FP3

December 2, 2024

1 Les Langages & leurs Interprétations

Un *langage* est simplement un ensemble de termes pouvant être définis récursivement par un type de données.

On parle en général d'arbre syntaxique (Abstract Syntax Tree - AST).

Le code suivant donne une illustration avec des valeurs et deux opérateurs:

[1]:
```
type Exp = Val of int | Add of Exp*Exp | Mul of Exp*Exp

let t, f = Val 1, Val 0

let e = Mul (t,Add (f,t))

let rec show = function
  | Val 0       -> "false"
  | Val 1       -> "true"
  | Add (e1,e2) -> "("+(show e1)+" | "+(show e2)+")"
  | Mul (e1,e2) -> "("+(show e1)+" & "+(show e2)+")"

show e
```

```
input.fsx (7,16)-(7,24) typecheck warning Incomplete pattern matches on this
expression. For example, the value 'Val (2)' may indicate a case not covered by
the pattern(s).
```

[1]:
```
(true & (false | true))
```

Une *inerprétation/évaluation* est alors une fonction qui retourne une valeur d'un certain type pour une expression, et plusieurs interprétations sont possibles.

[2]:
```
let min x y = if (x<y) then x else y
let max x y = if (x>=y) then x else y

let rec eval = function
  | Val v       -> v
  | Add (e1,e2) -> max (eval e1) (eval e2)
```

```
  | Mul (e1,e2) -> min (eval e1) (eval e2)

show (Val (eval e))
```

[2]: true

2 Application: Logique Descriptive

Comme expliqué dans le cours 1, un type peut être paramétré pour être rendu plus générique (réutilisable):

[82]:
```
type 't Exp = Val of 't | Add of ('t Exp)*('t Exp) | Mul of ('t Exp)*('t Exp)

let IR, ASE, A1, A2 = Val "IR", Val "ASE", Val "A1", Val "A2"
let (<|>) e1 e2 = Add (e1,e2)
let (<&>) e1 e2 = Mul (e1,e2)

let e = IR <&> A1
```

Comme expliqué aussi, il est possible de définir une fonction générique sur ce type:

[83]:
```
let rec interp v a m = function
  | Val x       -> v x
  | Add (e1,e2) -> a (interp v a m e1) (interp v a m e2)
  | Mul (e1,e2) -> m (interp v a m e1) (interp v a m e2)
```

De là, il reste à spécifier un domaine “sémantique” et le sens des différents éléments utilisés:

[84]:
```
let db = [("bob","ASE");("bob","A1");("kate","IR");("kate","A1");("max","IR");
 ↪("max","A2")]
```

Du cours 2, il est possible d’utiliser ce qui a été défini sur le type “dictionnaire” et les opérateurs union/intersection/différence vus dans le cours 1:

[85]:
```
let rec values key = function
  | []       -> []
  | (v,k)::kv -> if (k=key) then v::(values key kv) else (values key kv)

printfn "%A" (values "A1" db)
printfn "%A" (values "IR" db)
```

```
["bob"; "kate"]
["kate"; "max"]
```

[86]:
```
let rec isIn x = function
  | []    -> false
  | v::vs -> (x=v) || (isIn x vs)

let rec intersection xs ys =
```

```
  match xs with
  | []    -> []
  | v::vs -> if (isIn v ys) then v::(intersection vs ys) else (intersection vs␣
 ↪ys)

let rec union xs ys =
  match xs with
  | []    -> ys
  | v::vs -> if (isIn v ys) then (union vs ys) else v::(union vs ys)

let s1, s2 = ["bob"; "kate"], ["kate"; "max"]
printfn "%A\n%A" (union s1 s2) (intersection s1 s2)
```

```
["bob"; "kate"; "max"]
["kate"]
```

En mettant tous les éléments ensembles:

[88]:
```
let eval = interp (fun x -> values x db) union intersection

printfn "%A" (eval e)
```

```
["kate"]
```