Au-delà du noyau de Caml

- types somme (variants)
- exceptions

Construire des ensembles

- `type bla = A of int | B of bool`

  `bla` est défini comme la *somme disjointe* des ensembles $\mathbb{N}$ et $\mathbb{B}$

- `type ilist = Nil | Cons of int * ilist`

  `ilist` est le plus petit ensemble contenant `Nil` et clos par application de `Cons`

- la définition de types somme permet d'introduire de nouveaux types, que l'on peut voir comme des ensembles de *valeurs*

Calculer sur les types somme

```
type bla = A of int | B of bool
type ilist = Nil | Cons of int * ilist
```

- `A B Nil Cons` sont des *constructeurs* (majuscule au début)

  on *applique* un constructeur pour fabriquer une valeur appartenant à un type somme

  ```
  let f x = if x>0 then (A x) else (B false)
  ```

- pour calculer sur un objet appartenant à un type somme, on *filtre* en se fondant sur une liste de *motifs*

  ```
  let g t = function
   |  A k -> string_of_int k
   |  B b -> if b then "vrai" else "faux"
  ```

  on exploite l'injectivité des constructeurs

- on applique un constructeur, **mais ce n'est pas une fonction**
  - la métaphore pour le constructeur: *l'objet en plastique 'clippé' aux vêtements dans une grande surface*

A l'usage: motifs

- le premier motif qui filtre est sélectionné
- "_" est le motif "universel" *(wildcard)*

  `| _ ->` *dans tous les autres cas*

  `let is_zero n = match n with | (Succ _) -> false | _ -> true`
- attention aux filtrages imbriqués (utiliser `begin...end`)
- `Warning: this pattern-matching is not exhaustive.`

  et `Uncaught exception: Match_failure`
- linéarité des variables: `| Add (e,e) -> ...` interdit

Autres motifs

- destructurer tout en nommant le tout: `as`

```
let f = function
| (a,b,c,d) as q -> if a<0 then (0,0,0,0) else q
```

- contraintes ajoutées: `when`

```
let f = function
| (a,b,c,d) when (a<0) -> (0,0,0,0)
| q -> q
```

- tout cela n'est que sympathie (commodité d'écriture) de la part de Caml

  le 'vrai' filtrage, c'est un cas par constructeur

Filtrage – des exemples

```
 # let f = function x when x = x -> true;;
Warning:  Bad style, all clauses in this pattern-matching are guarded.
val f :  'a -> bool = <fun>
# let _::_::l = [1;3;5;7;9];;
Warning:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val l :  int list = [5; 7; 9]
# let l1@l2 = l;;
Syntax error
# let [x;y] = l;;
Warning:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Exception:  Match_failure ("", 2, -13).
```

DÉMO some_trees.ml

Constructeurs de types

- certaines définitions de types sont *paramétriques* en un type `'a`
  - `type 'a option = None | Some of 'a`
- idem pour `list`
  - du fait de leur ubiquité, les listes bénéficient d'une syntaxe particulière
    - constructeurs: `[]` et `::` infixe ('nil' et 'cons')
    - notation `[3;2;5;2]`
    - filtrage sur les listes: `[]`, `x::xs`, et aussi `[x]`, `[x;y]`,...
    - pourquoi ne peut-on pas écrire `match l with | l1@l2 -> ...` ?
  - une liste n'est pas un tableau, c'est une pile (moyen d'accès privilégié)
  - librairie `List` (`map`, `flatten`, `fold_left`, `fold_right`,...)
- statut de `list`, `option`, `*`: *constructeurs de types*

Organisation

- ASR + Algèbre?

Rappels

- OCaml est un langage de programmation
  - déclaratif, exécuter un programme c'est *évaluer* une expression
  - fonctionnel: *"tout est fonction"*
- variables, phénomène de liaison
  - fonctions et déclarations locales

    `fun x -> e`, `let x = e1 in e2`
  - filtrage
    - on compare un motif m (contenant éventuellement des variables) avec un terme t (une expression)
    - cela engendre une *substitution* σ telle que $\sigma(m) = t$
      Exemple: `(Succ k)` avec `(Succ (Succ (Succ 0)))`
      donne $\sigma = \{$`k`$\rightsquigarrow$`(Succ (Succ 0))`$\}$
    - lorsqu'un motif *filtre*, on enrichit l'environnement d'évaluation avec de nouvelles liaisons
    - d'ailleurs, la construction `function m -> ..` fait *toujours* du filtrage (dans `function x -> x+1`, on utilise le motif 'trivial' `x`)

Fonctions partielles

`head []` ?? `tail []` ??

utiliser le type somme `option`:

```
let head = function
| x::_ -> Some x
| [] -> None
```

- ▷ pour: présentation "mathématique", uniformité du calcul
- ▷ contre: on passe son temps à "empaqueter et dépaqueter"

```
let t0 = head l in
let t = (match t0 with | Some x -> x | _ -> 0) in ...
```

ce qu'on veut, c'est dire qu'il y a quelque chose qui cloche

et "renvoyer une erreur"

Types somme, toujours

- un type somme *extensible*: `exn`
- ajouter un constructeur: `exception Paf`

```
exception Paf
exception Pif
let f b = if b then Paf else Pif
val f :  bool -> exn = <fun>

   raise :  exn -> 'a
```

- `raise` a pour effet de court-circuiter le flot 'normal' des appels de fonctions
- `try...with`: se rétablir

exemple: recherche dans une liste, un arbre DÉMO `excep.ml`

Exceptions et typage

Rappel: *réduction du sujet*

$$\frac{\text{type}}{\text{expression} \xrightarrow{\text{calcul}} \text{valeur}}$$

"si c'est bien typé avant d'exécuter, il n'y aura pas de conflit de type à l'exécution"

```
let boum x = raise Paf;;   boum : 'a -> 'b = <fun>
let rec loop x = loop x;;   loop : 'a -> 'b = <fun>
```

$\longrightarrow$ quel est le problème?

une fonction $\mathcal{F}$ de type `'a -> 'b` permet de convertir n'importe quoi en n'importe quoi: $3+\underbrace{\mathcal{F}(\texttt{"hop"})}_{\texttt{bool?}} \longrightarrow \texttt{??}$

. . . mais `boum` et `loop` ne produisent pas de valeur: *ouf*

Exceptions en Java

- Java: langage pour la programmation *orientée objet*
  - *"tout est objet"*, les *objets* s'obtiennent à partir de *classes*
  - mécanisme de calcul: appeler une méthode au sein d'un objet (en Caml: passer un argument à une fonction)
- il y a des objets exception (qui héritent de la classe `Throwable`)

`raise`	`throw`
`try...with`	`try...catch`

Flot d'exécution

- dans d'autres langages, notion de flot d'exécution

  control flow *(or "flow of control"): The sequence of execution of instructions in a program. This is determined at run-time by the input data and by the control structures (e.g. "if" statements) used in the program.* → *"par où passe le calcul"*

- en Fortran, Basic (et C)

```
      ⋮
goto 911
      ⋮
911:  ...
```

- en C

  `return 0;` sort de la fonction
  `exit(0);` sort du programme (↔ `return 0;` dans le `main`)
  `break;` arrête l'instruction `for` courante (idem `while`, `do`)
  `continue;` passe à l'itération suivante dans un `for`, `while`, `do`

  toutes ces instructions "dévient" le flot

- N.B.: le saut conditionnel ("`ifgoto`") est volontiers naturel en assembleur