

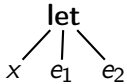


Représentation arborescente des programmes mini-ML

Comme dans n'importe quel langage de programmation, avant d'aborder l'exécution des programmes mini-ML, nous devons préciser la notion de *portée des variables*, c.-à-d. à quoi se réfère une variable donnée. Pour cela, une représentation graphique des programmes (les expressions suffisent) sous forme d'arbres est très commode.

Expression	Arbre
x	x
fun $x \rightarrow e$	fun $\swarrow \searrow$ $x \quad e$
$e_1 \ e_2$	$\$$ $\swarrow \searrow$ $e_1 \ e_2$

Représentation arborescente des programmes mini-ML (suite et fin)

Expression	Arbre
$e_1 + e_2$ etc.	 <pre>graph TD; A["+"] --- B["e1"]; A --- C["e2"];</pre>
0 ou 1 ou 2 etc. (e)	0 ou 1 ou 2 etc.  <pre>graph TD; A["e"];</pre>
let $x = e_1$ in e_2	 <pre>graph TD; A["let"] --- B["x"]; A --- C["e1"]; A --- D["e2"];</pre>

Construction des arbres de programme

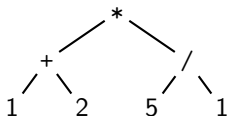
Intuitivement, la méthode générale consiste d'abord à parenthéser complètement l'expression qui fait le programme.

Chaque parenthèse correspond à une sous-expression et chaque sous-expression correspond à un sous-arbre.

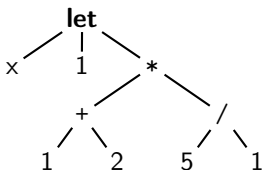
On construit l'arbre des feuilles vers la racine en parcourant les sous-expressions parenthésées les plus imbriquées vers les plus externes.

Exemples d'arbres de programmes

L'expression $(1+2)*(5/1)$ se représente

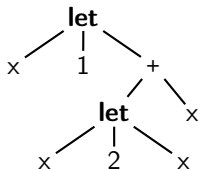


et `let x = 1 in (1+2)*(5/1)` devient (attention : x et x sont différents)



Exemples d'arbres de programmes (suite)

L'expression `let x = 1 in ((let x = 2 in x) + x)` est



Qu'en est-il de `fun y -> x + (fun x -> x) y`? Et de

```
let x = 1 in
  let f = fun y -> x + y in
    let x = 2
  in f(x)
```

Quid du programme page 11?

Liaison statique et environnement

Une phrase associe une expression e à une variable x : on parle de *liaison*, notée $x \mapsto e$. Un sous-programme définit donc un ensemble de liaisons appelé *environnement*.

Une liaison est *statique* si l'on peut déterminer à la compilation (c.-à-d. en examinant le code source) à quelle expression une variable donnée fait référence. Par exemple dans

```
let x = 0 in
  let id = fun x -> x in
    let y = id (x) in
      let x = (fun x -> fun y -> x + y) 1 2
        in x+1;;
```

à quelle expression fait référence x dans $x+1$?

Liaison statique et environnement (suite et fin)

Les liaisons sont ordonnées dans l'environnement *par ordre de définition*.
Ainsi

1. l'environnement est initialement vide : $\{\}$
2. après `let x = 0 in` il vaut $\{x \mapsto 0\}$
3. après `let id = fun x -> x in` il vaut $\{id \mapsto \mathbf{fun} \ x \rightarrow x; x \mapsto 0\}$
4. après `let y = id (x) in` il vaut
 $\{y \mapsto id(x); id \mapsto \mathbf{fun} \ x \rightarrow x; x \mapsto 0\}$
5. après `let x = ...` il vaut
 $\{x \mapsto \dots; y \mapsto id(x); id \mapsto \mathbf{fun} \ x \rightarrow x; x \mapsto 0\}$

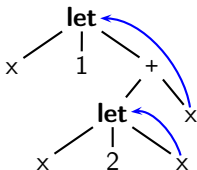
La liaison $x \mapsto 0$ est donc cachée, ou hors de portée, dans $x+1$.

On notera $\rho(x)$ la première liaison de x dans l'environnement ρ (si elle existe).

Variables libres et représentation graphique des liaisons dans une expression

La définition locale **let** $x = e_1$ **in** e_2 lie e_1 à x , noté $x \mapsto e_1$, dans e_2 . Il se peut que dans e_2 une autre définition locale lie la même variable...

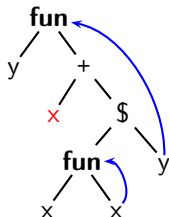
Pour y voir plus clair on applique le procédé suivant sur l'arbre de programme. À partir de chaque occurrence de variable, remontons vers la racine. Si nous trouvons un premier **let** liant cette variable, créons un arc entre son occurrence et ce **let**. Si, à la racine, aucun **let** n'a été trouvé, la variable est dite *libre* dans l'expression. On notera $\mathcal{L}(e)$ l'ensemble des variables libres de e .



Variables libres d'une abstraction

Une situation similaire se pose avec les fonctions **fun** $x \rightarrow e$: dans leur corps e le paramètre x cache une éventuelle variable x liée plus haut dans l'arbre. Il nous faut alors considérer que **fun** est un lieur comme **let**.

Reprenons `fun y -> x + (fun x -> x) y` :



Quid des programmes pages 11 et 17 ?

Expressions closes et évaluation

Une expression *close* est une expression sans variables libres. Seul un programme clos peut être évalué (exécuté). En effet, quel serait la valeur du programme réduit à la simple expression x ?

C'est pourquoi la première analyse statique des compilateurs consiste à déterminer les variables libres des expressions. Si le programme n'est pas clos, il est rejeté. Dans le cas de x , le compilateur OCaml imprimerait

Unbound value x

(c.-à-d. « Valeur x non liée ») et s'arrêterait. L'intérêt est que cette expression non close est rejetée à la compilation et ne provoque donc pas une erreur à l'exécution.