Présentation du Cours
Introduction à OCaml
INF 441 : Programmation Avancée

Xavier Rival

10 avril 2017

Outline

Objectifs du cours

Mieux comprendre la programmation
Apprendre à construire des programmes de meilleure qualité
(même pour des applications complexes nécessitant plus de code)

A priori,

- Vous avez appris à programmer ; cf INF 311, INF 321, INF 411
- Vous avez acquis de bonnes bases en algorithmiques ; cf INF 421
- Vous avez peut être abordé la concurrence en INF 431 ou acquis une expérience en programmation via un Modal...

Que vous reste-t'il donc à apprendre en programmation ?

Programmation à grande échelle : défis et contraintes

Penchons nous sur quelques développements logiciels open source...

Le noyau Linux :

- système d'exploitation développé depuis 25 ans
- environ **13 millions de lignes** pour Linux 4.0, 40 000 fichiers
- plus de **10 000 développeurs**, dont au moins 1 000 actifs
- support pour de nombreuses architectures : x86, x86-64, PPC, ARM...
- des milliers de modules disponibles : systèmes de fichiers, pilotes de périphériques, protocoles réseaux / crypto...
- impossible de connaître tout le code...

VLC :

- lecteur de fichiers multimédias, initialement un projet d'étudiant...
- 16 ans plus tard, environ **400 modules** supportant différents formats

Programmation à grande échelle : défis et contraintes

Les développements industriels font face à des défis similaires...

Google, **Facebook**, **Apple** :

- développeurs : 10 000+
- bases de code : en dizaines de millions de lignes
- contraintes importantes au niveau de la sécurité

Automobile :

- de l'ordre de 30 millions de lignes dans un véhicule récent
- de nombreux équipementiers
- contraintes de sûreté : freinage, airbags, contrôle moteur...
  sans parler des véhicules autonomes...

Comment de tels développements sont-ils possibles ?

Modularité en programmation

Pour développer des logiciels de grande taille, il faut **une organisation modulaire** :

- découpage en **composants** séparés, et indépendants les uns des autres
- qui peuvent ensuite **être assemblés**

Idéalement, ces composants indépendants doivent être faciles à :

- **réutiliser** dans des contextes différents (imprévisibles au départ)
- **remplacer** par d'autres composants équivalents
- **maintenir** et **faire évoluer** sans affecter le reste du système
- **tester** et **prouver** séparément

Nous allons donc étudier les mécanismes de modularité

Programmation et abstraction

Une notion fondamentale en programmation : l'abstraction

Programmation et abstraction

Une notion fondamentale en programmation : l'abstraction

Interprétation 1 :

abstraire, c'est cacher, encapsuler

- fournir des structure et opérations associées, cacher les détails
  ex : `Hashtbl` fournit une structure, des fonctions `mem`, `add`, `iter`...
- interprétation logique : **existentielle**
  il *existe* un algorithme tel que ...
- principe des **langages haut niveau** :
  éviter de penser en assembleur, se détacher des détails du processeur, et se concentrer sur le code haut niveau

Programmation et abstraction

Une notion fondamentale en programmation : l'abstraction

Interprétation 2 :

abstraire, c'est paramétrer

- définir des programmes **généraux**
  ex : implémentation des arbres binaires de recherche,
  indépendante de celle des clés
- interprétation logique : **universelle**
  pour tout $\mathcal{X}$, on peut définir $\mathcal{Y}$ tel que...

Mécanismes d'abstraction

Il existe de nombreux mécanismes d'abstraction en programmation

Les **fonctions** :

- paramétrisation :
  une définition, valable pour de nombreux arguments possibles
- masquage d'informations :
  une fonction cache son implémentation au code appelant

Les **objets** :

- paramétrisation :
  interaction via les méthodes
- mécanisme de masque des informations :
  catégories *publiques* / *privées* des champs et méthodes

Il en existe bien d'autres...

Langages et applications

Langage, champ d'application et mécanismes d'abstraction sont liés

Systèmes d'exploitation / drivers :

- il faut un accès direct à la **représentation machine**...
- **C** fournit un bas niveau d'abstraction, adapté à cela

Compilateurs / systèmes de preuve et vérification :

- on souhaite faire du **calcul symbolique**, manipuler des arbres syntaxiques, écrire des fonctions récursives
- les **langages fonctionnels** (OCaml...) sont idéaux

Systèmes réactifs (embarqué, musique, ...) :

- le **temps** et la notion d'**horloge** sont centraux
- on utilise des **langages synchrones** (Lustre, Esterel...)

Beaucoup d'autres exemples : tableurs, langages orientés objets...

Sémantique et programmation

Comment comprendre effectivement **ces mécanismes d'abstraction** ?

Il faut pouvoir définir précisément, voire formellement :

- la **signification** des traits de programmation correspondants
  i.e., comment un programme les utilisant est-il exécuté ?
- le **modèle de coût** qui leur est associé

Sémantique d'un programme

Définition mathématique de ses comportements

- la sémantique des langages de programmation est une discipline complexe, au-delà du cadre de ce cours...
- ... mais nous nous en inspirerons quand cela sera utile

Maîtriser l'abstraction en programmation

Objectifs du cours

**Comprendre la notion d'abstraction en programmation,
et les mécanismes d'abstraction offerts
par les langages de programmation modernes**

On s'intéressera à la fois aux **principes fondamentaux** et à leur **mise en pratique**, pour répondre à des questions telles que :

- à quel niveau d'abstraction le programmeur doit il réfléchir ?
- quel trait de langage est le plus adapté à un problème donné ?
- comment exploiter un langage pour exprimer / coder ses idées au mieux ?
- quel est le modèle de coût associé aux fonctionnalités du langage ?

Plan du cours

Programme sur huit blocs :

1. **Présentation et introduction à OCaml** (aujourd'hui)
2. **Types** : abstraction pour la sûreté
3. **Fonctions** : programmation fonctionnelle
4. **Modules et foncteurs** : code modulaire et paramétrage
5. **Objets** : classes et héritage
6. **Itérateurs** : itération sur les objets abstraits bornés
7. **Streams et threads** : itération perpétuelle et asynchrone
8. **Programmes, compilation et exécution** : implémentation d'un langage

Langage utilisé

Dans ce cours, **nous utilisons principalement OCaml**

car ce langage

- **illustre tous les concepts du cours**
- favorise l'écriture de **code concis et clair**
- a **une sémantique claire**
- est **complémentaire à Java**, utilisé dans les cours précédents
- *"is for the masses"*, dixit un trader haut fréquence :-)
  voir `http://queue.acm.org/detail.cfm?id=2038036`

Toutefois, il ne s'agit pas d'un cours d'OCaml

OCaml est ici un outil pour illustrer et étudier des concepts,
pas une fin en soi...

Pré-requis

Dans ce cours, les **pré-requis** sont **très limités** :

- connaissances **en programmation impérative**
  notions de variables, affectations, conditions, boucles...
- utilisation de **fonctions** (récursivité...)

Le plus important est **une curiosité pour la programmation**

$$\text{Programmer} = \begin{cases} \text{comprendre un problème} \\ \text{exprimer une solution dans un langage} \end{cases}$$

À l'inverse, **aucune notion exigée en OCaml !**

Projets

Détails :

- quatre sujets en ligne prochainement sur la page du cours
- mais vous pouvez proposer un sujet personnel
  (sous réserve que le travail nécessaire soit du même ordre)
- le projet peut être réalisé en binôme ou de manière individuelle
- faire votre choix avant le 25 avril 2017, et le faire connaître via :

  `xavier.rival@inria.fr`

  en fournissant toutes les informations (nom, prénom, projet choisi, binôme éventuel)
- à rendre pour le 31 mai 2017
- soutenances orales à partir du 7 juin 2017 (TBC)

Détails pratiques

Organisation du cours :

- **8 séances**
- **amphis** les lundi et certains mercredi après midis de 13h30 à 15h00
- **travaux pratiques** les mardi matins de 8h à 10h ou 10h15 à 12h15

Notation :

- **contrôle classant, le 07 / 06 / 2017**, de 9h à 12h
- **projet optionnel** (Nb : un projet INF 4xy exigé pour un PA d'info)
- **note finale** :

$$\mathrm{N} = \max\left(\mathrm{CC}, \frac{\mathrm{Projet} + 2\mathrm{CC}}{3}\right)$$

Outline

Présentation et historique

Un langage polyvalent :

- avant tout, **fonctionnel** et avec un **système de types forts**
- fournit également des traits de programmation **impérative** et **objet**

Bref historique :

- tout d'abord une étude théorique des machines abstraites :
  CAM = Categorical Abstract Machine, un interprète logique
  ML = un langage fonctionnel issu de la théorie de la preuve
- devient **Caml Light** en 1990 :
  ajout d'un interprète efficace, qui travaille sur du bytecode
- **OCaml** apparaît en 1996 :
  code natif, objets, système de modules, librairies...
- toujours des évolutions :
  système de types amélioré, support de la concurrence...

Applications

Qualités reconnues :

- une **sémantique bien définie**
- génère du **code compilé efficace**
- propice à l'écriture de code **clair** et **concis**

Utilisé dans de nombreux secteurs :

- **compilation** : Esterel technologies (ANSYS), Facebook (Hack, Flow...)
- **logique et preuve** : Coq
- **analyse de programmes** : AbsInt, Fasoo, TrustInSoft, Facebook Infer
- **finance** : Jane Street Capital, Bloomberg, Lexifi
- **sécurité** et **cryptographie** : Galois, Cryptosense
- **virtualisation** : Citrix Systems
- ...

Utilisation

Interprète `ocaml` :

- chaque déclaration est évaluée aussitôt qu'on la valide
- pratique pour apprendre et pour rechercher des bugs
- inutilisable pour un développement de taille même moyenne...

Plusieurs compilateurs :

- compilateur **bytecode** : `ocamlc`
  produit un langage interprété et portable
  portable : compatible d'une machine à une autre
  ressemble à ce que vous avez utilisé en Java
- compilateur **natif** : `ocamlopt`
  génère du code machine, efficace, mais non portable

Un programme OCaml

Une suite de déclarations et expressions :

- **déclaration de types** : définition d'un type personnalisé
- **déclaration de noms** : évalue une expression et en associe la valeur à un nom
- **expressions** : idem, mais sans donner de nom au résultat

Nous verrons chacune des ces constructions plus en détail dans quelques minutes...

```
(* Declaration de types *)
type t = int (* declaration d'un type entier *)

(* Declaration de noms *)
let x = 5 (* declaration d'un nom "x", valeur 5 *)
let b = true (* idem, pour "b", valeur true *)

(* Expression *)
x + 2 ;;
```

Expressions

Les **expressions** sont les éléments les plus communs du langage...
Elles jouent le même rôle que les **expressions** et les **instructions** Java.
Leur **évaluation** produit une **valeur**.

- **constantes** : `12`, `2.7`, `true`, `'c'`, `"abc"`...
- **noms** : `x`, `b`, `sin`, `List.hd`...
- **opérateurs binaires** : `e0 + e1` (somme entière), `e0 *. e1` (produit flottant), `e0 ^ e1` (concaténation), `e0 = e1` (test d'égalité)
- **conditionnelle** : `if e0 then e1 else e2`
  (nb : en java, c'est une instruction)
- et bien d'autres, que nous verrons bientôt...

Extrait du toplevel (chaque ligne commence par #) :

```
# let x = 25 ;;
# if x < 18 then x + 2 else x - 2 ;;
- : int = 23
```

Déclarations

Une **déclaration** attache **une valeur** à **un nom**

- **Syntaxe** : `let x = e0 in e1` (portée **locale** dans `e1`)
- **Sémantique** : **évalue** `e0`, puis **attache** la valeur à `x` et **évalue** `e1`
- Déclarations **globales** : pas de `in` (portée **globale**)

Un nom n'est pas la même chose qu'une variable

- on peut **modifier** la valeur d'une **variable**
- on **ne peut pas** le faire dans le cas d'un **nom**
- par contre, on peut définir **plusieurs noms** associés à la même chaîne de caractères, et dans ce cas les règles de scoping usuelles s'appliquent

```
let x =
  let a = true in
  if a then
    let x = 3 in x + 2
  else 8
```

Types (voir cours 2...)

OCaml repose sur un **système de types** :

- les programmes **qui ne peuvent être typés** sont **rejetés** avant interprétation ou compilation
- le compilateur **calcule lui-même** la plupart des types

Comparaison avec Java :

- comme OCaml, Java utilise un système de types et n'accepte que des programmes bien typés
- mais Java ne calcule pas les types, il faut donner le type des variables

```
# let x = 56 ;;
val x : int = 56
# let y = 3.2 ;;
val y : float = 3.2
# let b = y < x ;;
Error: This expression has type int but an expression
    was expected of type float
```

Définition d'une fonction et application (voir cours 3...)

Les **fonctions** jouent un rôle très important en OCaml

- une **fonction** est une **expression** comme une autre
- opérations :
  définition : `fun x -> corps_de_la_fonction`
  application : `fonction argument` (pas de parenthèse)

```
# let f = fun x -> x + 1 ;; (* expr. "fonction" *)
val f : int -> int = <fun>
# let g x = x - 2 ;;          (* contraction *)
val g : int -> int = <fun>
# let y = g (f 8) ;;
val y : int = 7
```

Il y a beaucoup à apprendre sur les fonctions en OCaml.
Plus d'informations sur les fonctions **lors du cours 3** !

Définition d'une fonction récursive

Par défaut une définition de fonction est vue **comme non récursive** :

```
# let f x = if x = 0 then 1 else x * f (x - 1);;
Error: Unbound value f
```

Noter que l'erreur vient du fait que le nom `f` est inconnu...

Pour déclarer une fonction récursive, **ajouter le mot clé `rec`** :

```
# let rec f x = if x = 0 then 1 else x * f (x - 1);;
val f : int -> int = <fun>
# f 4 ;;
- : int = 24
# f (-1) ;;
Stack overflow during evaluation (looping recursion?).
```

Les appels de fonctions sont implémentés à l'aide d'une pile (comme en Java)

Et les variables dans tout cela ?

Nous avons vu que les **noms** sont **immutables** :

- une fois définis on **ne peut les modifier**...
- ... mais on peut toujours **définir un nouveau nom** (et écraser l'ancien)

Comment retrouver les variables classiques ?

Références

- `ref v` définit une **cellule mutable** contenant la valeur `v`
- on peut assigner une référence à un nom : `let x = ref v`
- `!` permet de lire une référence
- `:=` permet de modifier une référence

```
let x = ref 3 ;;
x := !x + 2 ;;    (* modifie x, ne renvoie rien *)
!x ;; (* lit le contenu de x *)
```

Effets de bords

Regardons à nouveau les types relevés par l'interprète OCaml :

```
# let x = ref 3 ;;
val x : int ref = {contents = 3}
# x := !x + 2 ;;    (* modifie x, ne renvoie rien *)
- : unit = ()
# !x ;; (* lit le contenu de x *)
- : int = 5
```

On note que := modifie l'état mais ne produit pas de valeur

Effet de bord

Modification irréversible de l'état courant :

- **écriture dans une référence** via `e0 := e1`
- **affichage sur la sortie standard** via `Printf.printf`
- **envoie de données sur un canal** (vers un fichier, le réseau...)

Effets de bords et séquences

En général, l'**évaluation d'une expression**

1. **effectue un effet de bord**
2. **renvoie un résultat**, possiblement vide (`()` de type `unit`)

Programmes impératifs : suite d'effets de bords

Programmes fonctionnels purs : pas d'effet de bords

Séquence

- `e0; e1` évalue l'expression `e0` puis l'expression `e1`
- `e0` attendue de type `unit`...
- `e1` peut renvoyer un résultat ou non

```
let x = ref 10 and y = ref 8 in
x := !x + !y;
y := !y - !x;
!x + !y
```

Paires et tableaux

Paires

- `e0, e1` construit une **paire immutable** (se généralise aux **n-uplets**)

```
let sum_vect (x0, x1) (y0, y1) = x0 + y0, x1 + y1 ;;
sum_vect (1, 0) (2, 3) ;;
```

Tableaux

- structures **mutables** créées soit explicitement `[| 1; 2; 3 |]`, soit à partir d'un élément `Array.make 8 true`
- lecture d'une cellule `e0.(e1)`
- écriture dans une cellule `e0.(e1) <- e2`

```
let sum_vect a0 a1 =
  let a = Array.make 2 0 in
  a.(0) <- a0.(0)+a1.(0); a.(1) <- a0.(1)+a1.(1); a
```

Boucles

Constructions habituelles, avec le même comportement qu'en Java ou C :

- **itération avec compteur** :

```
for i = e0 to e1 do expr done
```

- **itération contrôlée par une condition** :

```
while condition do expr done
do expr while condition
```

Nous avons vu deux constructions itératives :

- **les fonctions récursives** : courantes en programmation fonctionnelle
- **les boucles** : courantes en programmation impérative
  reposent presque toujours sur effet de bords et références

```
let init_i a =
  for i = 0 to Array.length a - 1 do
    a.(i) <- i;
  done
```

Exceptions

- comme en Java : interrompent l'exécution + branchement non local
- **définition** : `exception E of t (*t: type *)` ou `exception E`
- **lancement** : `raise (E x)` ou `raise E`
- **rattrapage** : `try e0 with E v -> e1` ou `try e0 with E -> e1`

Utiles pour **les cas d'erreurs** ou **les branchements non locaux** :

```
exception Found of int
exception Not_found
let find_first_zero a =
  try
    for i = 0 to Array.length a - 1 do
      if a.(i) = 0 then raise (Found i)
    done;
    raise Not_found
  with Found pos -> pos
```

Quelques types utiles

Nous verrons la notion de **types** en détail au prochain amphi...
En attendant, voici quelques types utiles...

Types de base :

entiers	`int`	`53`
booléens	`bool`	`true`, `false`
flottants	`float`	`3.1415`
caractères	`char`	`'a'`
chaînes de caractères	`string`	`"abc"`

Types composés (nous en verrons d'autres prochainement) :

paires	`t0 * t1`	`(v0, v1)`
tableaux	`t array`	`[| v0; v1; v2 |]`
références	`t ref`	`ref v`

Des outils et librairies utiles

De **nombreux outils** sont disponibles :

- `opam` : gestionnaire de paquets
  la plupart des éléments ci-dessous peuvent être installés via `opam`
- `utop` : un toplevel / interprète plus joli
- `obuild` : compilation de projets multi-fichiers
  une sorte de `make` pour OCaml
- `ocamllex` (standard) et `ocamlyacc` (standard) ou `menhir` (via `opam`) : génération de lexeurs et parseurs

Et aussi de **nombreuses librairies** :

- `camomile` : support pour UTF-8
- `csv` : lecture et écriture de fichiers CSV
- ...

Environnement de développement et ressources utiles

Organisation du travail :

- écrire le code dans un fichier séparé :-)
- le **compiler** ou le **faire accepter par l'interprète**
  $\Rightarrow$ permet de vérifier les types (i.e., le programme a t'il un sens ?)
- **tester** à tous les niveaux (fonctions de base, programme complet...)

Note : l'interprète `ocaml` est très basique et utilisable seulement pour des programmes très simples...

- on préférera compiler...
- ou à défaut utiliser `utop` : permet naviguer dans le code et les sessions

Autres ressources utiles

Le cours n'a bien sûr pas vocation à répéter la documentation du langage...

Pour plus d'informations :

- Le **poly** se concentre principalement sur le cours, mais explique aussi les constructions OCaml les plus importantes
- **Documentation en ligne :**
  `http://www.ocaml.org`
- Excellent ouvrage **"Real World OCaml"**, aux Éditions O'Reilly
  `https://realworldocaml.org/v1/en/html/index.html`
- Et aussi, en français, **Apprendre à Programmer avec OCaml**, Sylvain Conchon, Jean-Christophe Filliâtre