Cours numéro 1: introduction du module

LI213 – Types et Structures de données

Christophe Gonzales – Pierre-Henri Wuillemin

Licence d'Informatique – Université Paris 6

Informations administratives

- 11 semaines de TD/TME.
- Pour chaque TME, compte-rendu par formulaire ouèbe.
- Contrôle continu = partiel.
- Note finale = 40% contrôle continu + 60% examen final.

Note finale = 60% examen + 40% partiel

- page ouèbe du module :
  `http ://www-desir.lip6.fr/~gonzales/teaching/li213-2008oct`
  contient les énoncés, les formulaires de soumission, l'emploi du temps, etc.

Objectif du module

Programmes informatiques :

$\Longrightarrow$ traitements complexes sur des données complexes

$\Longrightarrow$ nécessité d'utiliser des algorithmes efficaces

$\Longrightarrow$ nécessité d'utiliser une «bonne» représentation des données

But du module «Types et Structures»

1. présenter les structures de données les plus classiquement utilisées en informatique.
2. faire appréhender les situations dans lesquelles telle ou telle structure est plus adaptée qu'une autre.

Plan du cours (1/2)

Partie I : Types en OCAML

cours 1	Problématique et motivation du cours Principes & types de données fondamentaux.
cours 2	**Programmation fonctionnelle** : Environnement, fonctions, fonctions récursives.
cours 3	**Programmation impérative** : Tableaux, procédure, séquence, boucles, références. Mécanisme des exceptions.
cours 4	Polymorphisme, filtrage, Type «produit», enregistrements
cours 5	Les listes, filtrage sur les listes.
cours 6	Les types «sommes»

Plan du cours (2/2)

Partie II : structures de données pour la programmation

cours 7	Les différents types de listes.
cours 8	Les piles (LIFO, FIFO, etc).
cours 9 & 10	Les arbres : arbres binaires, de recherche, équilibrés (AVL), les tas.
cours 11	Les tables de hachage.

En pratique

Langage d'application : OCAML

Pourquoi ce langage ?

- Langage fonctionnel fortement typé :
  toute variable/identifiant a un et un seul type
- Programmation impérative accessible
- Pas de transtypage implicite :
  (float) (1/3) vaut 0 en Java, C, C++, etc
- Programmation opérationnelle industriellement
- Environnement de compilation sympathique :
  toplevel, compilation en bytecode ou en natif
- L'équipe enseignante adore ce langage

OCAML, langage opérationnel ?

des logiciels en OCAML

- FFTW : transformée de Fourier rapide – puissant optimiseur symbolique qui, étant donné un entier *N*, produit du code C hautement optimisé pour effectuer des *Discrete FTs* de taille *N*. FFTW a reçu en 1999 le prix Wilkinson pour les logiciels de calcul numérique.
- Unison : synchroniseur de fichiers
- MLdonkey : client pair-à-pair
- Astrée : analyseur statique de code C – a prouvé l'absence d'erreurs d'exécution dans le logiciel de contrôle primaire de la famille Airbus A340.

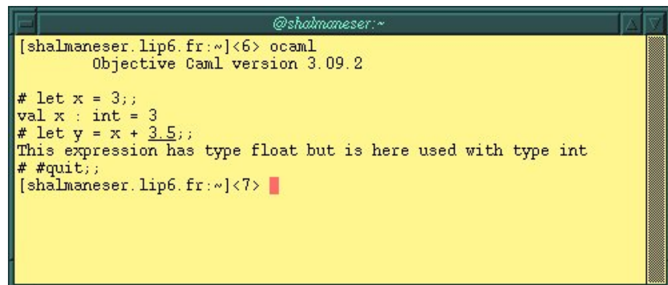
*des utilisateurs d'*OCAML

- CNRS / INRIA
  COQ : vérification et génération de preuves mathématiques, etc. . .
- Microsoft
  SLAM : vérification et intégrité dans les interactions noyau-drivers
  F# : (presque) OCAML dans l'environnement .NET
- Dassault : calculs scientifiques répartis
- LexiFi : modélisation et gestion de produits financiers complexes
- MLstate : serveurs web spécialisés, machine learning et data mining

Environnement de programmation (1/2)

Le Toplevel

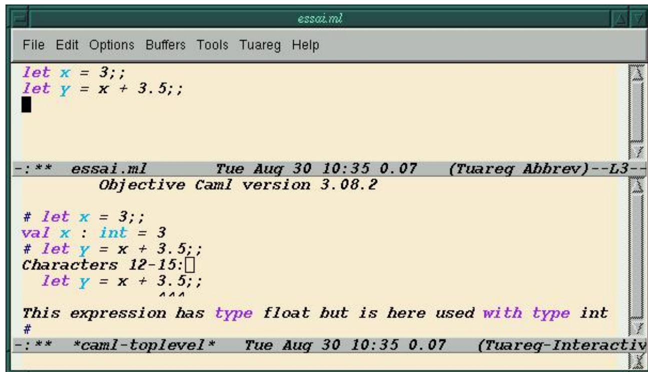
- se lance à partir d'une console avec la commande `ocaml`
- permet de saisir au clavier des expressions (prompt = #)
- compile ces expressions (après les ; ;)



Environnement de programmation (2/2)

Programmation sous emacs/xemacs

- permet d'éditer/sauvegarder des expressions
- mode `tuareg` exécute un toplevel dans emacs
- indentation et coloration syntaxique



Présentation du langage : les notations

Conventions dans les transparents :

1. On se place toujours dans le *Toplevel*
2. En noir : les instructions entrées par le programmeur
3. En bleu : les affichages du toplevel

Exemple

```
# let x = 3;;
val x : int = 3
# x + 5;;
- : int = 8
```

$\Longrightarrow$ chaque expression a un type

$\Longrightarrow$ langage fortement typé

OCaml type lui-même les expressions

Présentation du langage : les nombres

Les entiers : comment les désigner

10	-50	entiers exprimés en décimal
0XFF	0xf2	(zéro x) entiers exprimés en hexadécimal
0o77	0O22	(zéro o) entiers exprimés en octal
0b110	0B10	(zéro b) entiers exprimés en binaire

Les réels : comment les désigner

1.5	-6.	le «.» indique que ce sont des `float`
6.2e20	0.4E-22	utilisation d'un exposant

Règle : `int` = pas de «.», `float` = un «.»

Présentation du langage : opérations sur les nombres

- *Opérateurs sur les entiers :* +, -, *, / (division euclidienne), mod (modulo)
- *Opérateurs sur les réels :* +., -., *., /., ** (puissance)

Règle : les opérateurs ne s'appliquent qu'à un seul type de donnée

$\Longrightarrow$ `(0.1 + 0.7) * 10` provoque une erreur de compilation
Explication : pas de transtypage implicite
$\Longrightarrow$ évite d'obtenir l'entier 7 comme en C, C++, Java, Php, etc

Avantage : le programmeur connaît **précisément** les opérations effectuées : (1 / 3) * 3 = 0 car opérations sur les entiers.

Présentation du langage : transtypage sur les nombres

Fonctions de transtypage explicite

- int_of_float : renvoie le nombre sans sa partie décimale

```
# int_of_float 3.4;;
- : int = 3
# int_of_float (-3.4);;
- : int = -3
```

- float_of_int : renvoie le float correspondant à l'entier

```
# float_of_int 7;;
- : float = 7.
```

```
# float_of_int (7 / 3) *. 3.;;
- : float = 6.
```

Présentation du langage : les caractères et les chaînes

Les caractères (type char)

- ′caractère′ exemples : `′a′`, `′B′`
- caractères particuliers :

\n	newline	\t	tabulation
\′	apostrophe	\\	antislash

Les chaînes de caractères (type string)

- "chaîne" exemples : `"xxxx"`, `"chaîne"`
- caractères particuliers dans les chaînes :

\n	newline	\t	tabulation
\"	guillemet	\\	antislash

- ^ = opérateur de concaténation : `"xxx" ^ "yyy"`

Présentation du langage : transtypage

Fonctions de transtypage explicite

- `int_of_char` : renvoie le code ASCII d'un caractère

```
# int_of_char 'a';;
```

⇒ `- : int = 97`

- `char_of_int` : la fonction inverse

```
# char_of_int 97;;
```

⇒ `- : char = 'a'`

- `int_of_string` : renvoie l'entier représenté par une chaîne

```
# int_of_string "45";;
```

⇒ `- : int = 45`

- `string_of_int` : la fonction inverse

```
# string_of_int 45;;
```

⇒ `- : string = "45"`

- idem pour les float : `float_of_string` et `string_of_float`

Quelques exemples

```
# "La valeur décimale du nombre binaire 110001"
  ^ " est égale à " ^ (string_of_int 0b110001);;
```

```
- : string = "La valeur décimale du nombre
binaire 110001 est égale à 49"
```

Règle : le `;;` provoque la compilation de l'expression

```
# "La valeur entière de 234.56 est " ^
  (string_of_int 234.56);;
```

```
This expression has type float but is here
used with type int
```

$\Longrightarrow$ erreur de typage à la compilation

En dehors du toplevel, les instructions ci-dessus ne provoquent aucun affichage : ce sont seulement des expressions dont la valeur est une chaîne de caractères.

Présentation du langage : autres types de base

Les booléens (type bool)

- valeurs `true` et `false`
- opérateurs : not, && (le et logique), || (le ou logique)
- opérateurs renvoyant un booléen : $<$, $>$, $<=$, $>=$
  test d'égalité : = et non == comme en C
  test d'inégalité : $<>$ et non != comme en C

le `and` n'est pas un opérateur sur les booléens

```
# false && true || not (3 > 4);;
- : bool = true
```

Le type unit

- une seule valeur : `()`
- correspond au `void` du C

La notion d'environnement (1/2)

Principe des langages fonctionnels

- Au début du programme, 1 environnement contient des liaisons entre des identifiants ($\approx$ noms de variables) et des valeurs.
- Les fonctions prennent des données initiales et produisent un résultat en sortie. Elles modifient l'environnement.
- Programme = l'application de composées de fonctions sur un ensemble de données initiales : $f \circ g \circ \cdots \circ h$ (données)
programme $\approx$ l'application d'une fonction en maths

Notations

- (nom,valeur) = liaison entre l'identifiant «nom» et sa valeur
- $\lfloor$liaisons$\rfloor$ = l'environnement

Exemple $\lfloor$`(x,4),(y,3),(z,"toto"),(myvar,3.45)`$\rfloor$

La notion d'environnement (2/2)

La valeur d'une expression est calculée en cherchant les valeurs dans l'environnement de la dernière liaison entrée vers la plus ancienne.

Convention dans le cours : Les liaisons sont rajoutées sur la gauche $\Longrightarrow$ on parcourt l'environnement de la gauche vers la droite

Exemple :
Soit l'environnement $\lfloor$ `(x,4),(y,5),(x,6.5),(z,7)` $\rfloor$
alors l'expression `x + y * 2` vaut $4 + 5 \times 2 = 14$

Note : on peut avoir plusieurs fois des liaisons avec le même nom dans l'environnement. Les liaisons peuvent alors porter sur différents types (ici `int` et `float` pour `x`)

Modification de l'environnement

rajouter une liaison dans l'environnement

let nom = val ; ;

1. évaluation de val d'après la valeur de l'environnement
2. l'environnement est modifié en lui rajoutant la liaison `(nom,val)` sur sa gauche

Exemples

Soit l'environnement ⌊ `(x,4),...` ⌋

```
# let y = "toto";;
y : string = "toto"
```

⇒ environnement = ⌊ `(y,"toto"),(x,4),...` ⌋

```
# let x = 4.5 +. (float_of_int x);;
x : float = 8.5
```

⇒ environnement = ⌊ `(x,8.5),(y,"toto"),(x,4),...` ⌋

Les expressions sont toujours évaluées dans le dernier env

Modifications simultanées de l'environnement

rajouter plusieurs liaisons en même temps

let nom1 = val1 and nom2 = val2 and ... and nomk = valk ; ;

1. évaluation de val1,val2,..., valk dans l'environnement courant
2. rajout des liaisons de nom1, nom2, etc

Exemple

Soit l'environnement ⌊(x,4),(y,5),(z,6),...⌋

```
# let x = 3 and y = x + 2 and z = x + y;;
val x : int = 3
val y : int = 6
val z : int = 9
```

⇒ env =
⌊(x,3),(y,6),(z,9),(x,4),(y,5),(z,6),...⌋

Création d'environnements locaux

Environnement local pour évaluer une expression

let nom = val in expression ; ;

1. évaluation de val dans l'environnement courant E
2. création d'un environnement local E' = $\lfloor$ `(nom,val)` $\triangleleft$ `E` $\rfloor$
3. évaluation de l'expression dans E'
4. destruction de E'. On revient à E

Exemple

Soit l'environnement E = $\lfloor$ `(x,4),(y,5),...` $\rfloor$

```
# let x = x + 4 in y + x;;
```

⇒ `- : int = 13`

1. évaluation de `x + 4` dans E = 8
2. création de l'environnement E' = $\lfloor$ `(x,8)` $\triangleleft$ `E` $\rfloor$
3. évaluation de `y + x` dans E' = 13
4. destruction de E'

Résumé sur les environnements

Les différentes modifications de l'environnement

`let nom = val;;`	rajout de liaison
`let nom1 = val1` `and nom2 = val2;;`	liaisons simultanées
`let nom = val in expr;;`	liaison locale

on peut combiner ces différentes formes de déclarations

Exemple

Soit l'environnement E = ⌊ `(x,4),(y,5),...` ⌋

```
# let x = let x = x + 3 and y = x + 2 in x + y
  and y = x + 1;;
val x : int = 13
val y : int = 5
```

cf. détails sur les transparents suivants

Exemple de modification d'environnement (1/2)

Comment interpréter l'expression ci-dessous :

```
# let x = let x = x + 3 and y = x + 2 in x + y
  and y = x + 1;;
```

Indice : les sous-expressions sont toujours des `let ... in`
$\Longrightarrow$ chercher les `let ... in` les plus imbriqués
et les parenthéser.

```
# let x = let x = x + 3 and y = x + 2 in x + y
  and y = x + 1;;
```

d'où :

```
# let x = (let x = x + 3 and y = x + 2 in x + y)
  and y = x + 1;;
```

Exemple de modification d'environnement (2/2)

Au départ l'environnement E = $\lfloor$ `(x,4),(y,5)` $\rfloor$

```
# let x = (let x = x + 3 and y = x + 2 in x+y)
  and y = x + 1;;
```

Détails de l'évaluation de l'expression :

1. calcul de `x + 3` et `x + 2` dans E $\Rightarrow$ 7 et 6
2. création de l'env local E' = $\lfloor$ `(x,7),(y,6)` $\lhd$ `E` $\rfloor$
3. évaluation de `x + y` dans E' $\Rightarrow$ 13
4. destruction de l'environnement E'
5. évaluation de `x + 1` dans E $\Rightarrow 4 + 1 = 5$
6. rajout dans E des liaisons `(x,13)` et `(y,5)`

Le if ... then ... else

if ... then ... else

- `if` booléen `then` expr1 `else` expr2 `;;`
- c'est une expression :
  - sa valeur = valeur de expr1 si booléen = true, sinon valeur de expr2
  - son type : c'est l'intersection du type de expr1 et de expr2 $\Longrightarrow$ (pour l'instant) type de expr1 = type de expr2

Règle : ne jamais oublier le `else` expr2

Explication : pas de `else` expr2 $\Longrightarrow$ OCAML rajoute `else ()`
type de expr1 = type de expr2 $\Longrightarrow$ expr1 = `()`

```
# let x = if 1 < 2 then 10;;
This expression has type int but
is here used with type unit
```

Exemples de if ... then ... else (1/2)

Soit l'environnement E = ⌊`(x,4),(y,6),...`⌋

```
# let y = if y > 5 then x else y+1;;
```

```
val y : int = 4
```

⟹ env = ⌊`(y,4),(x,4),(y,6),...`⌋

```
# let y = if y > 5 then x else y+1;;
```

```
val y : int = 5
```

⟹ env = ⌊`(y,5),(y,4),(x,4),(y,6)`⌋

```
# let x =
    if y < 10 then
      if y > 5 then float_of_int (y - 5) /. 5.
      else (-1.)
    else 10.;;
```

```
val x : float = -1.
```

Exemples de if ... then ... else (2/2)

Soit l'environnement E = $\lfloor$ `(x,4),(y,6),...` $\rfloor$

```
# let x =
    if y < 10 then
      if y > 5 then float_of_int (y - 5) /. 5.
      else (-1)
    else ();;
```

Deux problèmes pour compiler cette expression :

- Règle «type de expr1 = type de expr2» $\Longrightarrow$ type de float_of_int (y - 5) /. 5. = float = type de (-1) = int
- Règle «toujours une else» $\Longrightarrow$ type de float_of_int (y - 5) /. 5. ET (-1) = type de ()