Programmation Fonctionnelle Avancée

Roberto Di Cosmo

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

roberto@dicosmo.org

September 17, 2013

Organisation

- Les cours ont lieu jusqu'à la semaine du 11 Décembre, les TD/TP commencent cette semaine (aujourd'hui)
- Période des examens : du 6 au 17 Janvier, avec une deuxième session du 16 au 30 Juin
- 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.dicosmo.org/CourseNotes/pfav`
- Support : copies des transparents
- Il y a des ressources en ligne (voir la page web du cours)
- Il est indispensable d'assister au cours et aux TD/TP
- Le TD est assuré par Gabriel Scherer

Le projet

Le sujet détaillé sera mis en ligne pendant la 4ème semaine du cours.

Organisation

- Projet complet à rendre au plus tard le Vendredi 13 Décembre
- Peut être fait en binôme, mais *la notation est individuelle*

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

Fonctions 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

```
fun x -> x ;;
|| - : 'a -> 'a = <fun>
```


Fonctions de première classe

être nommée

```
let f = fun x -> x ;;
|| val f : 'a -> 'a = <fun>
```

Ceci est équivalent

```
let f x = x ;;
|| val f : 'a -> 'a = <fun>
```


Fonctions de première classe

être placé à l'intérieur d'une structure de données

```
type 'a foo = {f: 'a -> 'a} ;;
|| type 'a foo = { f : 'a -> 'a; }

let foo = {f=fun x -> x} ;;
|| val foo : 'a foo = {f = <fun>}

foo.f 33;;
|| - : int = 33
```

Fonctions de première classe

être passé en paramètre à une fonction, et retournée comme résultat d'une fonction

```
let fst (a,b) = a ;;
|| val fst : 'a * 'b -> 'a = <fun>

let f = fst ((fun x -> x), (fun x -> x+2));;
|| val f : '_a -> '_a = <fun>

let compose f g = fun x -> f (g x);;
|| val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

compose (fun x -> x) (fun x -> x+2);;
|| - : int -> int = <fun>
```


Fonctions de première classe

être partiellement appliquées et crée dynamiquement

```
let mul x y = x*y;;
|| val mul : int -> int -> int = <fun>

let double = mul 2;;
|| val double : int -> int = <fun>

let square f = fun x -> f (f x);;
|| val square : ('a -> 'a) -> 'a -> 'a = <fun>

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


Fonctions de première classe

être définie localement à une fonction

```
let rev l =
   let rec aux acc =
      function
        [] -> acc
      | a::r -> aux (a::acc) r in
   aux [] l ;;
|| val rev : 'a list -> 'a list = <fun>
```


>

Le commencement de l'histoire...

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

A Theory of Type Polymorphism in Programming

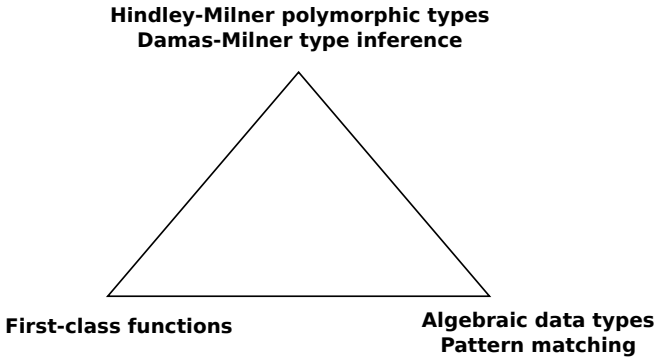
ROBIN MILNER

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 `http://events.inf.ed.ac.uk/Milner2012/X_Leroy-html5-mp4.html` !

Plusieurs traits intéressants

les fonctions fournissent un méchanisme d'abstraction puissant

```
    let double l = List.map (fun x -> x*2) l ;;
|| val double : int list -> int list = <fun>
```

un système de type polymorphe et implicite qui capture beaucoup d'erreurs

```
let index = input_line (open_in "data");;
|| val index : string = "2"

List.assoc index [1,"one";2,"two"];;
|| Characters 18-19:
||   List.assoc index [1,"one";2,"two"];;
||                    ^
|| Error: This expression has type int but an expression was expected of ty
|| pe
||          string
```


Plusieurs traits intéressants

la définition par cas simplifie et sécurise l'implémentation

```
let rec destutter l =
 match l with
  | []              -> []
  | x :: y :: rest ->
    if  x = y then destutter (y :: rest)
    else  x :: destutter (y :: rest) ;;
|| Characters 24-161:
||   .match l with
||     | []              -> []
||     | x :: y :: rest ->
||       if  x = y then destutter (y :: rest)
||       else  x :: destutter (y :: rest)...
|| Warning 8: this pattern-matching is not exhaustive.
|| Here is an example of a value that is not matched:
|| _::[]
|| val destutter : 'a list -> 'a list = <fun>
```

Plusieurs traits intéressants

pas de pointeurs explicites, GC efficace - programmation de très haut niveau

```
let rec rev_append l1 l2 =
match l1 with
  [] -> l2
| a :: l -> rev_append l (a :: l2);;
|| val rev_append : 'a list -> 'a list -> 'a list = <fun>

let rev l = rev_append l [];;
|| val rev : 'a list -> 'a list = <fun>
```


Plusieurs traits intéressants

évaluation stricte et structures de données mutables

```
let a = [|1;2;3|];;
|| val a : int array = [|1; 2; 3|]

a.(0)<- 3;;
|| - : unit = ()

a.(0);;
|| - : int = 3
```


Plusieurs traits intéressants

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.

Programmation Fonctionnelle Avancée
Introduction
Applications

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`

Programmation Fonctionnelle Avancée
Introduction
Applications

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*

Programmation Fonctionnelle Avancée
Introduction
Applications

Finance: JaneStreet

`JaneStreet` est une compagnie qui 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
- Utilisation avancée du *système de types*
  variants polymorphes, *GADT*
- Programmation avec *combinateurs*, DSL
- 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*)
- Construction *modulaire* d'interprètes
- *Monades* et transformateurs de Monades
- Transformations de programmes
- Concurrence et parallélisme

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
- Markus Mottl,
  `http://ocaml.info/home/ocaml_sources.html`

Pour commencer: les définitions par cas

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

```
let rec fact = function
| 0 -> 1
| n -> n * (fact (n-1));;
|| val fact : int -> int = <fun>
```

est équivalent au code suivant

```
let rec fact n =
if n=0 then 1 else n * (fact (n-1));;
|| val fact : int -> int = <fun>
```

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 comme

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

mais aussi, plus concisement

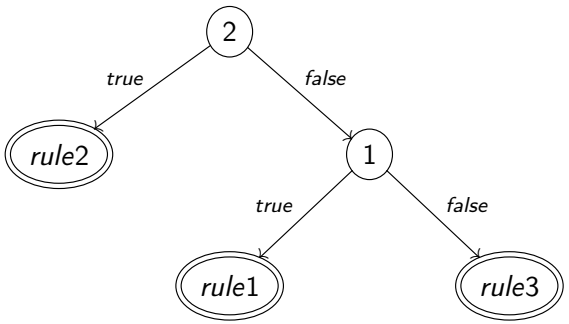
```
let f2 x y =
  if y then 2
  else
    if x then 1 else 3;;
```


Parenthèse: vérifions

```
let tests = [true,true;false,false;false,true;true,false];;
|| val tests : (bool * bool) list =
||   [(true, true); (false, false); (false, true); (true, false)]

let test f = List.map (fun (x,y) -> f x y) tests;;
|| val test : (bool -> bool -> 'a) -> 'a list = <fun>

test f;;
|| - : int list = [2; 3; 2; 1]

test f1;;
|| - : int list = [2; 3; 2; 1]

test f2;;
|| - : int list = [2; 3; 2; 1]
```


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*.
Le dernier code vu donne, par 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 précisement 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.

Un très grand nombre d'arbres de décision

Question:
Combien d'arbres de décisions différents y a-t-il pour une définition par cas à n arguments?

Exercice:
écrivez tous les encodages possibles, avec des conditionnelles, pour la fonction suivante

```
let f x y z = match x,y,z with
|   _   ,false, true -> 1
| false, true,   _    -> 2
|   _   ,  _  ,false -> 3
|   _   ,  _  , true -> 4;;
val f : bool -> bool -> bool -> int = <fun>
```


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)