Programmation fonctionnelle en OCaml

Christian Rinderknecht

3 Mars 2003

Objectifs didactiques

Découverte d'un nouveau paradigme de programmation : après les langages de script, la programmation structurée et à objets, présenter la programmation fonctionnelle illustrée par le langage OCaml.

Le temps imparti étant très court, on n'abordera que les traits fonctionnels et distinctifs d'OCaml.

L'approche sera très différente, voire déroutante, par rapport à celles que vous avez déjà pratiquées.

À cause de la nouveauté et du peu de temps, il sera donc essentiel de travailler en dehors des heures de classe.

Domaines d'usage de OCaml

Usage général

Domaines de prédilection
- calcul symbolique : preuves mathématiques, compilation, interprétation, analyses de programmes ;
- prototypage rapide et langages dédiés (*Domain Specific Languages*).

Enseignement et recherche
Classes préparatoires, grandes universités (Europe, USA, Japon).

Industrie CEA, EDF, France Télécom, Simulog.

Gros logiciels Coq, Ensemble, Unison.

Quelques mots-clés

Le langage OCaml est dit :
- *fonctionnel* : les fonctions peuvent être directement passées à d'autre fonctions et être retournées ;
- *à gestion de mémoire automatique* (comme Java) ;
- *fortement typé* : le système de types garanti l'absence d'erreurs à l'exécution dues à des incohérences sur les données ;
- *avec inférence de types statique* : les annotations de types sont facultatives car inférées par le compilateur ;
- *compilé* ou *interactif* (interactif comme une calculette) ;
- *à objets* et *à modules*.

Mini-ML
Commençons pas présenter un sous-ensemble du noyau fonctionnel de OCaml : mini-ML.

Une *phrase* est définie par les cas suivants, où e dénote une expression, x et f sont des variables et **let**et **rec**sont des mots-clés :

- définition globale — e ;;
  let $x = e$;;
- définition globale récursive — **let rec** $f = e$;;

Les phrases sont terminées par « ; ; » et un *programme* est une suite de phrases.

Méta-variables et variables
Lorsque l'on écrit que e est une expression, on veut dire que e désigne une portion de phrase OCaml qu'on classe syntaxiquement (c.-à-d. d'après leur forme) dans les expressions.

On dit que e est une *méta-variable* car, étant un nom c'est une variable, mais cette variable n'existe pas dans le langage décrit (OCaml) : elle existe dans le langage de *description*. Autrement dit, ce n'est pas du OCaml mais une notation pour décrire des fragments de OCaml (éventuellement une infinité).

Ainsi la méta-variable x désigne un ensemble (peut-être infini) de variables OCaml, et il ne faut pas la confondre avec la variable OCaml `x`. De même, la méta-variable e_1 désigne une infinité d'expressions.

Les expressions
Une expression e est définie *récursivement* par les cas suivants :

- variable — f, g, h (fonctions) ; x, y, z (autres).
- fonction (ou abstraction) — **fun** $x \to e$
- appel (ou application) — $e_1\ e_2$
- opérateur arithmétique — `(+)` `(-)` `(/)` `(*)`
- opération arithmétique — e_1 `+` e_2 ou e_1 `-` e_2
  ou e_1 `/` e_2 ou e_1 `*` e_2
- unité ou constante entière — `()` ou `0` ou `1` ou `2` etc.
- parenthèse — (e)
- définition locale — **let** $x = e_1$ **in** e_2

Remarques Ce que nous écrivons joliment $\to$ s'écrit `->` en ASCII.

Un programme correct

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

Remarques

— Les variables doivent débuter par une minuscule.

— La flèche est associative à *droite* : l'expression **fun** $x_1 \to$ **fun** $x_2 \to \ldots \to$ **fun** $x_n \to e$ est équivalente à **fun** $x_1 \to$ (**fun** $x_2 \to$ ($\ldots \to$ (**fun** $x_n \to e$)) $\ldots$)

Un programme correct (suite)

— L'appel de fonction est associatif à *gauche* : l'expression $e_1\, e_2\, e_3\, \ldots\, e_n$ est équivalente à $((\ldots(e_1\, e_2)\, e_3)\ldots)\, e_n)$

— L'application des fonctions est prioritaire par rapport à celle des opérateurs :

`f 3 + 4` équivaut à `(f 3) + 4`

— L'application des opérateurs est prioritaire par rapport à l'abstraction :

`fun x -> x + y` équivaut à `fun x -> (x + y)`

Phrase unique

Un programme, c.-à-d. une suite de définitions globales, peut toujours se réécrire en une seule phrase à l'aide de définitions locales imbriquées.

L'exemple précédent est équivalent à

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

Le symbole `_` désigne une variable dont le nom est unique et différent de toutes les autres. Sans perte de généralité, nous étudierons donc les programmes réduits à une seule expression. Le résultat du programme est l'*évaluation* de `x+1`.

Un autre programme correct

OCaml est dit fonctionnel car ses programmes sont bâtis sur des fonctions qui modélisent les fonctions mathématiques calculables.

Par exemple :

```
let compose = fun f -> fun g -> fun x -> f (g x) in
  let square = fun f -> compose f f in
  let double = fun x -> x + x in
  let quad = square double
in square quad;;
```

Interprétation mathématique

$$\begin{aligned}
\texttt{compose} &\equiv f \mapsto (g \mapsto (x \mapsto f(g(x)))) \equiv f \mapsto (g \mapsto (x \mapsto f \circ g(x))) \\
&\equiv f \mapsto (g \mapsto f \circ g) \quad \text{car} \quad \forall h.(x \mapsto h(x) \equiv h) \\
\texttt{square} &\equiv f \mapsto f \circ f \equiv f \mapsto f^2 \\
\texttt{double} &\equiv x \mapsto x + x \equiv x \mapsto 2x \\
\texttt{quad} &\equiv (f \mapsto f^2)(x \mapsto 2x) \equiv (x \mapsto 2x)^2 \equiv x \mapsto 4x \\
\texttt{square}\,(\texttt{quad}) &\equiv (f \mapsto f^2)(x \mapsto 4x) \equiv (x \mapsto 4x)^2 \equiv x \mapsto 16x
\end{aligned}$$

Donc le résultat du programme (`square quad`) est une fonction.

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 refère 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 $x \to e$	**fun** x e
$e_1\ e_2$	$\$$ e_1 e_2

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

Expression	Arbre
e_1 + e_2 etc.	+ e_1 e_2
0 ou 1 ou 2 etc.	0 ou 1 ou 2 etc.
(e)	e
let $x = e_1$ **in** e_2	**let** x e_1 e_2

Construction des arbres de programme

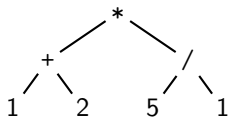
Intuitivement, la méthode générale consiste d'abord à parenthèser complètement l'expression qui fait le programme.

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

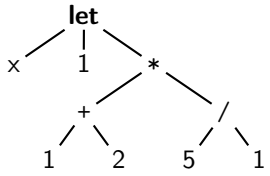
On construit l'arbre des feuilles vers la racine en parcourant les sous-expressions parenthèsées les plus imbriquées vers les plus externes.

Exemples d'arbres de programmes

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

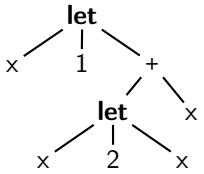


et `let x = 1 in (1+2)*(5/1)` devient (attention : `x` et x sont différents)



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 5 ?

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 $\{\mathsf{x} \mapsto 0\}$
3. après `let id = fun x -> x in` il vaut $\{\mathsf{id} \mapsto \textsf{fun}\ \mathsf{x} \to \mathsf{x}; \mathsf{x} \mapsto 0\}$
4. après `let y = id (x) in` il vaut $\{\mathsf{y} \mapsto \mathsf{id}\,(\mathsf{x}); \mathsf{id} \mapsto \textsf{fun}\ \mathsf{x} \to \mathsf{x}; \mathsf{x} \mapsto 0\}$
5. après `let x = ...` il vaut $\{\mathsf{x} \mapsto \ldots; \mathsf{y} \mapsto \mathsf{id}\,(\mathsf{x}); \mathsf{id} \mapsto \textsf{fun}\ \mathsf{x} \to \mathsf{x}; \mathsf{x} \mapsto 0\}$

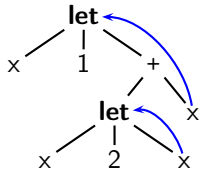
   La liaison $\mathsf{x} \mapsto 0$ est donc cachée, ou hors de portée, dans `x+1`.

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...

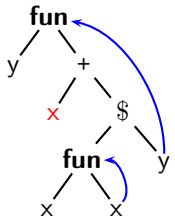
Pour y voir plus clair on applique le procédé suivant sur l'arbre de programme. À partir de chaque occurrence de variable, remontons vers la racine. Si nous trouvons un premier **let** liant cette variable, créons un arc entre son occurrence et ce **let**. Si, à la racine, aucun **let** n'a été trouvé, 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** $x \to e$: 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 5 et 7 ?

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.

Évaluation des programmes mini-ML

L'évaluation d'une expression (donc d'un programme) est une fonction partielle des expressions vers les *valeurs*.

Le fait que la fonction soit partielle modélise le fait qu'une évaluation peut ne pas terminer ou s'interrompre pour cause d'erreur.

Les valeurs v de mini-ML sont presqu'un sous-ensemble strict des expressions défini récursivement par les cas suivants :

- unité ou constante entière — `()` ou `0` ou `1` ou `2` etc.
- fermeture — $\langle \textsf{fun}\ x \to e, \rho \rangle$ où ρ est un environnement. Pour les opérateurs : $\langle (\texttt{+}), \rho \rangle$ etc.

Fermeture, liaison et environnement (le retour)

Une *fermeture* est une sorte de paire faite d'une fonction et d'un environnement. Cela signifie en particulier que les fonctions font partie des valeurs, c.-à-d. qu'elles peuvent être le résultat de l'évaluation d'un programme mini-ML (cf. exemple page 5) : *c'est la caractéristique d'un langage fonctionnel.*

Pour définir l'évaluation nous devons modifier le concept de liaison : une liaison associe maintenant une variable à une *valeur* (et non plus une expression).

L'évaluation (donc peut-être la valeur) dépendra de l'environnement au lieu où se trouve l'expression.

Notons $(x \mapsto v) \oplus \rho$ l'ajout d'une liaison $x \mapsto v$ dans ρ (en tête).

Évaluation des programmes mini-ML (suite)

Pour chaque expression e nous définissons une règle d'évaluation dans un environnement ρ qui donne la valeur v :

- x — Chercher la *première* valeur associée à x dans ρ.
- $\textsf{fun}\ x \to e$ — La valeur est $\langle \textsf{fun}\ x \to e, \rho \rangle$.
- `+ - / *` — La valeur est $\langle (\texttt{+}), \rho \rangle$ etc.
- e_1 `+` e_2 etc. — Évaluer e_1 et e_2 dans ρ et sommer, etc.
- `()` ou `0` ou `1` ou `2` etc. — La valeur est `()` ou `0` ou `1` etc.
- (e) — Évaluer e dans ρ.

Évaluation des programmes mini-ML (suite)

- **let** $x = e_1$ **in** e_2 — Évaluer e_1 en v_1 dans ρ ; évaluer e_2 en v dans $(x \mapsto v_1) \oplus \rho$.
- $e_1\ e_2$ — Évaluer e_1 et e_2 en v_1 et v_2 dans ρ (l'ordre n'est pas spécifié) ; v_1 doit être de la forme $\langle \textbf{fun}\ x \to e, \rho' \rangle$; évaluer e dans $(x \mapsto v_2) \oplus \rho'$: la valeur est v.

Exemple d'évaluation

L'environnement est initialement vide : $\rho = \{\}$. Évaluons

```
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;;
```

— L'évaluation de `let x = 0 in ...` impose d'évaluer d'abord `0` qui vaut 0. Ensuite on crée la liaison $\mathsf{x} \mapsto 0$, on l'ajoute à ρ, ce qui donne $\{\mathsf{x} \mapsto 0\}$, et on évalue la sous-expression élidée dans ce nouvel environnement.

— L'évaluation de `let id = fun x -> x in ...` se fait dans l'environnement $\{\mathsf{x} \mapsto 0\}$. La valeur v est alors la fermeture $\langle \textbf{fun}\ \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\} \rangle$. On étend l'environnement courant avec $\mathsf{id} \mapsto v$ et on évalue la sous-expression avec.

Exemple d'évaluation (suite)

— L'évaluation de `let y = id (x) in ...` se fait dans l'environnement $\{\mathsf{id} \mapsto \langle \textbf{fun}\ \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\} \rangle; \mathsf{x} \mapsto 0\}$.

  — On évalue d'abord `id(x)` dans l'environnement courant. Pour cela on évalue `id` et `x` séparément. Ce sont tous deux des variables, donc on cherche dans l'environnement la première liaison de même nom. La valeur de `id` est donc $\mathsf{id} \mapsto \langle \textbf{fun}\ \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\} \rangle$ et celle de `x` est 0. Il faut évaluer `x` dans l'environnement $(\mathsf{x} \mapsto 0) \oplus \{\mathsf{x} \mapsto 0\}$, ce qui donne 0.

  — On crée la liaison $\mathsf{y} \mapsto 0$, on l'ajoute à l'environnement courant et on évalue la sous-expression élidée avec le nouvel environnement.

— L'évaluation de `let x = (fun x -> fun y -> x + y) 1 2 in ...` se fait dans l'environnement $\{\mathsf{y} \mapsto 0; \mathsf{id} \mapsto \langle \textbf{fun}\ \mathsf{x} \to \mathsf{x}, \{\mathsf{x} \mapsto 0\} \rangle; \mathsf{x} \mapsto 0\}$.

Évaluation formelle des programmes mini-ML

Si on note $[\![e]\!]\rho$ la valeur obtenue en évaluant l'expression e dans l'environnement ρ, alors nous pouvons résumer l'évaluation des programmes mini-ML ainsi :

$$
\begin{aligned}
[\![\overline{n}]\!]\rho &= \dot{n} \quad \text{où } \overline{n} \text{ est un entier mini-ML et } \dot{n} \in \mathbb{N} \\
[\![e_1 + e_2]\!]\rho &= [\![e_1]\!]\rho + [\![e_2]\!]\rho \quad \text{etc.} \\
[\![(e)]\!]\rho &= [\![e]\!]\rho \\
[\![x]\!]\rho &= \rho(x) \quad \text{(la première liaison de } x \text{ dans } \rho) \\
[\![\mathsf{fun}\ x \to e]\!]\rho &= \langle \mathsf{fun}\ x \to e, \rho \rangle \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho &= [\![e_2]\!]((x \mapsto [\![e_1]\!]\rho) \oplus \rho) \\
[\![e_1\ e_2]\!]\rho &= [\![e]\!]((x \mapsto [\![e_2]\!]\rho) \oplus \rho') \\
&\qquad \text{où } [\![e_1]\!]\rho = \langle \mathsf{fun}\ x \to e, \rho' \rangle
\end{aligned}
$$

L'évaluation consiste à appliquer les équations de la gauche vers la droite jusqu'à la terminaison (s'il y a) ou une erreur (à l'exécution).

Exemples

On établit directement que $[\![(\mathsf{fun}\ x \to e_2)\ e_1]\!]\rho = [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho$, c.-à-d. que la liaison locale n'est pas nécessaire en théorie.

Reprenez la description informelle de l'évaluation du programme page 11 et donnez-en une description formelle.

Pour mieux comprendre la nécessité des fermetures, décrivez l'évaluation de

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

et du programme page 5.

Exercices

— Quelle différences y a-t-il entre
  — $f\,x\,y$
  — $(f\,x)\,y$
  — $f\,(x\,y)$
  — $(f\,(x))\,(y)$

— Quelle différences y a-t-il entre
  — f
  — **fun** $x \to f\,x$
  — **fun** $x\,y \to f\,x\,y$

Solution

La seconde expression retarde l'évaluation jusqu'à ce que le premier argument soit fourni.

La dernière retarde l'évaluation jusqu'à ce que les deux arguments soient fournis.

Cela peut délaisser ou dupliquer certains calculs :

```
let f x = let z = print_int x in fun y -> x + y
let test g = let h = g(1) in h(2) + h(3)
# test f;;
1- : int = 7
# test (fun x y -> f x y);;
11- : int = 7
```

Applications partielles et complètes

Si les fonctions sont des valeurs, elles peuvent être retournées en résultat (c.-à-d. être la valeur d'une application). Par exemple :

```
let add = fun x -> fun y -> x + y in
  let incr = add 1                    (* incr est une fonction *)
in incr 5;;
```

On parle d'application *partielle*, par opposition à application *complète*, qui ne retourne pas de fonction, comme (`add 1 5`). Les opérateurs peuvent aussi être utilisés en position préfixe dans une expression en les parenthèsant : `(+) 1 2`. Comme toute application, les opérations peuvent être aussi partiellement évaluées :

```
let incr = (+) 1  (* incr est une fonction *)
in incr 5;;
```

Non-terminaison

En théorie, nous pouvons d'ores et déjà calculer avec mini-ML tout ce qui est calculable avec l'ordinateur sous-jacent. Par exemple, nous avons déjà la récurrence, comme le montre le programme suivant qui ne termine pas :

```
let omega = fun f -> f f in omega omega
```

Cela se manifeste par

$$\begin{aligned}
&[\![\texttt{let omega = fun f -> f f in omega omega}]\!]\rho \\
&\quad = [\![\texttt{omega omega}]\!]((\texttt{omega} \mapsto [\![\texttt{fun f -> f f}]\!]\rho) \oplus \rho) \\
&\quad = [\![\texttt{f f}]\!]((\texttt{f} \mapsto \langle \texttt{fun f -> f f}, \rho\rangle) \oplus \rho) \\
&\quad = \mathit{idem}
\end{aligned}$$

Fonctions récursives

Pour mettre en évidence la puissance de mini-ML, voyons comment définir des fonctions récursives à l'aide de la fonction auto-applicative `omega`.

Définissons d'abord une fonction `fix`, traditionnellement appelée le *combinateur Y de point fixe de Curry* :

```
let omega = fun f -> f f;;
let fix = fun g -> omega (fun h -> fun x -> g (h h) x);;
```

Point fixe d'une fonction

On démontre (péniblement) que pour toute fonction f et variable x :

$$[\![(\mathsf{fix}\ f)\ x]\!]\rho = [\![f\ (\mathsf{fix}\ f)\ x]\!]\rho$$

En d'autre termes, pour tout x on a $(\mathsf{fix}\ f)\ x \equiv f\ (\mathsf{fix}\ f)\ x$, soit $(\mathsf{fix}\ f) \equiv f\ (\mathsf{fix}\ f)$

D'autre part, par définition, le point fixe p d'une fonction f vérifie $p = f(p)$. Donc le point fixe d'une fonction f, *s'il existe*, est $(\mathsf{fix}\ \mathsf{f})$.

Il est possible de définir la sémantique (c.-à-d. l'évaluation) d'une famille d'opérateurs de point fixes (et non d'un seul comme précédemment) en posant qu'un tel opérateur doit satisfaire

$$[\![\mathsf{fix}\ e]\!]\rho = [\![e_1]\!](f \mapsto [\![\mathsf{fix}\,(\mathbf{fun}\ f \to e_1)]\!]\rho \oplus \rho')$$

où $[\![e]\!]\rho = \langle \mathbf{fun}\ f \to e_1, \rho'\rangle$

Factorielle et cas général

```
let pre_fact =
  fun f -> fun n -> if n=1 then 1 else n * f(n-1);;
let fact = fix pre_fact;;
```

Donc `fact` est le point fixe de `pre_fact`, s'il existe, c'est-à-dire

$$\begin{aligned}[\![\texttt{fact}]\!]\rho &= [\![\texttt{pre\_fact (fact)}]\!]\rho \\ &= [\![\texttt{fun n -> if n=1 then 1 else n * fact(n-1)}]\!]\rho\end{aligned}$$

Donc `fact` est la fonction factorielle (équation de récurrence). On peut alors prédéfinir un opérateur de point fixe `fix` (qui n'est pas forcément celui de Curry) et permettre au programmeur de s'en servir directement, mais nous allons plutôt doter mini-ML d'une liaison récursive native :

$$[\![\textsf{let rec } f = e_1 \textsf{ in } e_2]\!]\rho = [\![\textsf{let } f = \mathsf{fix}\,(\textsf{fun } f \to e_1) \textsf{ in } e_2]\!]\rho$$

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

let $x = e_1$ **and** $y = e_2$ **in** e où $x \neq y$

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

let $z = x$ **in**
  let $x = e_1$ **in**
  let $y =$ **let** $x = z$ **in** e_2
in e

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 13).

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).

Les types

Un type t est défini *récursivement* par les cas :

- simple — char, bool, int, string, float, unit
- produit cartésien — $t_1 \times \ldots \times t_n$
- fonctionnel — $t_1 \to t_2$
- parenthésé — (t)
- variable libre — α, β, γ etc.
- type paramétré — α list

Remarque

Jusqu'à présent, nous n'avons pas rencontré de valeurs de type float, char ou string.

Règles syntaxiques sur les types

— Nous notons $\times$ ce qui s'écrit * en ASCII.

— Nous notons α, β etc. ce qui s'écrit respectivement 'a, 'b etc. en ASCII.

— Le produit cartésien est n-aire, et non binaire comme en mathématiques, car $\times$ n'est pas associatif en OCaml : $t_1 \times t_2 \times t_3 \neq (t_1 \times t_2) \times t_3 \neq t_1 \times (t_2 \times t_3)$

— La flèche est utilisée aussi dans les expressions. Elle associe à droite : $t_1 \to t_2 \to \ldots \to t_n$ équivaut à $t_1 \to (t_2 \to (\ldots (t_{n-1} \to t_n)) \ldots)$

— Le produit cartésien est prioritaire sur la flèche : $t_1 \times t_2 \to t_3$ équivaut à $(t_1 \times t_2) \to t_3$

Types, constantes simples et primitives

Les compilateur associe un type à chaque expression du programme : on parle d'*inférence de types statique*. Pour les constantes simples, nous avons

unit	()	
bool	**true false**	&& || not
int	1 2 max_int etc.	+ - * / etc.
float	1.0 2. 1e4 etc.	+. -. *. /. cos etc.
char	'a' '\n' '\097' etc.	Char.code Char.chr etc.
string	"a\tb\010c\n" etc.	^ s.[i] s.[i] ← c etc.

Les opérations sur les flottants sont notées différemment de leurs homologues sur les entiers. Ce que nous notons joliment ← s'écrit <- en ASCII.

L'évaluation des opérateurs booléens

— Les opérateurs booléens sont *séquentiels*, c.-à-d. qu'ils n'évaluent leurs arguments que si c'est nécessaire, l'évaluation se faisant de la gauche vers la droite.

Extension de la syntaxe des types et des phrases

On étend la syntaxe des phrases pour permettre de lier un type à un nom, comme on peut le faire pour les expressions.

- liaison de type (ou alias) type $q = t$;;

où q dénote une variable de type (commençant par une minuscule).

- types récursifs type $q_1 = t_1$ [**and** $q_2 = t_2$ ...];;

Pour utiliser ces variables, il faut étendre la syntaxe des types :

- variable q

On peut maintenant écrire

```
type abscisse = float;;
type ordonnée = float;;
type point = abscisse * ordonnée;;
```

Inférence de types

Les n-uplets sont homogènes et leur arité est fixée par leur type :

une paire	(1,2)	de type	int × int
et un triplet	(1,2,3)	de type	int × int × int

sont incompatibles.

```
# let milieu x y = (x+y)/2;;
val milieu : int → int → int
# let milieu (x,y) = (x+y)/2;;
val milieu : int × int → int
```

Polymorphisme

n-uplets

Les projections sont polymorphes sur les n-uplets de *même arité* :

$$\textbf{fun}(x, y, z) \to x \quad \text{a pour type} \quad (\alpha \times \beta \times \gamma) \to \alpha$$

Fonction puissance

```
# let rec power f n =
    if n <= 0 then fun x -> x
              else compose f (power f (n-1));;
val power : (α → α) → int → (α → α) = ⟨fun⟩
```

Polymorphisme (suite)

```
# let compose f g = fun x -> f (g (x));;
```
val compose : $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta = \langle fun \rangle$

Le type de la fonction compose se construit ainsi :
- le premier argument f est une fonction quelconque, donc de type $\alpha \to \beta$;
- le second argument g est une fonction dont le résultat doit être passé en argument à f, donc de type α ;
- le domaine de g est quelconque, donc g est de type $\gamma \to \alpha$;
- la fonction compose prend un argument x qui doit être passé à g, donc du type γ ; finalement, le résultat de compose est retouné par f, donc de type β.

Égalité structurelle

L'opérateur d'égalité est polymorphe et ne peut être défini en OCaml :

```
# ( = );;
```
- : $\alpha \to \alpha \to$ *bool* $= \langle fun \rangle$

Donc attention à ce qu'il coïncide avec *votre* notion d'égalité.

C'est l'égalité mathématique : deux valeurs sont égales si elles ont la même structure et si leurs parties respectives sont égales. Ne marche pas avec les expressions fonctionnelles (problème indécidable).

```
# 1 = 1 && "oui" = "oui";;
- : bool = true
# (fun x -> x) = (fun x -> x);;
Exception: Invalid_argument "equal: functional value".
```

On note <> la négation de l'égalité.

Le filtre à motifs

Étendons encore les expressions de mini-ML par les *filtres* et les *motifs* :

— **match** e **with** $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$

— où les p_i sont des motifs.

Les motifs sont définis récursivement par les cas suivants :

• variable	f, g, h (fonctions) et x, y, z (autres).
• unité ou constante	() ou 0 ou **true** etc.
• n-uplet	$p_1, \ldots, p_n$
• parenthèse	(p)
• joker	_

Remarque Les motifs irréfutables sont des motifs particuliers.

Le filtre à motifs (suite)

Typiquement, on se sert des filtres pour faire des définitions par cas de fonctions (mais pas uniquement), comme en mathématiques. Par exemple :

```
let rec fib n =
  match n with
    0 -> 1
  | 1 -> 1
  | _ -> fib(n-1) + fib(n-2)
```

Comme en mathématiques, les cas sont ordonnés par l'écriture et la définition précédente se lit donc : « Si la valeur de `n` *a la forme* de `0` alors `fib(n)` est la valeur de `1`, si la valeur de `n` *a la forme* de `1` alors `fib(n)` est la valeur de `1`, sinon `fib(n)` vaut la valeur de `fib(n-1) + fib(n-2)` ».

Le filtre à motifs (suite)

Que signifie la relation « v a la forme de p » (ou « p filtre v ») ?

— Une constante a la forme de la constante identique dans un motif.

— L'unité () a la forme de () dans un motif.

— Un n-uplet a la forme d'un n-uplet dans un motif.

— **Toute valeur a la forme d'une variable de motif ou du joker « _ ».**

Remarques

— Les motifs ne filtrent aucune fermeture, c.-à-d. que e dans **match** e **with** ne doit pas s'évaluer en une fermeture.

— Dans le cas des constantes et (), la relation se confond avec l'égalité.

Sémantique du filtrage

Le *filtrage* est l'évaluation d'un filtre. Informellement, l'évaluation de

$$\textsf{match } e \textsf{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

commence par celle de e en v.

Ensuite v est confrontée aux différents motifs p_i dans l'ordre d'écriture. Si p_i est le premier motif à filtrer v, alors la valeur du filtre est la valeur de e_i.

Par exemple, voici la définition curryfiée de la disjonction logique :

```
let or = fun (x,y) ->
  match (x,y) with
   (false, false) -> false
  | _ -> true
```

Les types sommes et le filtrage des leurs valeurs

Les définitions de types sommes

Les types sommes s'apparentent aux énumérations et aux union du langage C. Par exemple, les valeurs booléennes peuvent être définies ainsi :

```
type booléen = Vrai | Faux
let v = Vrai and f = Faux
```

Les constructeurs ont pour première lettre une majuscule.

Le filtrage permet d'examiner les valeurs d'un type somme :

```
let int_of_booléen = fun b ->
  match b with
     Vrai -> 1
   | Faux -> 0
```

Les types sommes et le filtrage de leurs valeurs (suite)

Les constructeurs peuvent aussi transporter de l'information, dont l'interprétation complète alors celle du constructeur lui-même. Par exemple, un jeu de cartes peut être défini par

```
type carte = Carte of ordinaire | Joker
and ordinaire = couleur * figure
and couleur = Coeur | Carreau | Pique | Trefle
and figure = As | Roi | Reine | Valet | Simple of int
```

Le jeu de cartes

Définissons des cartes et des fonction construisant des cartes :

```
# let valet_de_pique = Carte (Pique,Valet);;
val valet_de_pique : carte = Carte (Pique,Valet)
# let carte f c = Carte (c,f);;
val carte : figure → couleur → carte = ⟨fun⟩
# let roi = carte Roi;;
val roi : couleur → carte = ⟨fun⟩
```

Filtrage des cartes

```
let valeur c = match c with
  Carte (As)       -> 14
| Carte (Roi)      -> 13
| Carte (Reine)    -> 12
| Carte (Valet)    -> 11
| Carte (Simple k) -> k
| Joker            -> 0
```

Un motif peut capturer plusieurs cas :

```
let est_petite c = match c with
  Carte (Simple _) -> true
| _ -> false
```

Filtres incomplets

Rappel Lorsqu'une valeur v est filtrée par $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$, l'expression e_i associée au premier motif p_i qui filtre v est évaluée dans l'environnement courant étendu par les liaisons éventuelles créées par le filtrage de v par p_i.

Le filtre est *incomplet* s'il existe au moins une valeur qui n'est filtrée par aucun motif. Dans ce cas, un message d'avertissement est indiqué à la compilation :

```
# let simple c = match c with Carte (_,Simple k) -> k;;
Characters 15-51
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: Joker
val simple : carte → int = ⟨fun⟩
```

Il est préférable d'éviter les définitions incomplètes.

Les motifs non linéaires

Une variable ne peut être liée deux fois dans le même motif.

```
# fun (x,y) -> match (x,y) with (Carte z, z) -> true;;
Characters 41-42
This variable is bound several times in this matching.
```

Un tel motif est dit *non linéaire*.

Listes

Les listes peuvent être définies comme un type variant polymorphe récursif :

```
type 'a liste = Nil | Cons of 'a * 'a liste
```

Les noms de ces constructeurs sont traditionnels dans la communauté des langages fonctionnels. Le premier, `Nil`, dénote la liste vide ; le second, `Cons`, dénote la liste non vide. Une liste non vide est alors modélisée par une paire dont la première projection est un élément de la liste (de type `'a`) et la seconde la sous-liste restante (donc de type `'a liste`). Par exemple :

```
let liste_vide= Nil
let liste_singleton = Cons ('a', Nil)
let liste_singleton_bis = Cons (7, Nil)
let liste_longue = Cons (1, Cons (2, Cons (3, Cons (4, Nil))))
```

Les listes prédéfinies et la biliothèque List

Par défaut, le système prédéfinit un type `'a list`, dont le constructeur de liste vide est `[]`, et celui de liste non vide est `::` (utilisé en position infixe). La fonction de concaténation de deux listes est notée `@` (**Ce n'est pas un constructeur.**).

Exemples de formes équivalentes :

```
let l = 1 :: (2 :: 3 :: (4 :: []))
let l = 1 :: 2 :: 3 :: 4 :: []
let l = [1;2;3;4]
```

La biliothèque List fournit des fonctions sur les listes.

Les listes prédéfinies

Exemple Une fonction qui retourne une liste.

```
let rec reverse = function
  [] -> []
| h::l -> (reverse l) @ [h]
```

ou, plus efficacement :

```
let reverse l =
  let rec reverse_aux acc = function
    [] -> acc
  | h::t -> reverse_aux (h::acc) t
in reverse_aux [] l
```

Exceptions

Les exceptions de ML ont servi de modèle pour celles du langage C++.

```
exception Perdu

let rec cherche_la_clé k = function
  (h,v)::t -> if h = k then v else cherche_la_clé k t
| [] -> raise Perdu

let k =
  try
    cherche_la_clé "Louis" [("Georges",14); ("Louis",5)]
  with Perdu -> 10
```

Exercice 5 Réécrire le programme sans user d'exceptions.

Exceptions (suite)

Syntaxe

Définition (phrase)	`exception` C `[of` t`];;`
Lancement (expression)	`raise` e`;;`
Filtrage (expression)	`try` e `with` $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$`;;`

Remarquez l'analogie avec le filtrage des valeurs.

Typage

Les exceptions sont toutes de type `exn`, qui peut être considéré comme un type somme ouvert (de nouveaux constructeurs peuvent être ajoutés avec des déclarations `exception`).

Exceptions prédéfinies

The Core Library

Constructeur	Usage
`Invalid_argument of string`	Argument hors bornes
`Failure of string`	Fonction indéfinie pour un argument
`Not_found`	Échec de fonctions de recherche
`Match_failure of ...`	Échec de filtrage
`End_of_file`	Fin de fichier

Sémantique des exceptions

- Le type exn est le seul type somme *extensible*.
- Le lancement d'une exception arrête l'évaluation et retourne une valeur exceptionnelle (c-à-d. de type exn).
- Une exception ne peut être éventuellement filtrée que si l'expression a été encadrée par un bloc try e with m :
  - Si l'évaluation de e retourne une valeur normale, celle-ci est retournée sans passer par le filtre m.
  - Sinon, l'exception est passée au filtre m. Si un des motifs p_i filtre l'exception, alors e_i est évaluée, sinon l'exception est propagée (**Les filtres d'exceptions ne sont pas forcément complets.**).
- On peut observer une exception (c-à-d. la filtrer puis la relancer) : `try f x with Failure s as x -> prerr_string s; raise x`