Avant de démarrer

- groupes de TP
- pas cours de FDI ce matin
- je vous dois un TP de C/ un TP de Caml

Présentation du cours

- contenu
  - des concepts
  - de la morale, du bon sens
  - OCaml, C, ...
- organisation
  - TPs assurés par Guillaume MELQUIOND et Damien POUS
  - modalités: 2 DM (programmation), un examen
  - vous êtes ??, découpez-vous en deux groupes de ??/2

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 $\rightsquigarrow$ { type (:) ; valeur (=) }

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 = e1 in e2`

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

- 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

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 $P \to P'$: en *une* étape de calcul, P devient P'
  - **Théorème [réduction du sujet, 2è version]**
    Si $P : T$, alors pour tout P' tel que $P \to P'$, on a $P' : T$.
- questions qui se posent
  - qu'est ce que le résultat du calcul?
    $\text{eval}(P) = V$ lorsque $P \to P_1 \to P_2 \to \ldots \to V$ et $V \not\to$
  - si $P \to P_1$ et $P \to P_2$, il y a *confluence* ($P_1 \to \ldots \to \leftarrow \cdots \leftarrow P_2$)
- un langage de programmation $\leftrightarrow$ une stratégie
  - deux manières d'exécuter le `let...in`: Caml / Haskell
  - le théorème vaut pour *toutes les stratégies*
    - 1è version: Caml *ou* Haskell
    - 2è version: "tout langage fonctionnel"

Notion de portée

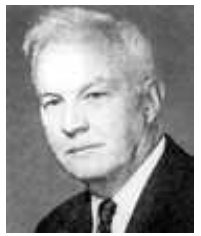
- `let x = e1 in e2`: hors de `e2`, `x` n'est pas connu:
  (en tout cas pas celui-là)
  - `e2` est la *portée* de la déclaration `x = e1`
  - l'*identificateur* `x` est *lié* à `e1` dans `e2`
- portée statique: *sens* d'une variable correspond à une zone du code

  ≠ portée dynamique: correspond au moment de l'évaluation à l'exécution (LISP)
- une liaison peut en cacher une autre (conflit de variables):
  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)

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 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

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`

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)

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

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

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