Extension de mini-ML

Ajoutons les expressions suivantes à mini-ML :

- constante booléenne — **true** ou **false**
- opérateurs booléens — `&&` ou `||` ou not
- n-uplet — $e_1, \ldots, e_n$
- conditionnelle — **if** e_0 **then** e_1 **else** e_2
- liaison locale récursive — **let rec** $f = e_1$ **in** e_2

De plus, généralisons la méta-variable x après **let** et **fun** pour en faire des *motifs irréfutables*, que nous notons $\overline{p}$:

- fonction — **fun** $\overline{p} \to e$
- définition locale — **let** [**rec**] $\overline{p} = e_1$ **in** e_2

Les motifs irréfutables

Un *motif irréfutable* $\overline{p}$ est défini *récursivement* par les cas suivants :

- variable f, g, h (fonctions) et x, y, z (autres).
- unité $()$
- n-uplet $\overline{p}_1, \ldots, \overline{p}_n$
- parenthèse $(\overline{p})$
- joker $\_$

Remarques

- Du point de vue syntaxique, les motifs irréfutables sont des cas particuliers d'expressions — hormis le joker.
- Un joker est un cas spécial pour que la phrase ne crée pas de liaison.

Exemples de phrases correctes (suite)

```
let x, y = 5, ('a',());;
let y = f x;;
let z = f x y;;
let _ = 5;;
let _, (_,_) = x, (y,z);;
let _ = (x,(y,z));;
let () = print_string "Hello world !";;
let rec fact = fun n -> if n = 1 then 1 else n *
fact(n-1);;
```

Règles syntaxiques supplémentaires sur les expressions

- La virgule est prioritaire sur la flèche :
  fun $x \to x, y$ équivaut à **fun** $x \to (x, y)$
- Pour alléger la notation **fun** $\overline{p}_1 \to$ **fun** $\overline{p}_2 \to \ldots \to$ **fun** $\overline{p}_n \to e$ on définit les constructions équivalentes
  - **let** [**rec**] $f =$ **fun** $\overline{p}_1\, \overline{p}_2 \ldots \overline{p}_n \to e$;; (nouvelle expression)
  - **let** [**rec**] $f\ \overline{p}_1\, \overline{p}_2 \ldots \overline{p}_n = e$;; (nouvelle phrase)
- Exemple :

```
let f = fun x y -> x + y in
  let g x y = x + y in
  let rec fact n = if n = 1 then 1 else n * fact(n-1)
in f 1 2 - g 3 4;;
```

Extensions de mini-ML (suite)

Nous étendons la syntaxe pour alléger certaines expressions.

Ainsi, par définition

- **let** $\overline{p}_1 = e_1$ **and** $\overline{p}_2 = e_2 \ \ldots$ **and** $\overline{p}_n = e_n$ **in** e;;
  équivaut à
- **let** $\overline{p}_1, \ldots, \overline{p}_n = e_1, \ldots, e_n$ **in** e;;

De même nous introduisons les définitions mutuellement récursives :

- **let rec** $\overline{p}_1 = e_1$ **and** $\overline{p}_2 = e_2 \ \ldots$ **and** $\overline{p}_n = e_n$ **in** e;;

De plus, la phrase e;; équivaut à **let** _ $= e$;;

Les expressions parallèles

Considérons le cas où les motifs irréfutables sont des variables

$$\textbf{let } x = e_1 \textbf{ and } y = e_2 \textbf{ in } e \quad \text{où } x \neq y$$

Si $x \in \mathcal{L}(e_2)$, nous la définissons comme étant équivalente à

$$\begin{array}{l} \textbf{let } z = x \textbf{ in} \\ \quad \textbf{let } x = e_1 \textbf{ in} \\ \quad \textbf{let } y = \textbf{let } x = z \textbf{ in } e_2 \\ \textbf{in} e \end{array}$$

où $z \notin \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \cup \mathcal{L}(e)$, pour n'être capturé ni par e_1, ni par e_2, ni par e.

Ce n'est donc pas une construction élémentaire.

Les expressions mutuellement récursives

Le **let rec** multiple (avec **and**) peut toujours se ramener à un **let rec** simple (avec **in**) en paramétrant l'une des définitions par rapport à l'autre. Posons que

$$\textbf{let rec } x = e_1 \textbf{ and } y = e_2 \textbf{ in } e$$

où $x \neq y$, est équivalent à

$$\begin{array}{l}\textbf{let rec } x = \textbf{fun } y \to e_1 \textbf{ in} \\ \quad \textbf{let rec } y = \textbf{let } x = x\,y \textbf{ in } e_2 \textbf{ in} \\ \quad \textbf{let } x = x\,y \\ \textbf{in } e\end{array}$$

Ce n'est donc pas une construction élémentaire.

D'autres exemples de phrases correctes

```
let x = 5 and y = ();;
let id x = x;;
let rec even n = (n=0) || odd (n-1);;
and odd n = if n = 0 then false else even(n-1);;
let x = 5 and y = 'a' and z = ();;
```

Extensions de la syntaxe des valeurs

L'ajout de nouvelles expressions au langage nous oblige à étendre les valeurs qui sont maintenant définies par

- unité ou constantes : $()$ ou 0 ou **true** etc.
- fermeture : $\langle \textbf{fun}\ x \to e, \rho \rangle$ où ρ est un environnement. Pour les opérateurs : $\langle (+), \rho \rangle$ etc.
- n-uplet : $v_1, \ldots, v_n$

Fonctions curryfiées

Une fonction est dite *curryfiée* (du nom du logicien Curry) si elle retourne une fonction. Cela permet d'effectuer des applications partielles (cf. page 33).

En passant, n'oublions pas qu'une fonction OCaml prend toujours un seul argument.

Si l'on souhaite le passage simultané de plusieurs valeurs il faut alors employer une structure de donnée, par exemple un n-uplet. Ainsi

```
# let add x y = x + y;;
val add : int → int → int
# let add' (x,y) = x + y;;
val add' : int × int → int
```

La fonction `add` est curryfiée et `add'` ne l'est pas.

Les termes ouverts revus

Nous avons présenté une analyse statique qui nous donne les variables libres d'une expression. Nous avons vu qu'une expression close ne peut échouer par absence de liaison. Tous les compilateurs (comme OCaml) rejettent les programmes ouverts (c.-à-d. non-clos), mais, du coup, rejettent d'innocents programmes, comme **if true then** 1 **else** x.

Pour accepter ce type d'exemple (ouvert), il faudrait pouvoir prédire le flot de contrôle (ici, quelle branche de la conditionnelle est empruntée pour toutes les exécutions). Dans le cas ci-dessus cela est trivial, mais en général le problème est indécidable, et ce ne peut donc être une analyse statique (car la compilation doit toujours terminer).