Rappel sur les flots en Objective Caml

Laure Gonnord et Jean-François Monin

01 avril 2007

Ce document est à lire en couleurs. Conventions :
– en noir : la ponctuation
– en bleu : la grammaire
– en rouge : le calcul.

On considère des expressions bien parenthésées conformément à la grammaire suivante :

$$S ::= P\,S \mid \varepsilon$$
$$P ::= (\,S\,) \mid x$$

Dans un premier temps écrivons la structure d'un analyseur pour de telles expressions (attention, en tant que code Ocaml, c'est incomplet) :

```
let rec a_S = parser
    | [< '('  ;  a_P  ;  ')'  ;  a_S >] →
    | [< >] →
and a_P = parser
    | [< '('  ;  a_S  ;  ')' >] →
    | [< 'x' >] →
```

Complétons. Si l'on veut rendre une simple indication de succès de l'analyse, on peut se contenter d'un résultat de type `unit` :

```
let rec a_S = parser
    | [< '('  ;  ok1 = a_P  ;  ')'  ;  ok2 = a_S >] → ()
    | [< >] → ()
and a_P = parser
    | [< '('  ;  ok = a_S  ;  ')' >] → ()
    | [< 'x' >] → ()
```

D'une manière générale, les fonctions comme a_P rendent un résultat, par exemple $()$, et ce résultat doit être lié, par exemple à ok_1, lors de l'invocation. Ces liaisons ne sont pas utilisées dans l'exemple précédent, mais elles le sont dans le suivant. Si on veut rendre le nombre total de x, on écrira en effet :

```
let rec a_S = parser
    | [< '('  ;  n1 = a_P  ;  ')'  ;  n2 = a_S >] → n1 + n2
    | [< >] → 0
and a_P = parser
    | [< '('  ;  n = a_S  ;  ')' >] → n
    | [< 'x' >] → 1
```

Les fonctions comme a_P prennent un flot en argument, mais elles peuvent prendre d'autres arguments avant. Ainsi, si l'on veut rendre le nombre de x présents à une profondeur égale à p, on introduit un argument entier p :

```
let rec a_S p = parser
    | [< '('  ;  n1 = a_P (p - 1)  ;  ')'  ;  n2 = a_S p >] → n1 + n2
    | [< >] → 0
and a_P p = parser
    | [< '('  ;  n = a_S (p - 1)  ;  ')' >] → n
    | [< 'x' >] → if p = 0 then 1 else 0
```

Les liaisons effectuées dans un motif de filtrage de flots peuvent être utilisées, à leur droite, à l'intérieur de ce même motif (les flots sont consommés de gauche à droite). On a donc une autre manière de calculer le nombre total de x, dans un style par accumulateur. La version suivante de a_S admet un argument a et elle rend la somme du nombre total de x et de a :

```
let rec a_S a = parser
    | [< '('  ;  a1 = a_P a  ;  ')'  ;  a2 = a_S a1 >] → a2
    | [< >] → a
and a_P a = parser
    | [< '('  ;  a1 = a_S a  ;  ')' >] → a1
    | [< 'x' >] → 1 + a
```

Remarque : les conventions sur les portées sont les mêmes que d'habitude. Voici une autre écriture de l'exemple précédent, où on utilise le masquage pour ne pas risquer l'utilisation d'un accumulateur obsolète :

```
let rec a_S a = parser
    | [< '('  ;  a = a_P a  ;  ')'  ;  a = a_S a >] → a
    | [< >] → a
and a_P a = parser
    | [< '('  ;  a = a_S a  ;  ')' >] → a
    | [< 'x' >] → 1 + a
```