Sommes et exceptions

Types sommes

exemples

Types sommes, exemples

- définition
```
type bst = Noeud of int * bst * bst | Feuille
let singleton = fun k -> Noeud(k,Feuille,Feuille)
let arbre = Noeud(34, singleton 12, singleton 45)
```
- utilisation
```
let rec taille = function
 | Feuille -> 0
 | Noeud (_,a1,a2) -> taille a1 + taille a2 + 1

let rec taille a acc = match a with
 | Feuille -> acc
 | Noeud (_,a1,a2) -> taille a1 (taille a2 (acc+1))
```
- DÉMO `some_trees.ml`

à regarder chez vous

Les listes en OCaml

- ▷ un cas particulier de type somme, tellement fréquent qu'il est *'cablé en dur'* dans le langage (tradition des langages déclaratifs)
- ▷ liste vide: `[]` ajout d'un élément en tête: `::`
  constructeurs "nil" et "cons"
- ▷ une liste n'est pas un tableau, c'est *une pile*
  (moyen d'accès privilégié)
- ▷ concaténer deux listes: `@`

pourquoi ne peut-on pas écrire `match l with | l1@l2 -> ...` ?

- ▷ filtrage sur les listes: `[]`, `x::xs`, et aussi `[x]`, `[x;y]`,...
- ▷ librairie `List` (`map`, `flatten`, `fold_left`, `fold_right`,...)

Constructeurs, application

- `Feuille`, `Noeud` sont des *constructeurs*
- application d'une fonction $\neq$ application d'un constructeur
  - application d'une fonction: ça calcule
    `(fun x -> x^x) "cou"`
  - application d'un constructeur: on "plaque" le constructeur
    `Noeud (3,Feuille,Feuille)`
- `type toto = ...| C of t |...`

  `C` envoie de manière univoque un `t` dans `toto`
  `C` est une injection de `t` dans `toto`

(un peu) formellement

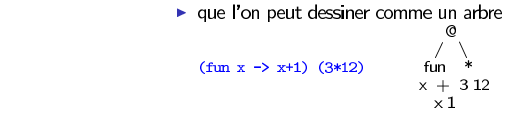
Définition d'un type somme

- un type peut être vu comme un *ensemble de valeurs*
  - un type somme emblématique: les entiers de Peano
    `type nat = Zero | Succ of nat`
  - à un type somme correspond une *définition inductive*
    - les entiers naturels sont le plus petit ensemble contenant 0 et clos par successeur
    - les listes... nil... cons
    - ...
- mécanisme de filtrage:
  - les constructeurs décrivent les éléments d'un ensemble de valeurs de manière *exhaustive* et *univoque*
  - lien de parenté avec la preuve par récurrence

Expressions, termes, valeurs

- une *expression* (un programme) est représentée par un terme
  - que l'on peut dessiner comme un arbre

`(fun x -> x+1) (3*12)`



- une *valeur* est une expression qu'on ne peut plus réduire

  valeurs $\subseteq$ expressions
  - `2, true, "coucou", [], ...`
  - `fun x -> 3*2*x`
  - `Noeud(3,Feuille,Feuille)`

Le filtrage, plus formellement

- lors du filtrage, on compare une *valeur* avec un "terme avec des trous", que l'on appelle un *motif*
  - valeur: `Noeud(12, Noeud(3,Feuille,Feuille), Feuille)`
  - motif: `Noeud(k,a1,a2)`
    - les "trous" sont des *variables*: `k`, `a1`, `a2`
    - *variable* ici signifie 'inconnue'
      $\neq$ variable dans un programme ('variable x initialisée à 0')
- lorsque le filtrage réussit, il engendre une *substitution*
  - $k \leftarrow 12$, $a1 \leftarrow$ `Noeud(3,Feuille,Feuille)`, $a2 \leftarrow$ `Feuille`
  - substitution: fonction des variables vers les valeurs
- lorsque l'on écrit `function x->x+1`, on utilise le motif 'trivial' `x`

Types somme, suite

rappels

- les constructeurs sont des injections vers le type somme que l'on définit
- appliquer un constructeur $\neq$ appliquer une fonction
- pour filtrer, on compare une valeur avec un motif
- lexique

termes — variables (*variable liée*) — motifs — substitution

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` (fonction non totale)
- filtrage "implicite" avec les produits

Autres motifs

- linéarité des variables: `| Add (e,e) -> ...` interdit
- 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
```

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).
```

Pragmatiquement (cf. plus tard)

manipulation intensive des types somme

```
let li = [3;2;52]
let li' = List.tl li
let li'' = List.map (fun x -> 2*x) li'

let rec somme acc = function
 | [] -> acc
 | x::xs -> somme (acc+x) xs
```

beaucoup d'opérations ont lieu 'sous le tapis'

Un modèle "lisse"

- jusqu'ici tout va bien:
  - toutes les fonctions renvoient un résultat
  - l'ordre dans lequel on fait les calculs n'importe pas
  - on n'a pas à "voir" la mémoire
- mais il y a parfois besoin de s'écarter d'une vision si uniforme
  - fonctions non totales
  - questions d'efficacité
  - données à portée globale
  - "réalisme"

Fonctions partielles

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

- utiliser le type somme `option`

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

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"*

Lever et rattraper des exceptions

- 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 (un saut dans la pile des appels)
- `try...with`: rattraper une exception
- exemple: recherche dans une liste, un arbre

DÉMO `excep.ml`

Revenons à la sûreté du typage

- rappel: *subject reduction*

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

si e a le type t et s'évalue en v, alors v a le type t

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

→ quel est le problème?

- une fonction $\mathcal{F}$ de type `'a -> 'b` permet de convertir n'importe quoi en n'importe quoi: $3+\underbrace{\mathcal{F}(2)}_{\texttt{bool}} \rightarrow ??$
- ...mais `boum` et `loop` ne produisent pas de valeur: *ouf*

Exceptions dans d'autres langages

- ça, c'était le point de vue OCaml sur les exceptions
- en JAVA: *objets* exception (qui héritent de la classe `Throwable`), proches dans le principe

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

- 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"*

Exceptions dans d'autres langages, suite

- 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