Programmation Fonctionnelle

Emmanuel Chailloux

22 juin 2020

Informations sur le cours Programmation Fonctionnelle

Site

`http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2019/ue/LU2IN019-2019oct/`
`https://www-apr.lip6.fr/~chaillou/Public/enseignement/2019-2020/pf/`

Équipe pédagogique :

Emmanuel Chailloux, **Guillaume Hivert**

Emails

`Emmanuel.Chailloux@lip6.fr`,`hivert.is.coming@gmail.com`

Principaux points abordés dans l'UE

- expressions, expressions fonctionnelles avec leur type, définitions, définitions locales, filtrage, n-uplets
- définitions récursive, exceptions (pour les fonctions partielles) récurrence terminale, itérateur (fonctionnelle)
- listes, filtrage, récurrence structurelle
- itérateurs : fonctionnelles avancées pour les listes
- type somme pour les arbres binaires
- exception comme structure de contrôle
- arbres généraux et récurrences mutuelles
- généricité, polymorphisme, modules

Choix d'un langage fonctionnel support :

- typés dynamiquement
  - Lisp, Scheme (utilisé en L1 de 2004-2016)
  - JavaScript (prrogrammation client Web)
- typés statiquement
  - OCaml (Inria) : fonctionnel, impératif, objet, modules paramétrés, typage fort, environnement Emacs ou Eclipse
  - Haskell (Glasgow) : fonctionnel, évaluation paresseuse, typage fort, surcharge, environnement Emacs ou Eclipse
  - F# (Microsoft) : fonctionnel et impératif à la OCaml, objet à la C#, environnement Visual Studio, syntaxe à la OCaml
  - Java (Oracle) : objet et fonctionnel (depuis 1.8), peu modulaire
  - Scala (EPFL) : objet et fonctionnel, mixin pour la modularité
  - Swift (Apple) : fonctionnel et impératif à la OCaml, objets avec classes et structures, environnement X-code
  - Reason (Facebook) : nouvelle syntaxe pour OCaml
  - Kotlin (JetBrains) : fonctionnel et objets, typé statiquement, pour le développement Android

$\Rightarrow$ choix d'**OCaml** réduit à la partie fonctionnelle pure, avec exceptions, un peu d'entrées/sorties, et modules simples

Plan du cours

1. présentation de l'UE
   généralités sur les langages fonctionnels
   boucle d'interaction ou compilation en ligne de commande
   langages d'expressions, types de base, fonction et définition par cas déclarations globales et locales, fonctions récursives
2. structures linéaires (listes)
   définition de types : type somme, type produit
   filtrage par motifs, fonctions totales et partielles,
   exceptions : déclenchement, récupération et style de programmation - type optionnel, valeur par défaut
3. fonctionnelles, polymorphisme paramétrique
   récursivité terminale, itérateurs
   autres structures linéaires (files),
   modules simples, compilation séparée
4. appels récursifs, autres structures : arbres binaires, de syntaxe abstraite, de recherche,itérateurs sur ces structures
5. arbres généraux, lexicaux, représentation mémoire, E/S

Logiciels du cours

- Langage OCaml 4.xx (Inria)
  - pré-installé à la PPTI
  - à installer à la maison à partir du site `http://ocaml.org`
  - dans un navigateur avec
    - Learn-OCaml (environnement des TME)
    - TryOCaml : `http://try.ocamlpro.com/`

- Environnements de développement
  - Emacs et mode Tuareg : `http://www.gnu.org/software/emacs/` et `http://tuareg.forge.ocamlcore.org/`
  - Emacs et Merlin : `https://github.com/ocaml/merlin`
  - Eclipse et plug-in Ocaide : `http//www.eclipse.org` et `http://www.algo-prog.info/ocaide/`
  - VisualStudioCode : `https://code.visualstudio.com`
  - Atom script : `https://atom.io/packages/script` décrit au premier TME
  - Xamarin : `https://xamarin.com/studio`
  - Netbean : `http://ocamlplugin.loki-a.com/index.php`

Bibliographie (1)

- OCaml
  - Pascal Manoury. Programmation de droite à gauche, et vice-versa. Paracamplus, 2012.
  - Xavier Leroy et al. The OCaml system : documentation and user's manual Inria, 2016.
  - Emmanuel Chailloux, Pascal Manoury et Bruno Pagano. Développement d'Applications avec Objective Caml. O'Reilly, 2000. en ligne.
  - Guy Cousineau et Michel Mauny. Approche fonctionnelle de la programmation. Dunod, 1995.
  - Philippe Nardel. Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml. Vuibert, 2005
  - Sylvain Conchon et Jean-Christophe Filliâtre. Apprendre à programmer avec OCaml. Eyrolles, 2013. item Yaron Minsky, Anil Madhavapeddy, Jason Hickey - Real World OCaml - O'Reilly - 2013

autres références sur :

`https://ocaml.org/learn/books.html`,

`http://ocaml.org/docs/cheat_sheets.html`

Bibliographie (2)

- programmation fonctionnelle
  - Harold Abelson, G. Sussman, J. Sussman - Structures et interprétation des programmes informatiques - InterEditions, 1997
  - Chris Okasaki - Purely functional data structures, Cambridge University Press, 1998,
- autres langages proches :
  - Michael Hansen, Hans Rischel - Functional Programming usins F# - Cambridge University Press - 2013
  - Don Syme, Adam Granicz, Antonio Cisternino - Expert F# 4.0 - Apress - 2015
  - Apple Inc - The Swift Programming Language (Swift 5.1) - on line

autres références sur :

`https://en.wikipedia.org/wiki/Functional_programming`

`http://fsharp.org/about/learning.html`

Evaluation

- pas d'évaluation, cours de mise à niveau
- note Bonus sur les exercices réalisés , la participation et le projet

OCaml (ou REason ou F# ou Swift)

- langage fonctionnel,
- typé statiquement,
- polymorphe paramétrique,
- avec inférence de types,
- muni d'un mécanisme d'exceptions,
- et de traits impératifs
- possèdant un système de modules paramétrés (OCaml)
- et un modèle objet
- exécutant des processus légers
- et communiquant sur le réseau Internet,
- indépendant de l'architecture machine.

Historique (1)

- l'ancêtre : ML (*meta-langage*) de LCF (80)
- machines abstraites (84)
  - la FAM : Cardelli
  - la CAM : Curien, Cousineau
- spécifications : Standard ML (84 - Milner)
- premières implantations
  - CAML - Suarez - Weis - Mauny (87)
  - SML/NJ - Mc Queen - Appel (ATT - 88)
- nouvelles implantations : Caml-light (90) Leroy - Doligez
- modules paramétrés : Caml Special Light (95)
- extension objet : OCaml (96)
- labels, options, variants polymorphes, modules récursifs, de 1ère classe,...
- types algébriques généralisés (GADT)
- types extensibles, récupération d'exceptions du filtrage

Historique (2)

- de F# 1.0 (2005) à F# 4.1 (2017) :
  - noyau fonctionnel/impératif + modules simples d'OCaml, objets à la C#
  - langage d'expressions
  - mode compatible pour le noyau OCaml (ML compatibility mode)
  - extensions sur la concurrence (programmation asynchrone), typage (unités de mesure), quotation, ...
  - sous Windows et Mono (Linux, MacOSX)
- de Swift 1.0 (2014), 2.1 (2015) avec passage en open Source, 3.0 (2016) uniformisation du langage
  - type optionnel, valeur par défaut (nil), déclaration par valeur et par variable,
  - langage d'instructions
  - sous MacOSX, Linux (Windows ?)
- de REason (2015-), couche syntaxique d'OCaml, interface avec JS

OCaml : mise en œuvre (1)

- compilateur natif (commande **ocamlopt**)
  - pour Intel, ARM, Sparc, HP-pa, PowerPC, ...
  - pour Linux, Macosx, Windows, ...

```
1  $ cat t.ml
2  let f x = x + 1 ;;
3  let main() = print_string "f(18) = "; print_int (f(18)); print_newline() ;;
4
5  main ();;
6
7  $ ocamlopt -o tn.exe t.ml
8
9  $ ./tn.exe
10 f(18) = 19
```

- fin de déclaration : `;;`
- séquence : `;`
- fonctions de sorties : `print_string` et `print_int`
- fonctions d'entrées :
  - `read_line : unit -> string`
  - `read_int : unit -> int`

OCaml : mise en œuvre (2)

- compilateur byte-code (commande **ocamlc**)
  - ligne de commande

```
1  $ cat t2.ml
2  let f x = x + 1 ;;
3  let main() =
4   let inv = read_int() in
5    print_string "f("; print_int inv; print_string ") = ";
6    print_int (f(inv)); print_newline() ;;
7
8  main ();;
9
10 $ ocamlc -i -custom -o tb.exe t2.ml
11 val f : int -> int
12 val main : unit -> unit
13
14 $ ./tb.exe
15 2019
16 f(2019) = 2020
```

OCaml : mise en œuvre (3)

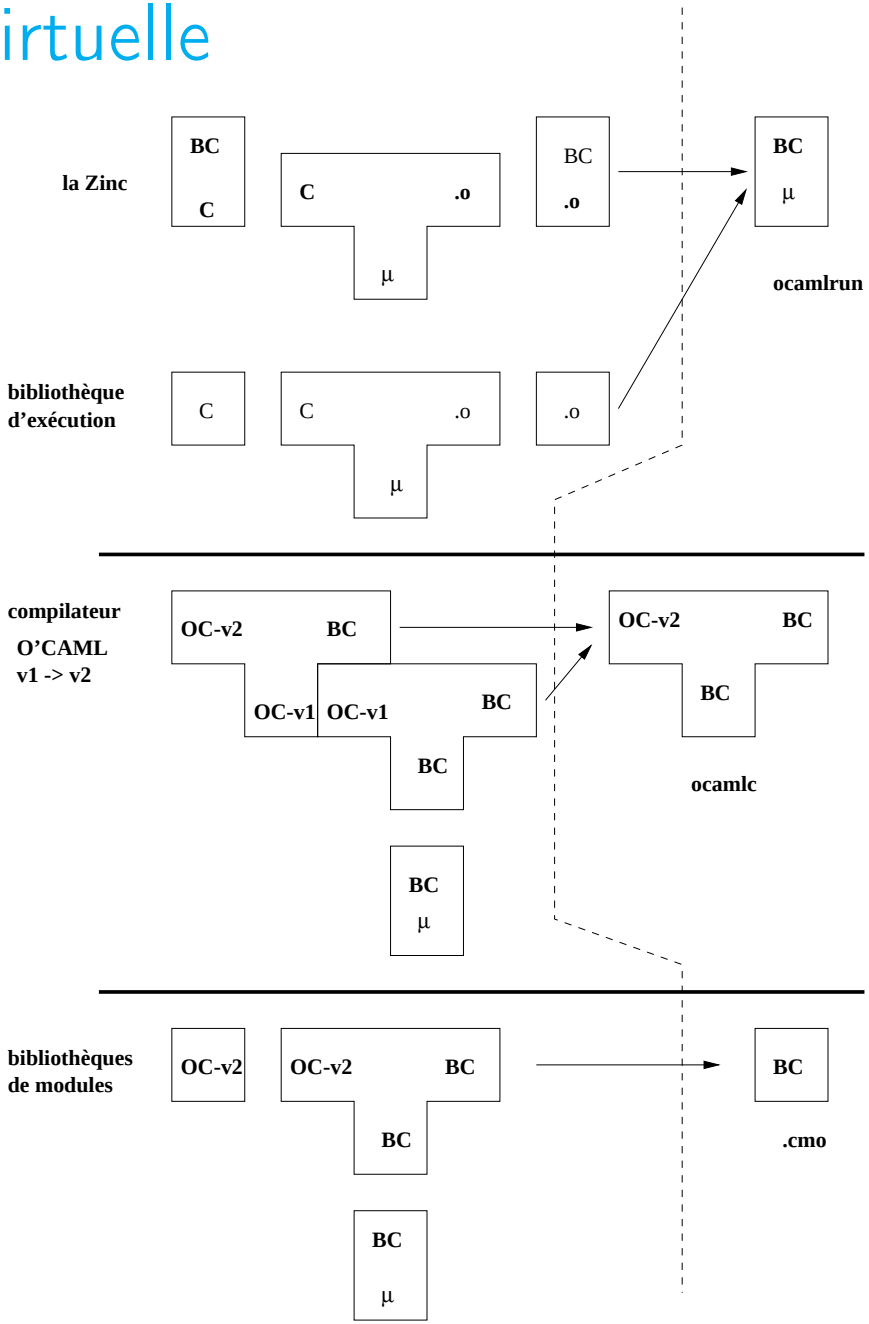
- compilateur byte-code (commande **ocaml**)
  - boucle d'interaction

```
$ ocaml
        OCaml version 4.07.1

# let f x = x + 1 ;;
val f : int -> int = <fun>
# let main() = print_string "f(18) = "; print_int (f(18)); print_newline() ;;
val main : unit -> unit = <fun>
# main ();;
f(18) = 19
- : unit = ()
#  #quit ;;
$
```

- $: invite Unix
- # : invite OCaml
- #quit : directive OCaml

Machine virtuelle



Phrases du langage

Se terminent par `;;` au *toplevel*

- expressions
- déclarations
  - de valeurs
  - de types
  - d'exceptions

- déclarations de modules
- déclarations de classes

Notation qualifiée `Module.nom` pour accéder à un champs d'un module.

F# et Swift : mise en œuvre

- F# : compilateur vers .NET (`fshaprc`), interopérabilité avec les langages compilant vers .NET (comme C#), JIT, REPL :

```
$  fsharpi
F# Interactive for F# 4.1 (Open Source Edition)

> let f x = x + 1  ;;
val f : x:int -> int
> f 18 ;;
val it : int = 19
```

- Swift : compilateur vers LLVM (swiftc), interopérabilité avec Objective C, JIT, REPL :

```
Welcome to Apple Swift version 5.1.3 (swiftlang-1100.0.282.1 clang↩
     -1100.0.33.15).
Type :help for assistance.
  1> func f(x:Int) -> Int {return x + 1}
  2> f(x:18)
$R0: Int = 19
```

Noyau fonctionnel

Programmes en OCaml (ou F#)

Seuls les programmes **correctement typés** peuvent être exécutés ! ! !

un programme est :

- le calcul d'une expression
- c'est-à-dire l'application d'une fonction à ses arguments
- avec évaluation immédiate des arguments
- et rupture d'évaluation si calcul impossible

Types de base, valeurs et fonctions

- nombres
  - *int* ($[-2^{30}, 2^{30}-1]$ sur 32 bits ou $[-2^{62}, 2^{62}-1$ sur 64 bits) voir `min_int` et `max_int`
  - *float* (IEEE 754) mantisse 53bits, exposant $\in [-1022, 1023]$
- caractères : *char*
- chaînes de caractères : *string*
- booléens : *bool*

Opérations sur les nombres

entiers		flottants	
`+`	addition	`+.`	addition
`-`	soustraction	`-.`	soustraction
`*`	multiplication	`*.`	multiplication
`/`	division entière	`/.`	division
`mod`	modulo	`**`	exponentiation

opérateurs infixes ! ! !

```
1 # 1 + 3;;
2 - : int = 4
3 # 1.8 *. 9.1;;
4 - : float = 16.38
```

Fonctions sur les nombres

`ceil`		`cos`	cosinus
`floor`		`sin`	sinus
`sqrt`	racine carrée	`tan`	tangente
`exp`	exponentielle	`acos`	arccosinus
`log`	log népérien	`asin`	arcsinus
`log10`	log en base 10	`atan`	arctangente

angles en radiant ! ! !

Calculs sur les nombres (1)

```
1  # 1 + 2;;
2  - : int = 3
3
4  # 9/2;;
5  - : int = 4
6
7  # max_int + 1;;
8  - : int = -4611686018427387904
9
10 # 9.1 /. 2.2;;
11 - : float = 4.13636363636
12
13 # 1. /. 0.;;
14 - : float =          Inf
```

Calculs sur les nombres (2)

```
1  # 2 + 2.1;;
2  This expression has type float but
3  is here used with type int
```

Un *int* ne peut pas remplacer un *float* ! ! !

```
1  # sin;;
2  - : float -> float = <fun>
3
4  # asin 1.;;
5  - : float = 1.57079632679
```

Calculs sur les chaînes

```
1  # 'B';;
2  - : char = 'B'
3
4  # int_of_char 'B';;
5  - : int = 66
6
7  # "est une chaine";;
8  - : string = "est une chaine"
9
10 # (string_of_int 1987)^" est l'annee de CAML";;
11 - : string = "1987 est l'annee de CAML"
```

Opérations sur les booléns

Type des constantes et opérateurs:

```
true  : bool
false : bool
not   : bool -> bool
&&    : bool -> bool -> bool
||    : bool -> bool -> bool
```

si *e*1 et *e*2 sont des expressions booléennes alors `true`, `false`, `not e1`), `(e1 || e2)`, `(e1 && e2)` sont des expressions booléennes et réciproquement.

```
1  # true && false ;;
2  - : bool = false
3  # not (true && false) ;;
4  - : bool = true
```

Relations d'égalité et d'inégalité

=	égalité structurelle	<	inférieur
==	égalité physique	>	supérieur
<>	négation de =	<=	inférieur ou égal
!=	négation de ==	>=	supérieur ou égal

égalités et inégalités sont génériques (polymorphes)
elles s'appliquent à des arguments de n'importe quel type (les deux arguments doivent être du même type) ! ! !

```
= : 'a -> 'a -> bool
== : 'a -> 'a -> bool
```

```
1 # 3 == (1 + 2);;
2   - : bool = true
3 # not(true) = false ;;
4 - : bool = true
```

Produits cartésiens, n-uplets

- constructeur de valeurs : `,`
- constructeur de types : `*`
- accesseurs (couples) : `fst` et `snd`

```
# (28, "janvier");;
- : int * string
# (20, "janvier", 2020);;
- : int * string * int
# fst (20, "janvier");;
- : int = 20
# snd (20, "janvier", 2020);;
```

erreur de typage sur la dernière ligne : la fonction `snd` est appliquée à un triplet et non pas à un couple.

Listes homogènes

- liste vide : `[]`
- constructeur `::`
- type *list*
- accesseurs `List.hd` et `List.tl`

```
# [] ;;
- : 'a list
# 1::2::3::[] ;;
- : int list
# [1; 2; 3;] ;;
- : int list
# [1; "hello"; 3] ;;
erreur de typage
# List.hd [1.1; 1.2; 1.3] ;;
- : float = 1.1
# List.hd [] ;;
Exception: Failure "hd".
```

voir cours 2.

Expression conditionnelle

Syntaxe:

if expr1 **then** expr2 **else** expr3

- *expr1* de type *bool*
- *expr2* et *expr3* de même type

```
1  # if 3 = 4 then 0 else 4;;
2  - : int = 4
3  # if 5 = 6 then 1 else "Non";;
4  - : erreur de typage
5  # (if 3 = 5 then 8 else 10) + 5;;
6  - : int = 15
7  # 5 = 6 || 7 = 9;;
8  - : bool = false
```

Déclarations de valeurs (1)

Déclarations globales:

Syntaxe:

let p = e

```
1  # let pi = 3.14159;;
2  val pi : float = 3.14159
3
4  # sin (pi /. 2.0);;
5  - : float = 0.999999999999
```

Déclarations de valeurs (2)

Déclatations locales:

Syntaxe:

let p = e1 **in** e2

```
# let x = 3 in
    let b = x < 10 in
      if b then 0 else 10;;
- : int = 0

# b;;
Unbound value b
```

Déclarations combinées

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$ ... **and** $p_n = e_n$;;

```
# let x = 1 and y = 2;;
val x : int = 1
val y : int = 2
# x + y;;
- : int = 3
```

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$ ... **and** $p_n = e_n$ **in** expr ;;

```
# let a = 3.0 and b = 4.0 in sqrt(a*.a+.b*.b);;
- : float = 5
```

Valeurs fonctionnelles, fonctions

Syntaxe:

function p -> e

- fonction anonyme : **function** p -> e
- de type : typede(p) -> typede(e)

```
# function x -> x + 1;;
 - : int -> int
# (function x -> x + 1) 2019 ;;
- : int = 2020
# function x -> if x < 0 then -x else x;;
- : int -> int
# function x -> (function y -> 2*x + 3*y);;
- : int -> int -> int
```

Déclaration de valeurs fonctionnelles

```
# let succ = function x -> x + 1;;
succ : int -> int
# succ 420;;
- : int = 421
```

Syntaxe:

let f = **function** p -> e

ou **let** f p = e *ou* **let** f (p) = e

```
# let pred(x) = x -1;;
pred : int -> int = <fun>
# let f c = 2*(fst c) + 3*(snd c);;
f : int * int -> int = <fun>
# f(1,2);;
- : int = 8
# let g = function x -> (function y -> 2*x + 3*y);;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
# let h = g 1 ;;
val h : int -> int = <fun>
# h 2 ;;
- : int = 8
```

Explicitation des types des paramètres des fonctions

Coercicion de types:

Syntaxe:

(e : t)
let p : t = e

```
1  # let u = [];;
2  val u : 'a list = []
3  # let v = (u : int list);;
4  val v : int list = []
```

C'est principalement utilisé pour les fonctions, on peut écrire, et on écrira

```
1  let xor (x:bool) (y:bool) : bool = (x || y) && (not (x && y))
```

indiquant ainsi que le paramètre x est de type `bool`, le paramètre y de type `bool`, et la fonction retourne un booléen.

Portée des déclarations

- portée lexicale (liaison statique)

```
1  # let p = q + 1;;                    (* erreur q inconnue *)
2
3  # let p = p + 1;;                    (* erreur p inconnue *)
4
5  # let p = 10;;                        p : int = 10
6
7  # let addp x = x + p;;                addp : int ->int = <fun>
8
9  # addp 8;;                           - : int = 18
10
11 # let p = 40;;                        p : int = 40
12
13 # addp 8;;                           - : int = 18
14
15 # p;;                                - : int = 40
```

Appels de fonction

```
1  let fois (a, b) = a * b ;;
2  let moins (a, b) = a - b ;;
3  let carre x = fois (x, x) ;;
4  let delta (a, b, c) = moins (carre b, fois (4, fois (a, c))) ;;
```

empilement dans le cadre d'appel des arguments et des adresses de retour pour chaque appel de fonction

```
1  #  #trace moins ;;
2  moins is now traced.
3  #  #trace fois ;;
4  fois is now traced.
5  # #trace carre ;;
6  carre is now traced.
7  #  #trace delta;;
8  delta is now traced.
```

dépilement des arguments empilés au retour de la fonction

```
1  # delta (2,5,2) ;;
2  delta <-- (2, 5, 2)
3  fois <-- (2, 2)
4  fois --> 4
5  fois <-- (4, 4)
6  fois --> 16
7  carre <-- 5
8  fois <-- (5, 5)
9  fois --> 25
10 carre --> 25
11 moins <-- (25, 16)
12 moins --> 9
13 delta --> 9
14 - : int = 9
```

Curryfication

Passage d'une fonction à plusieurs paramètres en une fonction à un paramètre qui retourne une nouvelle fonction sur le reste des paramètres, cela revient à écrire uniquement des fonctions à un paramètre,

exemple en OCaml : passage d'une fonction prenant un couple d'entiers $(x, y) \rightarrow 2x + 3y$ en une fonction prenant un paramètre entier x et retournant une fonction prenant un paramètre y et calculant $2x + 3y$: $x \rightarrow (y \rightarrow 2x + 3y)$

```
# let f (x,y) = 2*x + 3*y ;;
val f : int * int -> int = <fun>
# f (1,0) ;;
- : int = 2
# f (1,1) ;;
- : int = 5
# f (1,2) ;;
- : int = 8
# let g = function x ->
            (function y -> 2*x + 3*y);;
val g : int -> int -> int = <fun>
```

```
# let h = g 1 ;;
val h : int -> int = <fun>
# h 0 ;;
- : int = 2
# h 1 ;;
- : int = 5
# h 2 ;;
- : int = 8
```

Récursivité (1) : définitions

Syntaxe:

let rec p = *expr*

```
# let rec sigma x = if x = 0 then 0
                    else x + sigma(x-1);;
sigma : int -> int = <fun>

# sigma 10;;
 - : int = 55
```

récursion mutuelle:

```
# let rec odd x = if x = 0 then false else even(x-1)
  and even x = if x = 0 then true else odd(x-1);;
val odd : int -> bool = <fun>
val even : int -> bool = <fun>

# odd 27;;
- : bool = true
```

Récursivité (2) : trace et schéma d'évaluation

trace au toplevel

```
1  # #trace sigma;;
2  sigma is now traced.
3  # sigma 4 ;;
4  sigma <-- 4
5  sigma <-- 3
6  sigma <-- 2
7  sigma <-- 1
8  sigma <-- 0
9  sigma --> 0
10 sigma --> 1
11 sigma --> 3
12 sigma --> 6
13 sigma --> 10
14 - : int = 10
15 # #untrace sigma;;
16 sigma is no longer ↩
       traced.
17 # sigma 4;;
18 - : int = 10
```

schéma d'évaluation

$$
\begin{aligned}
(\text{sigma } 4) &= 4 + (\text{sigma } 3) \\
&= 4 + (3 + (\text{sigma } 2)) \\
&= 4 + (3 + (2 + (\text{sigma } 1))) \\
&= 4 + (3 + (2 + (1 + (\text{sigma } 0)))) \\
&= 4 + (3 + (2 + (1 + 0))) \\
&= 4 + (3 + (2 + 1)) \\
&= 4 + (3 + 3) \\
&= 4 + 6 \\
&= 10
\end{aligned}
$$

2 phases :

- la descente récursive : empilement des additions tant qu'il y a des appels à `sigma`
- remontée récursive : effectuer les additions jusqu'au résultat final

Récursivité (3) : terminale

```
# let rec sigma_aux (n,a) = if n = 0 then a else sigma_aux (n-1,a+n);;
val sigma_aux : int * int -> int = <fun>
# let sigma n = sigma_aux (n,0);;
val sigma : int -> int = <fun>
```

```
# #trace sigma_aux;;
sigma_aux is now traced.
# #trace sigma;;
sigma is now traced.
# sigma 4;;
sigma <-- 4
sigma_aux <-- (4, 0)
sigma_aux <-- (3, 4)
sigma_aux <-- (2, 7)
sigma_aux <-- (1, 9)
sigma_aux <-- (0, 10)
sigma_aux --> 10
sigma_aux --> 10
sigma_aux --> 10
sigma_aux --> 10
sigma_aux --> 10
sigma --> 10
- : int = 10
```

un appel récursif dans lequel la fonction n'exécute aucune instruction après l'appel est un appel récursif terminal.

$$\begin{aligned}(\text{sigma } 4) &= \text{sigma\_aux } (4,0)\\ &= \text{sigma\_aux}(3,4)\\ &= \text{sigma\_aux}(2,7)\\ &= \text{sigma\_aux}(1,9)\\ &= \text{sigma\_aux}(0,10)\\ &= 10\end{aligned}$$

dans un appel récursif terminal il est possible de ré-utiliser le cadre d'appel précédent, ce qui permet de ne pas comsommer de mémoire supplémentaire.

Récursivité (4) : définition locale

fonction auxilaire locale curryfiée.

```
1  # let sigma n =
2      let rec aux n a =
3        if n = 0 then a
4        else aux (n-1) (a+n)
5      in
6        aux n 0 ;;
7        val sigma : int -> int =↩
               <fun>
8
9  # #trace sigma;;
10       sigma is now traced.
11
12 # sigma 4 ;;
13 sigma <-- 4
14 sigma --> 10
15 - : int = 10
```

pas de trace de fonctions locales

schéma d'évaluation de `(sigma 4)` :

$$
\begin{aligned}
(\text{sigma } 4) &= \text{aux } 4\ 0 \\
&= \text{aux } 3\ 4 \\
&= \text{aux } 2\ 7 \\
&= \text{aux } 1\ 9 \\
&= \text{aux } 0\ 10 \\
&= 10
\end{aligned}
$$

le premier cadre d'appel de aux est réutilisé dans les autres appels à aux. Ces appels récursifs sont en taille mémoire constante.

Récursivité (5) : définition locale

Une fonction locale peut utiliser l'ensemble de l'environnement accessible.

```
1  # let pow x n =
2      let rec aux n a =
3        if n = 0 then a
4        else aux (n-1) (x * a)
5      in
6        aux n 1 ;;
7  val pow : int -> int -> int = <fun>
8  # pow 2 5;;
9  - : int = 32
```

ici à la ligne 4 le paramètre x de pow est utilisé dans aux.

C'est un style camélien assez classique.

Polymorphisme paramétrique (1)

appelé « généricité » dans d'autres langages

- même code pour des arguments de types différents
- la fonction n'utilise pas la structure de l'argument

```
# let mp a b = a,b;;
val mp : 'a -> 'b -> 'a * 'b = <fun>

# let x = mp 3 6.8;;
val x : int * float = 3, 6.8

# let y = mp true [1];;
val y : bool * int list = true, [1]

# fst x;;
- : int = 3
```

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let app x y = x y;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
# app id 1;;
- : int = 1
```

OCaml (et F#) : Résumé des expressions rencontrées

```
expr ::= constante
       | ( expr )
       | ident
       | Mod.ident
       | op expr
       | expr infix-op expr
       | if expr then expr else expr
       | function ident -> expr
       | expr expr
       | expr , expr
       | epxr :: epxr
       | let [rec] ident = expr
           [ and ident = expr ]*          // pas en F#
         in expr
```

OCaml : List.hd et List.tl

F# : List.head et List.tail

Filtrage par motif

permet l'accès aux structures de données

- en testant une valeur
- en nommant une partie de la structure

motif:

- assemblage correct (syntaxe et type) d'objets
  - de types de base (*int*, *bool*,...)
  - de paires, listes et constructeurs
  - d'identificateurs
  - du motif « ramasse tout » (_)
- ce n'est pas une expression (il n'y a pas de calcul)

Syntaxe du filtage par motif

Syntaxe:

match e **with** p_1 -> e_1 | p_2 -> e_2 ...| p_n -> e_n

- filtrage séquentiel de l'expression e par les différents motifs p_i.
- Si un des motifs correspond à la valeur de e, alors sa branche e_i est évaluée.
- tous les e_i sont du même type, idem pout les p_i et le type de e
- motif linéaire (motif non-linéraire comme (x,x)) interdit)
- détection d'un filtrage non exhaustif
- détection de branches inutiles

Exemple : fonction imply

- Enumération des cas

```
# let imply v =
     match v with
     |    (true,true) -> true
     |   (true,false) -> false
     |   (false,true) -> true
     |  (false,false) -> true;;
val imply : bool * bool -> bool = <fun>
```

- Version plus compacte

```
# let imply v =
     match v with
     |  (true,x)  -> x
     |  (false,_) -> true;;
val imply : bool * bool -> bool = <fun>
```

Warnings

- filtrage non exhaustif :

```
# let f x =
     match x with
     | (true, x) -> true
     | (false, false) -> true;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(false, true)
val f : bool * bool -> bool = <fun>
```

- cas inutile :

```
# let f x =
     match x with
     | (a,b) -> true
     | (true,false) -> false;;
Warning: this match case is unused.
val f : bool * bool -> bool = <fun>
```

Linéarité des motifs et motif-OU

- erreur : motif non linéaire

```
# let eq_c c =
     match c  with
     | (x,x) -> true
     | (x,y) -> false
  ;;
Error: Variable x is bound several times in this matching

# let (a,a) = (1,3);;
Error: Variable a is bound several times in this matching
```

- motif-OU

```
# let proj x =
     match x with
     | (true,i,_) | (false,_,i) -> i
  ;;
val proj : bool * 'a * 'a -> 'a = <fun>
```

Une variable apparaissant dans différents motif-OU doit être d'un seul type.

Motif dans les déclarations

- **Formes équivalentes:**

  Syntaxe:

  match *e* **with** filtrage $\equiv$ (**function** filtrage) *e*

```
#  match (2,1)
    with  (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d ;;
- : int = 2
# function (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d ;;
- : int * int -> int = <fun>
# (function (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d)
   (2,1) ;;
- : int = 2
```

- **Déclarations déstructurantes:**

  Syntaxe:

  let $p = e$ p est un motif

```
# let (x,y) = (3,true);;
val x : int = 3
val y : bool = true
```

Opérateur xor sur les booléens

Style camélien : définition par cas:

3 versions avec filtrage de motif de OU exclusif (xor)

```
# let xor (b1:bool) (b2:bool) : bool =
  match b1, b2 with
  | true, false  -> true
  | false, true  -> true
  | false, false -> false
  | true, true   -> false ;;
val xor : bool -> bool -> bool = <fun>
# let xor (b1:bool) (b2:bool) : bool =
  match b1, b2 with
  | true, false -> true
  | false, true -> true
  | _ -> false ;;
val xor : bool -> bool -> bool = <fun>
# let xor (b1:bool) (b2:bool) : bool =
  match b1, b2 with
  | true, true -> false
  | true, false -> true
  | false, r -> r ;;
val xor : bool -> bool -> bool = <fun>
```

Exemple d'un additionneur (1 bit)

On commence par un demi-additionneur, la «demi-somme» des bits b1 et b2 est donnée par le ou exclusif (xor) et la retenue par la conjonction.

```
# let half_adder (b1:bool) (b2:bool) : (bool * bool) =
    ( xor b1 b2 , b1 && b2 ) ;;
val half_adder : bool -> bool -> bool * bool = <fun>
```

```
# let adder (b1:bool) (b2:bool) (c0:bool) : (bool*bool) =
   match (half_adder b1 b2) with
   | (s1,c1) -> ( match (half_adder s1 c0) with (s2,c2) -> (s2, c1 || c2))
  ;;
val adder : bool -> bool -> bool -> bool * bool = <fun>
```

Exemple d'un additionneur (4 bits)

Un couple, plus généralement un n-uplet, peut être argument d'une fonction. On utilise notre additionneur pour définir une fonction d'addition de deux quartets (4 bits). On représente un quartet par un quadruplet de booléens, de type `bool * bool * bool * bool`. On peut définir un racourci pour ce type :

```
1   type quartet = bool * bool * bool * bool
```

L'additionneur 4 bits prend en argument deux quartets `q1` et `q2`, et donne le couple formé du quartet qui contient la somme de `q1` et `q2` et de la retenue de cette somme :

```
1  # let add4bits (q1:quartet) (q2:quartet) : (quartet*bool) =
2    match q1, q2 with
3    | (a1,a2,a3,a4), (b1,b2,b3,b4) ->
4      let (s1,c1) = adder a1 b1 false in
5      let (s2,c2) = adder a2 b2 c1 in
6      let (s3,c3) = adder a3 b3 c2 in
7      let (s4,c4) = adder a4 b4 c3 in
8            ((s1,s2,s3,s4), c4) ;;
9  val add4bits : quartet -> quartet -> quartet * bool = <fun>
```