Cours - Programmation Fonctionnelle

Louis Thevenet

Table des matières

1. Types abstraits

1. Il se situe dans le monde mathématique
2. On a une fonction d'abstraction AF : $C \to A$ avec C un type concret
   - AF n'est pas implémentable mais à documenter
   - Surjective
   - Pas forcément injective
3. Une opération concrète est correcte si, quand elle respecte ses éventuelles préconditions, elle commute avec AF sur RI (domaine de définition de AF)

RI Invariant de repérsentation (souvent implémentable)

```
module type INT_SET = sig
  type t

  val empty : t
  val member : int -> t -> bool
  val add : int -> t -> t
end

module NoDupList : INT_SET = struct
  type t = int list

  let check_rep l =
    if List.(length (sort_uniq Int.compare l) = length l) then l
    else failwith "RI"

  let empty = check_rep []
  let member n l = List.mem n l
  let add n l = check_rep (n :: check_rep l)
end
```

2. Types fantômes

2.1. Définition

Un type fantôme est un type paramétré :

1. dont au moins un des paramètres n'apparaît pas dans la définition des valeurs de ce type
2. dont la définition est abstraite par une signature

2.2. Utilité

- caractériser un état interne/caché (type-state) ⇝ plutôt pour du code impératif (ou monade d'état, cf. cours 6)
- ne pénalise pas l'exécution (zero cost abstraction)

```
module type FichierLecture1Car = sig
  type debut
  type fin
  type _ fichier

  val _open : string -> debut fichier
  val lecture : debut fichier -> char * fin fichier
  val close : fin fichier -> unit
end

module Impl : FichierLecture1Car = struct
  type debut
  type fin
  type _ fichier = in_channel

  let _open = open_in
  let lecture fichier = (input_char fichier, fichier)
  let close = close_in
end

(* let wrong = *)
(* let f = Impl._open "toto" in *)
(* let c, f = Impl.lecture f in *)
(* Impl.lecture f *)
```

3. Types non uniformes

3.1. Définition

Un type (récursif) non uniforme `'a t` fait apparaître des instances différentes du paramètre dans sa définition, fonctions de `'a`.

3.2. Exemples

- listes alternées:

```
type ('a, 'b) alt_list = | Nil | Cons of 'a * ('b, 'a) alt_list
```

- arbres binaires équilibrés:

```
type 'a perfect_tree = | Empty | Node of 'a * ('a * 'a) perfect_tree
```

3.3. Usage

- représenter des invariants de structure "descendants"
- meilleure spécification
- nécessite la "récursion polymorphe"

Il faut utiliser la **récursion polymorphe** par contre.

Par exemple :

```
let rec depth = function
| Empty -> 0
| Node (_, sub) -> 1 + depth sub;;
Error: This expression has type ('a * 'a) perfect_tree
but an expression was expected of type 'a perfect_tree
The type variable 'a occurs inside 'a * 'a

(* devient *)
(* chaque application de depth doit avoir un type universellement quantifié*)
(* littéralement : pour tout alpha, ce type, i.e. 'a . ('a perfect_tree)*)
let rec depth : 'a . 'a perfect_tree -> int= function
| Empty -> 0
| Node (_, sub) -> 1 + depth sub;;
val depth : 'a perfect_tree -> int = <fun>
```

4. Types algébriques généralisés (GADT)