Programmation Fonctionnelle Avancée

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

PARIS DIDEROT

treinen@irif.fr

10 septembre 2018

© Roberto Di Cosmo et Ralf Treinen

Organisation

- 12 cours, dernier cours le 28 novembre.
- 12 séances de TP. Premier TP : le 13 septembre.
- Il y a un projet de programmation, mais pas de partiel
- Contrôle de connaissances :

$$\frac{1}{3} * \text{projet} + \frac{2}{3} * \text{exam}$$

- Note 2ème session :

$$\max(\frac{1}{3} * \text{projet} + \frac{2}{3} * \text{exam2}, \text{exam2})$$

Organisation

- http://www.irif.fr/~treinen/teaching/pfav/
- Support : copies des transparents
- Il y a des ressources en ligne (voir la page web du cours)
- Les TP sont assurés par Pierre Letouzey
- Il est indispensable d'assister au cours et aux TD/TP
- Au TP vous auriez besoin de votre login de l'UFR (sinon le récupérer au bureau 3061)

Le projet

- Le sujet détaillé sera mis en ligne plus tard.
- Projet complet à rendre (la date limite sera annoncée plus tard).
- Soutenance prévue pendant la période des examens en janvier.
- Peut être fait en binôme, mais *la notation est individuelle*.

Pré-requis de ce cours

- Ce cours est la continuation du cours *PF5 - Programmation Fonctionnelle* du L3.
- Mais il peut être suivi par des étudiants qui ont suivi un autre cours de programmation fonctionnelle en OCaml (même si niveau L1) ...
- ... ou un autre langage fonctionnel (Scheme, Haskell).
- Dans ce cas il va falloir vous mettre à niveau.

Pour vous rafraîchir la mémoire ou vous mettre à niveau

- Les transparents du cours du L3
- Le document *Initiation à la programmation fonctionnelle* par Jean-Christophe Filliâtre, jusqu'à section 2.1.2 incluse.
- Dans le livre *Développement d'applications avec Objective Caml* :
  - Chapitre 2 : Programmation fonctionnelle
  - Chapitre 3 : Programmation impérative
  - Chapitre 7 : Compilation
  - Chapitre 14 : seulement Modules Simples

OCaml : Le MOOC

- 7 semaines, 17/9 -> 12/12
- `https://www.fun-mooc.fr/courses/course-v1:parisdiderot+56002+session03/about`
- gratuit, inscription sur FUN
- en VO ou VOSTF

Objectif du cours

John Carmack

> *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*

Dans ce cours, nous allons explorer ensemble

- plusieurs aspects *avancés*
- de la *programmation fonctionnelle*
- en utilisant *le langage OCaml*

Les fonctions sont des valeurs de première classe

Dans les langages dits *fonctionnels*, les fonctions sont des *entités de première classe*, i.e. une fonction

- peut être construite sans besoin d'être nommée : il y a des expressions dont la valeur est une fonction ;
- être définie et nommée au même moment.

Il y a deux mécanismes indépendants :

- des expressions fonctionnelles,
- lier un identificateur à une valeur (fonctionnelle ou pas).

Exemples (fonctions1.ml)

```
(* une valeur fonctionnelle *)
function x -> x+2;;

(* definir et nommer une fonction *)
let f = function x -> x ;;

(* plus court *)
let f x = x ;;
```


Les fonctions sont des valeurs de première classe

Dans les langages dits *fonctionnels*, les fonctions sont des *entités de première classe*, i.e. une fonction

- peut être placée à l'intérieur d'une structure de données ;
- peut être passée en paramètre à une fonction, et retournée comme résultat d'une fonction.

Exemples (fonctions2.ml)

```
(* fonctions dans un record *)
type 'a foo = {n:int; f: 'a -> 'a} ;;
let foo = {n=42; f=fun x -> x} ;;
foo.f 33;;

(* fonctions dans une paire. *)
let fp = ( (fun x -> x+1), (fun x -> x*x) );;

(* attention aux parantheses *)
let fp = ( fun x -> x+1, fun x -> x*x );;

(* extraction des fonctions d'une paire *)
(fst fp) 42;;
(fst fp) ((snd fp) 10);;
(fst fp) (snd fp) 10;; (* attention aux parantheses *)
```

Exemples (fonctions3.ml)

```
let double x = 2*x ;;

(* une fonction avec un argument fonctionnel *)
let apply_twice_to f n = f (f n);;

apply_twice_to double 5;;

(* arguments, et resultat fonctionnels *)
let compose f g = fun x -> f (g x);;

compose double (fun x -> x+2);;

compose double double 10;;
```


Les fonctions sont des valeurs de première classe

Dans les langages dits *fonctionnels*, les fonctions sont des *entités de première classe*, i.e. une fonction

- peut être partiellement appliquée ;
- peut créée dynamiquement ;
- peut être définie localement à une fonction.

Exemples (fonctions4.ml)

```
let mul x y = x*y;;

(* application partielle *)
let double = mul 2;;

let scale f k =
  let factor = double k in
  fun x -> factor * (f x) ;;

scale (fun x -> x+1) 2 10;;
```


Exemples (fonctions5.ml)

```
(* fonctions locale a une autre fonction *)

let rev l =
  let rec aux acc =
    function
    | [] -> acc
    | a::r -> aux (a::acc) r in
  aux [] l ;;

rev [1; 2; 3];;
```

Liaison statique

- en anglais : *lexical scoping*
- Un identificateur libre (c-à-d qui n'est pas un paramètre) dans une expression fonctionnelle garde la liaison de son occurrence textuelle dans le programme.
- On parle d'une *clôture* (angl. : *closure*)
- Tous les langages de programmation modernes (fonctionnels ou pas) suivent cette politique.

Exemples (fonctions6.ml)

```
(* n est une variable libre dans function x-> x*n *)
let multiplier_avec n = function x -> x*n;;
let f = multiplier_avec 5;;
f 3;;

let m = 17;;
let g x = x + m;; (* m est libre *)
let m = 42;;
g 1;; (* quel est le resultat ? *)

let f x = x + 55;;
let g y = f (f y);;
let f x = x + 1000;;
g 10;;
```


Des fonctions de première classe dans Java 8 !

Exemple : Appeler une procédure qui prend un filtre en argument (Java 8) :

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

En Java on est obligé de spécifier le type de l'argument.
Voir : http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Des fonctions de première classe dans Python !

```
from functools import reduce
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

envoie $(((1+2)+3)+4)+5=15$.

- L'utilisation des lambda en Python est restreinte dû à la distinction entre expressions et instructions.
- Pas de typage statique en Python.

Des fonctions de première classe dans les maths

- Alonzo Church, debut des années 1930 : Lambda Calculus.
- Notation : $\lambda x.x + 1$
- À l'époque utilisé pour l'étude de la fondation des mathématiques
- Aujourd'hui : informatique théorique
- Voir le cours de *Sémantique* du M1

Le commencement de l'histoire...

JOURNAL OF COMPUTER AND SYSTEM SCIENCES 17, 348–375 (1978)

A Theory of Type Polymorphism in Programming

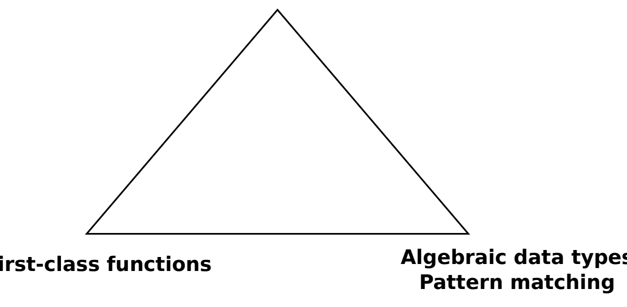
ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm $\mathscr{W}$ which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong" and a Syntactic Soundness Theorem states that if $\mathscr{W}$ accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on $\mathscr{W}$ is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system.

... Core ML



Histoire d'OCaml

- 1973 ML Milner (tactiques de preuves pour le prouveur LCF)
- 1980 Projet Formel à l'INRIA (Gérard Huet), Categorical Abstract Machine (Pierre-Louis Curien)
- 1984-1990 Définition de SML (Milner)
- 1987 Caml (implémenté en Lisp) Guy Cousineau, Ascander Suarez, (avec Pierre Weis et Michel Mauny)
- 1990-1991 Caml Light par Xavier Leroy (et Damien Doligez pour la gestion de la mémoire)
- 1995 Caml Special Light puis 1996 OCaml (Xavier Leroy, Jérôme Vouillon, Didier Rémy, Michel Mauny)

Voir `https://ocaml.org/learn/history.fr.html`

Traits intéressants du langage

- Les fonctions fournissent un mécanisme d'abstraction puissant.
- Un système de type polymorphe et implicite qui capture beaucoup d'erreurs.
- La définition par cas simplifie et sécurise l'implémentation
- Pas de pointeurs explicites, GC efficace - programmation de très haut niveau.
- Évaluation stricte et structures de données mutables.

Exemples (types.ml)

```
let index = input_line (open_in "data");;

let dict =  [1,"one"; 2,"two"];;

List.assoc index dict;;

List.assoc (int_of_string index) dict;;
```


Exemples (cases.ml)

```
let rec destutter l =
 match l with
  | [] -> []
  | x :: y :: rest ->
    if  x = y then destutter (y :: rest)
    else  x :: destutter (y :: rest) ;;

destutter [1; 2; 2; 3; 4];;
```


Exemples (mutable.ml)

```
let a = [|1;2;3|];;

a.(0)<- 5;;

a.(0);;

a;;
```

Inférence de types en OCaml

- Pas besoin de spécifier les types des identificateurs on OCaml - le système trouve le type le plus général.
- Combine concision du code avec la sûreté du typage fort.
- Java : typage fort (sûr), mais on est obligé de spécifier les types (lourd).
- Python : typage “dynamique” (pendant l’exécution du code). Code concis et élégant, mais on perd en sûreté (erreurs apparaissent seulement pendant les tests).

Typage explicite en OCaml

- On a le *droit* de spécifier le type des identificateurs (en fait, des expressions quelconques).
- Syntaxe : (<expression> : <type>)
- Il faut que le type donné soit compatible avec le type trouvé par le compilateur (le même, ou plus restrictif).
- Il ne s’agit **pas** d’un mécanisme de conversion de type.

Exemples (explicit.ml)

```ocaml
(* specifier un type *)
let (x:int) = 42;;

let triple x = (x, x, x);;

let triple (x: string) = (x, x, x);;

let triple x = ( (x,x,x) : string*string*string );;

(* pas un mecanisme de conversion *)
let (x:float) = 42;;
```


Pattern matching et définitions par cas

Une des caractéristiques les plus appréciées des langages de la famille ML comme OCaml est la définition par cas :

```ocaml
let rec fact = function
| 0 -> 1
| n -> n * (fact (n-1));;
```

est équivalent au code suivant

```ocaml
let rec fact n =
  if n=0 then 1 else n * (fact (n-1));;
```

Jusqu’ici, on ne voit pas trop ce qu’on gagne par rapport à faire des conditionnelles en cascade.

Une définition par cas résume plusieurs façons de faire des tests

Voyons une définition un peu moins banale :

```
let f x y = match x,y with
  | true,false -> 1
  |    _  , true -> 2
  | false,   _   -> 3;;
```

elle peut se traduire en :

```
let f1 x y =
  if x then
    if y then 2 else 1
  else
    if y then 2 else 3;;
```

mais aussi, plus concisément :

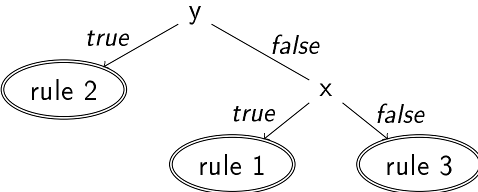
```
let f2 x y =
  if y then 2
  else
    if x then 1 else 3;;
```



```
let f x y = match x,y with
  | true,false -> 1
  |    _  , true -> 2
  | false,   _   -> 3;;


let f1 x y =
  if x then
    if y then 2 else 1
  else
    if y then 2 else 3;;

let f2 x y =
  if y then 2
  else
    if x then 1 else 3;;

let tests = [true,true;false,false;false,true;true,false];;
let test f = List.map (fun (x,y) -> f x y) tests;;
test f;;
test f1;;
test f2;;
```


Les arbres de décision

- Pour chaque définition par cas dans le source OCaml, le compilateur doit produire une série de tests qui permettent de déterminer, pour toute donnée en entrée, quelle ligne de la définition s'applique.
- On peut représenter cette série de tests avec un *arbre de décision*. Sur l'exemple :



Utilité des arbres de décision

Étant donné un arbre de décision associé à une définition par cas, il est facile de

- identifier les définitions incomplètes : les cas oubliés sont les branches absentes d'un noeud de décision
- identifier les définitions inutiles : si une régle n'apparaît dans aucune feuille, elle ne sera jamais utilisée

C'est précisément ce qui rend la définition par cas si utile et populaire parmi les programmeurs ML : ce n'est *pas* la même chose qu'une librairie de motif ajoutée au langage comme on peut en trouver pour Scheme ou Java.

Exemples (redundant.ml)

```
(* tester l'appartenance a une liste *)
let rec is_member x = function
  | [] -> false
  | x::r -> true
  | h::r -> is_member x r
;;

(* ne fait pas ce qu'on attend ! *)
is_member 42 [1; 42; 73];;
is_member 42 [];;
is_member 17 [1; 42; 73];;
```


Pointeurs pour aller plus loin

La compilation des définitions par cas (*pattern matching*) a été étudiée depuis les années 1980.

Marianne Baudinet and David Macqueen.
Tree pattern matching for ml (extended abstract).
Technical report, Stanford University, 1985.

Fabrice Le Fessant and Luc Maranget.
Optimizing pattern-matching.
In Proceedings of the 2001 International Conference on Functional Programming. ACM Press, 2001.

Luc Maranget.
Compiling pattern matching to good decision trees.
ML'2008.

Quelques résultats

- Marianne Baudinet et David MacQueen, 1985 : trouver le plus petit arbre de décision est un problème NP-Complet
- Lefessant et Maranget, 2001 : des heuristiques efficaces pour OCaml (c'est l'algorithme implanté aujourd'hui)

Traits intéressants de l'environnement de programmation

un outillage avancé

- un "toplevel" ou REPL (Read-Evaluate-Print Loop)
- un compilateur bytecode (portable)
- un compilateur natif, très performant
- un compilateur vers JavaScript (`js_of_ocaml`)
- un système de module puissant
- une couche objet
- des gestionnaires de paquets (en particulier `opam`)
- ... etc.

Qui utilise OCaml ?

- l'enseignement : ici, par exemple !
- la recherche : Coq, Astree, Ocsigen, Mirage, ...
- la communauté : Unison, MLDonkey, ...
- l'industrie : Citrix, Dassault, Esterel, Lexifi, Jane Street Capital, OcamlPro, Baretta, Merjis, RedHat, ...

Voyons quelques exemples concrets

Operating systems : Citrix, Xen

Les outils de gestion de `Xen`, l'hyperviseur qui fait fonctionner des millions de machines virtuelles dans le `cloud` sont écrits en OCaml.

> *OCaml has brought significant productivity and efficiency benefits to the project. OCaml has enabled our engineers to be more productive than they would have been had they adopted any of the mainstream languages.*
>
> *Richard Sharp, Citrix*

David Scott, Richard Sharp, Thomas Gazagnaire and Anil Madhavapeddy.
Using functional programming within an industrial product group : perspectives and perceptions.
International Conference on Functional Programming, 2010.

Operating systems : Mirage

Mirage is an exokernel for constructing secure, high-performance network applications across a variety of cloud computing and mobile platforms.

> *Code can be developed on a normal OS such as Linux or MacOS X, and then compiled into a fully-standalone, specialised microkernel that runs under the Xen hypervisor. Since Xen powers most public cloud computing infrastructure such as Amazon EC2, this lets your servers run more cheaply, securely and finer control than with a full software stack. Mirage is based around the OCaml language ...*

http://www.openmirage.org

Social networks : Mylife.com

`Mylife.com` est un agrégateur de réseaux sociaux écrit en OCaml, avec plusieurs contributeurs, dont Martin Jambon

> *The ocaml language and its libraries offer a good balance between expressiveness and high performance, and we dont have to switch to lower-level languages when we need high performance.*
>
> *Martin Jambon, Mylife.com*

Finance : JaneStreet

L'entreprise `JaneStreet` utilise OCaml pour son activité de trading.

Voilà ce que dit Yaron Minsky dans

Yaron Minsky.
OCaml for the masses.
Communications of the ACM, September 2011

Concision

Our experience with OCaml on the research side convinced us that we could build smaller, simpler, easier-to-understand systems in OCaml than we could in languages such as Java or C#. For an organization that valued readability, this was a huge win.

Bug detection

Programmers who are new to OCaml are often taken aback by the degree to which the type system catches bugs. The impression you get is that once you manage to get the typechecker to approve of your code, there are no bugs left.
This isn't really true, of course ; OCaml's type system is helpless against many bugs.
There is, however, a surprisingly wide swath of bugs against which the type system is effective, including many bugs that are quite hard to get at through testing.

Performance

We found that OCaml's performance was on par with or better than Java's, and within spitting distance of languages such as C or C++. In addition to having a high-quality code generator, OCaml has an incremental GC (garbage collector). This means the GC can be tuned to do small chunks of work at a time, making it more suitable for soft real-time applications such as electronic trading.

Pure, mostly

Despite how functional programmers often talk about it, mutable state is a fundamental part of programming, and one that cannot and should not be done away with. Sending a network packet or writing to disk are examples of mutability.
A complete commitment to immutability is a commitment to never building anything real.

Static Analysis : Esterel, Absynthe, Airbus, Facebook ...

Esterel code generator written in OCaml.

Facebook does sophisticated static analysis using OCaml over their vast PHP codebase to close security holes.

Airbus, Absynthe use of Astree, written in OCaml, to prevent bugs in Airbus code.

Microsoft SLAM and Static Driver Verifier

...

Nouvelles recentes : OCamlLabs et OCamlPro

Il y a désormais des initiatives concrètes pour assurer le support industriel de OCaml :

- OCamlPro est une entreprise de service française dédiée au support
- OCamlLabs est une structure non-for-profit qui se consacre au dévéloppement d'une plateforme OCaml stable

Et ils cherchent des personnes qui aiment bien programmer.

Le plan preliminaire du cours

- Système de *modules* avancé
  encapsulation, types *privés*, interfaces, foncteurs ou modules paramétrés
- Structures de données fonctionnelles efficaces *zippers*, files, arbres équilibrés
- Analyse de coût amorti, raisonnement équationnel
- Structures de données *infinies et paresseuses* : flots (*streams*)
- Structures de données *compactes* (*hash-consing*, *memoization*)
- Utilisation avancée du *système de types variants polymorphes*, *GADT*
- Programmation avec *combinateurs*, DSL, combinateurs pour le calcul parallèle
- *Monades*

Bibliographie

- Xavier Leroy et al : The Objective Caml system : documentation and user's manual
  `http://caml.inria.fr`
- Emmanuel Chailloux, Pascal Manoury et Bruno Pagano : Développement d'Applications avec Objective Caml
  O'Reilly, 2000 disponible en ligne
- Chris Okasaki : Purely Functional Data Structures
  Cambridge University Press, 1998 disponible en ligne