FP3b

December 4, 2024

1 Rappel/compléments C2: définition de types

1. Type “linéaire”

[11]:
```
type 't List =
  | Nil
  | Cons of 't*('t List)

let l:int List = Cons (3,Cons (1,Cons(2,Nil)))
printfn "%A" l
```

```
Cons (3, Cons (1, Cons (2, Nil)))
```

2. Type “arborescent”

[12]:
```
type 't Tree =
  | Leaf
  | Node of 't*('t Tree)*('t Tree)

let t:int Tree = Node (2,Node(1,Leaf,Leaf),Node(3,Leaf,Leaf))
printfn "%A" t
```

```
Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Leaf))
```

3. Transformation “générique” (& isomorphisme)

a. Sur les listes

[26]:
```
let rec redlist c f l =
  match l with
  | Nil         -> c
  | Cons (v,l2) -> f v (redlist c f l2)

let list = redlist [] (fun v l -> v::l)

printfn "%A" (list l)
```

```
[3; 1; 2]
```

[27]:
```
let list1 = List.fold (fun l v -> Cons (v,l)) Nil
printfn "%A" (list1 [3;1;3])
```

```
Cons (3, Cons (1, Cons (3, Nil)))
```

b. Sur les arbres

[15]:
```
let rec redtree c f t =
  match t with
  | Leaf          -> c
  | Node (v,l,r) -> f v (redtree c f l) (redtree c f r)

let flatten = redtree [] (fun v l r -> l@[v]@r)
printfn "%A" (flatten t)
```

```
[1; 2; 3]
```

[17]:
```
let rec insert v t =
  match t with
  | Leaf           -> Node (v,Leaf,Leaf)
  | Node (v2,l,r) -> match (v<=v2) with
                     | true -> Node (v2,insert v l,r)
                     | _    -> Node (v2,l,insert v r)
printfn "%A" (insert 4 t)
```

```
Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Node (4, Leaf, Leaf)))
```

[22]:
```
let tree = redlist Leaf (fun v l -> insert v l)
printfn "%A" (tree l)
```

```
Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Leaf))
```

[28]:
```
printfn "%A" (flatten (tree (list1 [3;2;1])))
```

```
[1; 2; 3]
```

4. Types "synonymes"

a. Map (Dictionnaires)

[38]:
```
type Map<'k,'v> = ('k * 'v) list

let d = [("le","the");("homme","guy");("petit","small")]

let rec lookup k d =
  match d with
  | []         -> failwith "not found !"
  | (k2,v)::d2 -> match (k=k2) with
                  | true -> v
                  | _    -> lookup k d2

printfn "%A" (List.map (fun k->lookup k d) ["le";"petit";"homme"])
```

```
["the"; "small"; "guy"]
```

[39]:
```
type Map2<'k,'v> = ('k * 'v) Tree
```

b. Arbres n-aires

[46]:
```
type NTree<'t> = L of 't | N of 't*(list<NTree<'t>>)

let doc = N ("html",[N("body",[N("h1",[L "Welcome"]);N("p",[L "under␣
↪construct"])])])

let rec html doc =
  match doc with
  | L v      -> v
  | N (v,es) -> "<"+v+">"+(List.reduce (+) (List.map html es))+"</"+v+">"

printfn "%A" (html doc)
```

```
"<html><body><h1>Welcome</h1><p>under construct</p></body></html>"
```

2 Arbres "syntaxiques" (langages et interprétation)

Un *langage* est défini par une *grammaire* pouvant se représenter par un *type*.

Par exemple, représenter le *terme* `t` ci-dessous:

[48]:
```
type T<'t> =
  | Val of 't
  | Add of T<'t>*T<'t>
  | Mul of T<'t>*T<'t>

let t = Mul(Val 1,Add (Val 2,Val 3))
```

Ex1. Proposer une fonction d'évaluation (interprétation) pour calculer la valeur de `t`

[49]:
```
let rec eval = function
  | Val v       -> v
  | Add (t1,t2) -> (eval t1)+(eval t2)
  | Mul (t1,t2) -> (eval t1)*(eval t2)

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

```
5
```

Ex2. En fait, les valeurs 1,2,3 correspondent à des valeurs logiques 1=true, 2=false, 3=true et les opérateurs Add/Mul à la conjonction/disjonction. Comment définir cette nouvelle interprétation ?

[52]:
```
let ctx = [(1,true);(2,false);(3,true)]

let rec eval2 = function
  | Val v       -> lookup v ctx
  | Add (t1,t2) -> (eval2 t1) && (eval2 t2)
```

```
  | Mul (t1,t2) -> (eval2 t1) || (eval2 t2)

printfn "%A" (eval2 t)
```

```
true
```

Ex3. Définir une fonction "générique" permettant de définir plus simplement/rapidement eval/eval2.

[53]:
```
let rec interp v a m = function
  | Val x       -> v x
  | Add (t1,t2) -> a (interp v a m t1) (interp v a m t2)
  | Mul (t1,t2) -> m (interp v a m t1) (interp v a m t2)

let show = interp string (fun t1 t2->t1+"+"+t2) (fun t1 t2->t1+"*"+t2)
printfn "%A" (show t)
```

```
"1*2+3"
```

Ex4. On désire travailler en "logique floue" dans laquelle les valeurs de vérité sont des valeurs de probabilité comprises entre 0 et 1. Par exemple, "1" est vrai seulement à 70%, "2" à 40% et "3" à 60%. Les opérateurs Add/Mul correspondent alors au min/max.

Comment définir cette nouvelle interprétation ?

[57]:
```
let ctx = [(1,0.7);(2,0.4);(3,0.6)]

let eval3 = interp (fun v -> lookup v ctx) max min
printfn "%A" (eval3 t)
```

```
0.6
```

Ex5. On désire maintenant travailler en "logique ensembliste" et "1" correspond aux élèves de 1A:["bob","kate","bill"], "2" aux élèves de IR:["kate","max"] et "3" aux ASE:["john","bill"].

Comment interpréter alors `t` ?

[59]:
```
let ctx = [(1,"1A");(2,"IR");(3,"ASE")]

let eval4 = interp (fun v->lookup v ctx) (fun t1 t2 -> "("+t1+" or "+t2+")")␣
 ↪(fun t1 t2 -> "("+t1+" and "+t2+")")

printfn "%A" (eval4 t)
```

```
"(1A and (IR or ASE))"
```

Ex6. Comment trouver alors la valeur de cette expression ?

[63]:
```
let ctx = [(1,["bob";"kate";"bill"]);(2,["kate";"max"]);(3,["john";"bill"])]

let eval5 = interp (fun v->Set.ofList (lookup v ctx)) Set.union Set.intersect
printfn "%A" (eval5 t)
```

```
set ["bill"; "kate"]
```