéfinit `b` :

```
let b=10;;
b : int = 10
```

la déclaration est ajoutée en tête de l'environnement qui devient :
$\lfloor$(b,10);(c,4);(b,3);(a,2)$\triangleleft$ $E_{init}\rfloor$

Si après cette redéfinition nous demandons à nouveau la valeur de `b` c'est la dernière définition (celle qui est le plus près de la tête de la liste) qui est prise en considération.

Exemple 6. Après la redéfinition précédente :

```
#b;;
- : int = 10
```

La valeur associée à un identificateur peut aussi être une valeur fonctionnelle.

Exemple 7.

```
#let carre x = x*x;;
carre : int -> int = <fun>
```

associe à l'identificateur `carre` la fonction $x \rightarrow x * x$ [3]. Dans une telle définition, `x` est appelé un paramètre formel.

[3] ou avec les notations du λ-calcul : $\lambda x.x * x$

On pourrait penser que lors d'une telle définition, il suffit d'ajouter un couple (`carre, fun x-> x*x`) en tête de notre environnement. Dans un tel cas, dans un environnement $\lfloor E \rfloor$ que nommerons **environnement appelant** lorsque nous évaluerons l'application d'une fonction, les valeurs respectives des paramètres effectifs (ceux que l'on passe à la fonctions) sont liés aux paramètres formels (ceux de la définition), l'environnement est momentanément étendu avec ces nouvelles liaisons pour obtenir un environnement d'évaluation. On évalue le corps de la fonction dans cet environnement étendu, enfin on restaure l'environnement appelant.

Exemple 8. Évaluons `carre c` dans l'environnement :
$\lfloor E_{appelant} \rfloor \equiv \lfloor$(b,10);(carre,fun x->x*x);(c,4);(b,3);(a,2)$\triangleleft$ $E_{init} \rfloor$

Dans l'environnement appelant, le paramètre `c` est évalué, sa valeur est `4`, on étend donc momentanément l'environnement en associant la valeur du paramètre effectif `c` au paramètre formel `x`. On obtient un **environnement d'évaluation** :
$\lfloor E_{evaluation} \rfloor \equiv \lfloor$(x,4)(b,10);(carre,fun x->x*x);(c,4);(b,3);(a,2)$\triangleleft$ $E_{init} \rfloor$

Dans lequel on évalue `x*x` avant de **restaurer l'environnement appelant**.

Neanmoins nous allons voir que ce modèle ne convient pas pour expliquer le comportement de Caml.

Exemple 9. Considérons la session suivante :

```
let c=2
let a=4;;
let f x = x+a;;
let a =10;;
f c;;
```

Quelle sera la valeur de `f c` ? Deux solution sont possibles :

1. Si c'est la dernière valeur de `a` qui est prise en compte, on a
   f c $\rightsquigarrow$ 12
2. Si c'est la valeur de `a` au moment de la définition de `f` qui est prise en compte, on a
   f c $\rightsquigarrow$ 6

Dans le premier cas on parle de **liaison dynamique (c'est la cas en Lisp)**. **Dans le** second **cas** qui, est celui **de Caml**, on parle de **liaison statique**.

La liaison dynamique s'explique très bien par le modèle précédent, pour caractériser la liaison statique, il faut pourvoir associer au nom d'une fonction non seulement le code à exécuter mais aussi l'environnement au moment de la définition de la fonction.

Pour cela, on introduit le concept de **fermeture**. Une fermeture est un doublet contenant le code d'une fonction et l'environnement de la fonction au moment de sa définition.

Lorsqu'on défini une nouvelle fonction, cette fermeture est calculée et une liaison (identificateur, fermerture) est ajoutée en tête de l'environnement.

Ensuite le principe d'évaluation est le suivant :

1. évaluer les paramètres effectifs dans l'environnement appelant $E_{appelant}$,
2. évaluer la fermeture et retrouver l'environnement de définition $E_{dfinition}$ et le code de la fonction,
3. étendre l'environnement de définition en ajoutant les liaisons, associant à chaque paramètre formel la valeur du paramètre effectif calculé en 1,
4. évaluer le corps de la fonction dans cet environnement étendu,
5. restaurer l'environnement appelant.

Exemple 10. Considéreons la session de l'exemple 4.2 Au moment où on cherche à évaluer `f c` l'environnement est :
$\lfloor E\rfloor \equiv \lfloor$(a,10);(f,(fun x->x*a),((a,4);(c,2)));(a,4);(c ,2)$\triangleleft E_{init}\rfloor$
l'evaluation s'effectue alors comme suit :

1. évaluer `c` dans l'environnement appelant. On obtient `2`.
2. récupérer l'environnement de la fermeture et l'étendre par la liaison `(x,2)`, on obtient l'environement d'évaluation suivant :
   $\lfloor E\rfloor \equiv \lfloor$(x,2);(a,4);(c ,2)$\triangleleft E_{init}\rfloor$
3. évaluer le corps de la fonction dans cet environement, Il vient :
   `2+4 =6`
4. réstaurer l'environnement appelant (l'environmement d'évaluation est détruit)

Pourquoi le mot clef `rec` ?

Considérons la fonction factorielle, et regardons ce qui se passe si on essaie de la définir sans le mot clef `rec`

```
#let  fac n = if n=0 then 1 else n*(fac (n-1));;
Toplevel input:
>let  fac n = if n=0 then 1 else n*(fac (n-1));;
>                                   ^^^
The value identifier fac is unbound.
```

Avant toute chose, Caml a cherché à typer le corps de la fonction et a évaluer ce corps tous les identificateurs qui ne sont pas des paramètres. Or avant de l'avoir défini, il ne connait pas `fac`. Il faut donc lui signaler que l'identificateur `fac` doit être ajouté à l'environnement avant la définition. On peut lui associer n'importe quelle valeur.

Remarque 3. Les références permettent d'écrire des fonctions récursives sans utiliser le mot clef `rec`. On peut par exemple définir la fonction factoriel par :

```
let fact =
let fac = ref (fun x ->x+1) in  (* extension de l'environnement *)
(fac :=fun n->if n=0 then 1 else (n*(!fac (n-1))));
!fac;;
```

On commence par créer par un `let ...in`, une référence à l'identificateur `fac` qui est donc ajouté à l'environement. Sa valeur initiale est sans importance, elle doit juste être du type attendu pour `fac`, soit `int ->int`. L'identificateur est ajouté à l'environnement. Puis dans cet environnement étendu, on évalue `(fac :=fun n->if n=0 then 1 else (n*(!fac (n-1))))`. Comme `fac` est présent dans l'environnement, le membre gauche est valide mais l'évaluation de `(n*(!fac (n-1)))` est laissée en suspens jusqu'au calcul de `fac 0` qui ne contient pas d'appel recursif et qui initialise `fac` avant de dépiler les calculs en suspens.