Université de Bordeaux
Année 2018-2019

Mastère 2
T.A.P

TD n°5 - Types variants

▷ Les *types variants* sont des types de données OCaml permettant de représenter à l'intérieur d'un même type des valeurs correspondant à des cas disjoints. Ils s'écrivent avec la syntaxe suivante :

```
type ⟨nom_type⟩ =
  | ⟨Constr_1⟩ of ⟨Type_1⟩
  | ⟨Constr_2⟩ of ⟨Type_2⟩
    ...
  | ⟨Constr_n⟩ of ⟨Type_n⟩
```

Exemple :

```
type value =
  | Int of int
  | Float of float;;
```

Pour utiliser les types variants, on a recours à un procédé appelé *filtrage* :

```
match ⟨expr⟩ with
  | ⟨p_1⟩ →  ⟨expr_1⟩
  | ⟨p_2⟩ →  ⟨expr_2⟩
    ...
  | ⟨p_n⟩ →  ⟨expr_n⟩
```

Exemple :

```
let string_of_value x =
 match x with
  | Int i   → string_of_int i;
  | Float f → string_of_float f;;
```

Noter que ce procédé permet de filtrer des expressions non triviales comme des listes (par exemple `x::c`), des n-uplets (par exemple `(a,b,c)`) ou bien simplement des expressions constantes (par exemple `42`).

Exercice 1: Types variants / Filtrage

Dans cet exercice, on introduit les *types variants* en OCaml.

1. Écrire un type variant `weight` permettant de manipuler des poids en kilos, en livres et en carats.
2. Écrire les fonctions de traduction d'un élément de type `weight` en kilos. (*Indice culturel :* 1kg = 2.205 lbs = 5000 carats)

Le compilateur sait vérifier si un filtrage est exhaustif ou pas.

3. Pour vérifier cela, tester l'exemple suivant et le corriger :

```
let sum_of_values a b = match (a,b) with
  | (Int i,Int j)     → Int(i+j)
  | (Int i,Float f)   → Float((float i)+.f)
  | (Float f,Float g) → Float (f+.g);;
```

4. Écrire un type variant permettant de manipuler des pièces de monnaie de 1, 7 et 13 zlotys. Écrire un alorithme glouton qui, étant donné un nombre de zlotys n, renvoie une liste de pièces (en zlotys) dont la somme des valeurs est n.
5. Écrire un type variant permettant de regrouper les types réels et complexes. Inclure ce type dans un autre type variant permettant de manipuler les solutions des équations des trinômes du second degré, ainsi que la fonction de résolution.

Exercice 2: Arbres binaires et plus

Considérons un type variant générique `'a bintree` permettant de représenter des arbres binaires. Ce type est fourni avec une fonction `bintree_build` permettant de construire un arbre binaire de hauteur donnée, étant donné :

— une valeur à la racine et

— une fonction `f : 'a → ('a * 'a)` qui, étant donnée la valeur stockée dans un noeud, fournit le couple des valeurs stockées dans les deux noeuds fils.

```
type 'a bintree =
| BinEmpty
| BinNode of 'a * 'a bintree * 'a bintree

let rec bintree_build f h x =
  if (h <= 0)
  then BinEmpty
  else let (x1,x2) = f(x) in
          BinNode (x,
            (bintree_build f (h-1) x1),
            (bintree_build f (h-1) x2));;
```

1. Tester `bintree_build` en utilisant la fonction de génération `let f x = (2*x,2*x+1)`.
2. Écrire une fonction `bintree_map` qui permet d'appliquer uniformément une fonction `f` à tous les éléments d'un arbre binaire.

Dans le cas des arbres binaires ordonnés, les éléments de l'arbre sont des entiers, et étant donné un noeud x, tous les éléments stockés dans le sous-arbre gauche sont inférieurs à x, tous les éléments stockés dans le sous-arbre droit lui sont supérieurs.

3. Écrire une fonction `bintree_insert` permettant d'insérer un nombre entier dans un arbre de manière à préserver la propriété d'ordre.
4. Dans une représentation des arbres avec des pointeurs, comment pourrait-on implémenter cette fonction en créant un minimum de nouveaux objets (et dont réutilisant les valeurs passées en paramètre) ?
5. Proposer une fonction `bintree_fold` effectuant un pliage sur un arbre binaire.

Plaçons nous maintenant dans le cas où les noeuds ne possèdent plus un nombre fixé, mais un nombre arbitraire de fils.

6. Écrire un type adapté à de tels arbres, et construire la fonction `tree_build` associée.
7. Écrire un itérateur `tree_map` permettant d'appliquer une fonction `f` uniformément sur tous les éléments de l'arbre.
8. Comment faire pour que cette fonction soit récursive terminale ?

▷ Le langage Typescript (`https://www.typescriptlang.org`) est un langage de programmation développé par Microsoft, basé sur l'idée d'étendre le langage Javascript en ajoutant des outils de vérification statique. Pour programmer en Typescript, il faut utiliser les outils suivants :

- Compilation : utiliser `tsc` pour transformer un fichier `.ts` en un fichier `.js` :

  `tsc file.ts`

- Exécution : utiliser la machine virtuelle `node` pour exécuter le code :

  `node file.js`

Voici un exemple très simple de programme Typescript :

```
let a : number = 7;
let b : number = 6;
console.log("The␣answer␣is␣equal␣to␣" + a * b);
```

Pour développer en Typescript au CREMI, il est nécessaire de :

`source /net/cremi/renault/script`

Exercice 3: Interprète `BASIC`

Une utilisation habituelle des types variants consiste à représenter des grammaires, qui à leur tour permettent d'écrire des compilateurs. Nous proposons dans cet exercice de construire un interpréteur pour langage `BASIC` (`https://en.wikipedia.org/wiki/BASIC`), un langage conçu en 1964 avec l'idée de simplicité d'utilisation. Cet interpréteur sera écrit en Typescript, en utilisant la version locale des types variants appelés *discriminated unions* (cf. `https://www.typescriptlang.org/docs/handbook/advanced-types.html#discriminated-unions`).

Pour simplifier, nous allons alléger la grammaire utilisée. Dans un premier temps, les expressions du langage utilisent un certain nombre d'opérateurs possibles, et prennent la forme suivante :

```
// Operators
interface Plus  {  kind: "op_plus";  }
interface Minus {  kind: "op_minus";  }
interface Equal {  kind: "op_equal";  }
interface Mult  {  kind: "op_mult";  }
type Op = Plus | Minus | Equal | Mult;
```

```ts
// Expressions
interface ExprInt { kind: "expr_int", int: number }                    // Integer expression
interface ExprVar { kind: "expr_var", var: string }                    // Variable expression
interface ExprBin { kind: "expr_bin",  lhs: Expr,  op: Op, rhs : Expr } // Binary operation
type Expr = ExprInt | ExprVar | ExprBin;
```

Ici, on ne gère que des expressions numériques ou booléennes. Les valeurs gérées par l'environnement seront donc du type :

```ts
// Values
interface ValInt  { kind: "val_int", val: number }     // Integer value
interface ValBool { kind: "val_bool", val: boolean }  // Boolean value
type Value = ValInt | ValBool;
```

Les variables évoluent pendant l'exécution à l'intérieur d'un environnement, à savoir ici un dictionnaire :

```ts
type Env = { [id: string] : Value; }
```

1. Écrire une fonction `eval_expr` qui prend en argument une expression et un environnement et qui renvoie la valeur correspondant à cette expression (en remplaçant les variables par les valeurs qu'elles possèdent dans l'environnement).

La syntaxe des expressions du langage se résume à :

```ts
// Instructions
interface InstrGoto   { kind: "instr_goto", line: number }
interface InstrJumpIf { kind: "instr_jumpif", cond: Expr, line: number }
interface InstrLet    { kind: "instr_let", id: string, expr: Expr }
type Instr = InstrGoto | InstrJumpIf | InstrLet ;
```

Nous allons tenter d'écrire un mini-interpréteur de ce langage. Pour cela, nous allons utiliser les définitions suivantes pour les valeurs et l'environnement. Remarquer que les instructions conditionnelles `IF` sont obligatoirement suivies d'un `GOTO`. De plus, les lignes sont numérotées de 10 en 10. Le programme `BASIC` que nous cherchons à interpréter calcule la factorielle d'un entier :

```
interface Line { instr : Instr, line: number }
type Prog = Line[];
interface State { env: Env, line: number }

let prog : Prog = [
    { line: 10,
      instr: { kind: "instr_let", id: "i",
               expr: { kind: "expr_int", int: 1}} },
    { line: 20,
      instr: { kind: "instr_let", id: "s",
               expr: { kind: "expr_int", int: 1}} },
    { line: 30,
      instr: { kind: "instr_let", id: "i",
               expr: { kind: "expr_bin", op: { kind: "op_plus"},
                       lhs: { kind: "expr_var", var: "i"},
                       rhs: { kind: "expr_int", int: 1} }} },
    { line: 40,
      instr: { kind: "instr_let", id: "s",
               expr: { kind: "expr_bin", op: { kind: "op_mult"},
                       lhs: { kind: "expr_var", var: "i"},
                       rhs: { kind: "expr_var", var: "s"} }} },
    { line: 50,
      instr: { kind: "instr_jumpif",
               cond: { kind: "expr_bin", op: { kind: "op_equal" },
                       lhs: { kind: "expr_var", var: "i"},
                       rhs:  {kind: "expr_var", var: "n"}},
               line: 70} },
    { line: 60,
      instr: { kind: "instr_goto", line: 30} },
]
```

```
10 INPUT N
20 LET I = 1
30 LET S = 1
40 LET I = I + 1
50 LET S = S * I
60 IF (I = N) THEN GOTO 80
70 GOTO 40
80 PRINT S
```

2. Écrire une fonction `eval_instr` qui prend en argument une instruction et un état, et renvoie un nouvel état.
3. Écrire une fonction `eval_prog` qui prend en argument un programme, une ligne et un environnement, et renvoie l'environnement obtenu à la fin de l'exécution du programme.

Le résultat de l'évaluation avec $n = 4$ devrait produire un affichage proche de :

```
Evaluating line 10
Evaluating line 20
Evaluating line 30
Evaluating line 40
Evaluating line 50
{ n: { kind: 'val_int', val: 4 }, i: { kind: 'val_int', val: 2 }, s: { kind: 'val_int', val: 2 } }
Evaluating line 60
Evaluating line 30
Evaluating line 40
Evaluating line 50
{ n: { kind: 'val_int', val: 4 }, i: { kind: 'val_int', val: 3 }, s: { kind: 'val_int', val: 6 } }
Evaluating line 60
Evaluating line 30
Evaluating line 40
Evaluating line 50
{ n: { kind: 'val_int', val: 4 }, i: { kind: 'val_int', val: 4 }, s: { kind: 'val_int', val: 24 } }
Evaluating line 70
End of evaluation
{ n: { kind: 'val_int', val: 4 }, i: { kind: 'val_int', val: 4 }, s: { kind: 'val_int', val: 24 } }
```