Présentation du cours

- contenu
  - des concepts
  - de la morale, du bon sens
  - OCaml, C, ...

- organisation
  - TPs assurés par Florent Bouchez, Aurélien Pardon et Julien Robert
  - modalités: 2 DM (programmation), un examen
  - vous êtes ??, découpez-vous en trois groupes de ??/3

Survol du cœur de Caml

- fonctions
- types

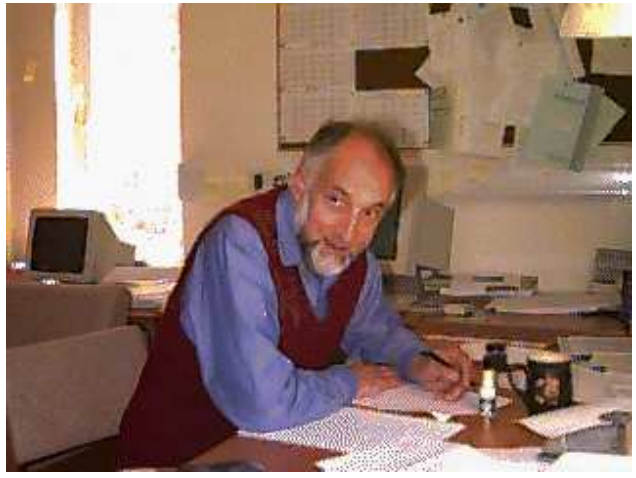
- types somme
- exceptions

- programmation impérative: références, effets de bord

- l'abstrait mais aussi le concret: mémoire, partage

Programmation Fonctionnelle

λ-calcul
Lisp, Scheme



ML: années 80, R. Milner

meta langage pour *LCF*

logic of computable functions

Haskell, Standard ML

Caml, Caml Light, Objective Caml

- projet Formel, puis Cristal, à l'INRIA
- postulat: *à l'oral, Caml = OCaml*
- `http://caml.inria.fr`
- Haskell OCaml Java Pascal C

OCaml, démarrage

- évaluation

```
# 32 + 52;;
- :   int =   84
```

type *valeur*

- la notion de *programme*

▷ on n'*exécute* pas un programme, on *évalue* une expression

▷ on n'écrit pas une suite d'instructions, on *déclare* des égalités

- expression $\rightsquigarrow$ $\begin{cases} \text{type} & (:) \\ \text{valeur} & (=) \end{cases}$

Valeurs et types

- évaluation, typage:
  le type affiché est-il le type avant ou après l'évaluation?
- construction à deux étages: $\dfrac{\text{type}}{\text{expression} \xrightarrow{\text{calcul}} \text{valeur}}$

  Théorème [réduction du sujet, 1è version]
  Si $P : T$ et $\text{eval}(P) = V$, alors $V : T$.

  fait le lien entre une analyse *statique* du programme (typage)
  et la *dynamique* du programme (évaluation)
- ce qui est bien, c'est moins ce que dit le théorème que la possibilité d'énoncer celui-ci
- lorsque le programme tourne, il n'y a plus de types
- $+$ en Caml, le système "devine" les types: *inférence de types*
  ($\neq$ Pascal, C, Java: indications à fournir)

Déclarations locales

`let id =` e_1 `in` e_2

1. évaluer $e_1 \rightsquigarrow v_1$
2. évaluer e_2 dans un *environnement* dans lequel id est associé à $v_1 \quad \rightsquigarrow \quad v_2$

- autre manière d'évaluer une telle expression?

évaluer e_2, et calculer e_1	à chaque fois que id apparaît la première fois que id apparaît

- le choix de la stratégie est-il influent?
  - ▷ **non** du point de vue du résultat *(sauf si effet de bord)*
  - ▷ **oui** du point de vue de l'implémentation / efficacité

Un exemple

```
let f g = if (g 3)>0 then g 32 else g 52
let t x = x*2-8
```

```
f t
= (fun g -> if (g 3)>0 then g 32 else g 52)  t
→ if (t 3)>0 then t 32 else t 52
= if ((fun x->x*2-8) 3)>0 then ((fun x->x*2-8) 32) else ((fun x->x*2-8) 52)
  soit
→→→ if (6*2-8>0) then 32*2-8 else 52*2-8
→→→→→→ if -2>0 then 76 else 96
→ 96
  soit
→ if (6*2-8>0) then ((fun x->x*2-8) 32) else ((fun x->x*2-8) 52)
→→→ (fun x->x*2-8) 52
→ 96
```

Plusieurs manières d'exécuter un programme?

- **Théorème [réduction du sujet, 1è version]**
  Si $P : T$ et $\text{eval}(P) = V$, alors $V : T$.
- a priori plusieurs manières de "faire évoluer" un programme P
  - on définit $P \longrightarrow P'$: en *une* étape de calcul, P devient P'
  - **Théorème [réduction du sujet, 2è version]**
    Si $P : T$, alors pour tout P' tel que $P \longrightarrow P'$, on a $P' : T$.
- le *résultat* de l'évaluation du programme P est V, ce que l'on note $\text{eval}(P) = V$, lorsque $P \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \ldots \longrightarrow V$ et $V \not\longrightarrow$
- l'unicité de ce résultat découle de la *confluence*: si $P \longrightarrow P_1$ et $P \longrightarrow P_2$, alors $(P_1 \longrightarrow \ldots \longrightarrow \leftarrow \cdots \leftarrow P_2)$
- a priori, plusieurs *stratégies d'évaluation*

  Caml et Haskell n'évaluent pas le `let..in` de la même manière
- la 2è version du thm. de réduction du sujet garantit que toutes les stratégies envisageables préservent le typage

Notion de portée

- `let x = e1 in e2`: hors de `e2`, `x` n'est pas connu:

  (en tout cas pas celui-là)

  - `e2` est la *portée* de la déclaration `x = e1`
  - l'*identificateur* `x` est *lié* à `e1` dans `e2`
- une liaison peut en cacher une autre: dans un `let a =...in`, on ne change pas la valeur de `a`, on "en fait un autre" *(les valeurs ne sont ni reprises ni échangées)*
- la construction fonctionnelle est liante aussi
  - dans `fun x -> e`, `x` est lié dans `e`
  - quid de `fun x -> fun x -> e` ?
- N.B.: les "`;;`" a top-level sont des "`in`"

  (les ;; sont superflus tant qu'on ne veut pas évaluer)
- portée *statique* (ou *lexicale*): *sens* d'une variable correspond à une zone du code $\neq$ *portée dynamique: correspond au moment de l'évaluation à l'exécution (LISP)*

```
let x = 1
let f () = x
let g () = let x = 2 in f()
```

`g ()` renverrait `2`

Les fonctions

- en Caml, une fonction est une entité 'de première classe'

  on utilise la même syntaxe pour passer un entier ou une fonction à une fonction

- type `t -> t'` valeur `<fun>` (`->` appartient au langage des types)
  - égalité et fonctions DÉMO `eq_fun.ml`
  - on n'a pas accès au code d'une fonction
- syntaxe
  - définition:

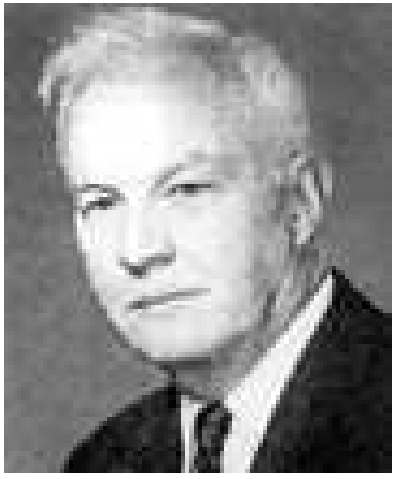
    `let f x =..` `let f = fun x ->..` `let f = function x ->..`

    fonctions récursives: `let rec f x = ...f machin...`

    récursivité mutuelle: `let rec f = ...and g = ...`
  - appliquer une fonction: `f x`, `f (x)`, `(f x)`
- une fonction non typable sans utiliser les types de base?

  `let rec f x = f f x`
- la flèche fige le calcul `fun x -> (print_string "bing"; x*x)`

Curryfication



Haskell Curry
(*Logique Combinatoire*)

- deux versions d'une fonction à deux arguments:
  - ▷ fonction qui renvoie une fonction `let f x y = ...`
    présentation curryfiée
  - ▷ fonction qui prend un couple `let f (x,y) = ...`
    présentation décurryfiée

→ toutes les fonctions ont $\boxed{1}$ argument (le λ est *monadique*)

- associativités: `a -> (b -> c)` et `(f x) y`

La pile à l'exécution

- un programme Caml qui s'exécute, c'est "le plus souvent" une fonction qu'on applique à un argument
- les appels aux sous-fonctions (et en particulier les appels récursifs) s'*empilent*

  (ceci n'est pas propre à Caml)
- cas particulier: récursivité terminale: le **premier** appel récursif est la **dernière** chose que fait la fonction

  DÉMO `rec_term.ml`

Faisons le point

- le mécanisme central est l'évaluation des programmes (expressions)
- les fonctions sont des objets comme les autres
- avant de commencer à calculer, on détermine le type d'une expression
- début de lexique
  type expression portée valeur programme évaluation
- mal typé = qui plante? qui boucle?
  c'est mal de boucler?

Importance des types en ML

- types de base

  `int bool float char string`

- une structure “rigide”

  pas de relation entre types: pas de coercion (*cast*)

  `float: int -> float, chr: int -> char`

- pas de surcharge des opérateurs: `+` et `+.`
- opérations prédéfinies
  - “prélude” librairie `Pervasives`

    N.B.: position des opérateurs: préfixe, infixe, postfixe

    parenthésage possible des opérateurs infixes `(+), (-),...`
  - librairies: faire `List.iter`, ou `open List` et après `iter`

Types de base et types produit

- les booléens – type à deux éléments: `true` et `false`
  un type à un seul élément? `():  unit`
- types produit: couples, tuples

```
type coup = char * int;;
```

le mot-clef `type` permet de définir des abréviations

(attention à la redéfinition: DÉMO `redef.ml`*)*

Enregistrements (*records*)

- ce sont des *produits avec champs nommés:*

```
type coup = { col :  char ; lig :  int };;
let c = { col = 'E'; lig = 2 };;
let x = ...c.col ...;;
```

- différence avec les structures de C:
  on ne peut modifier en place *(par défaut)*

```
# let deplace x = { col = x.col; lig = x.lig+2 };;
val deplace :  coup -> coup = <fun>
```

Polymorphisme

- types polymorphes:

```
let id x = x          ⇝        val id : 'a -> 'a = <fun>
let fst = fun (x,y) -> x        ⇝        'a * 'b -> 'a
let f x = [];;                  ⇝        'a -> 'b list
```

- `'a` = *n'importe quel type que l'on appelera* `'a`

  (ou aussi: *soit* `'a` *un type, . . .*)

  `'a` (‘*quote a*’) *variable de type*

- on *ne peut pas*

  donner le type `'a -> 'a` à `fun x -> x+1`

- statut de `list`, `option`, `*`: **constructeurs de types**

```
type 'a option = None | Some of 'a
```

les constructeurs de types sont des “fonctions vers les types”

Les types comme spécification

- avoir un système de types contraignant a une influence sur le cycle de développement des programmes

  davantage d'erreurs à la compilation, moins à l'exécution

- plus le système de types est riche, plus un type 'raconte ce que fait le programme'

  à la limite, deux langages: un de programmation, un de spécification

- exemple: chercher dans les bibliothèques

la documentation de Standard ML:

```
val foldr :
('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

la documentation d'OCaml:

```
val fold_right :
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Isomorphismes de types

- définition: $T_1 \leftrightarrows T_2$ *ssi il existe* f *et* g *tels que* $g \circ f = Id_{T_1}$ *et* $f \circ g = Id_{T_2}$
- les isomorphismes de types de ML (partie fonctionnelle)

$$
\begin{array}{rcl}
T_1 * T_2 & \leftrightarrows & T_2 * T_1 \\
T_1 * (T_2 * T_3) & \leftrightarrows & (T_1 * T_2) * T_3 \\
T_1 \rightarrow (T_2 * T_3) & \leftrightarrows & (T_1 \rightarrow T_2) * (T_1 \rightarrow T_3) \\
(T_1 * T_2) \rightarrow T_3 & \leftrightarrows & T_1 \rightarrow (T_2 \rightarrow T_3) \\
T_1 * \texttt{unit} & \leftrightarrows & T_1 \\
T_1 \rightarrow \texttt{unit} & \leftrightarrows & \texttt{unit} \\
\texttt{unit} \rightarrow T_1 & \leftrightarrows & T_1
\end{array}
$$

+ 3 autres lois avec les `'a` `'b`*...*

- à noter que l'on est en train de faire des manipulations dans le langage des types