Université Pierre & Marie Curie DESS – GLA

Programmation – Spécifications formelles
I. Spécification algébriques

P. Manoury

Nov. - Déc. *2001*

Librairie standard Objective Caml
Fichier d'interface `stack.mli`

```
(* This module implements stacks (LIFOs), with in-place modification. *)

type 'a t
        (* The type of stacks containing elements of type ['a]. *)

val create: unit -> 'a t
        (* Return a new stack, initially empty. *)
val push: 'a -> 'a t -> unit
        (* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t -> 'a
        (* [pop s] removes and returns the topmost element in stack [s],
           or raises [Empty] if the stack is empty. *)
val clear : 'a t -> unit
        (* Discard all elements from a stack. *)
val length: 'a t -> int
        (* Return the number of elements in a stack. *)
```

Les piles : spécification algébriques

Sort: `stack`

Uses: `int, elt`

Symbols:

```
create :  → stack
push : elt, stack → stack
top : stack → elt
pop : stack → stack
clear : stack → stack
length : stack → int
```

Axioms: ∀ `s:stack; x:elt`

```
(top (push x s)) = x
(pop (push x s)) = s
(length create) = 0
(length (push x s)) = 1+(length s)
(length (clear s)) = 0
```

valeurs : `push x3 (push x2 (push x1 create))`

Librairie standard Objective Caml

Fichier d'implantation `stack.ml`

(ou presque)

```
(* $Id: stack.ml,v 1.6 2000/04/13 12:16:02 xleroy Exp $ *)
(* Patch par nos soins  *)

type 'a t = { mutable c : 'a list }

exception Empty

let create () = { c = [] }

let clear s = s.c <- []

let push x s = s.c <- x :: s.c; s

let pop s =
  match s.c with
    hd::tl -> s.c <- tl; s
  | []     -> raise Empty

let top s =
  match s.c with
    hd::_ -> hd
  | []     -> raise Empty

let length s = List.length s.c
```

Files d'attentes

spécification *vs* implantation

Sort: `queue`

Uses: `int, elt`

Symbols:

```
create :  → queue
add : elt, queue → queue
peek : queue → elt
rem : queue → queue
clear : queue → queue
length : queue → int
```

Axioms: ∀ `q:queue; x,y:elt;`

```
(take (add x create)) = x
(take (add x (add y q))) = (take (add y q))
(rem (add x create)) = create
(rem (add x (add y q))) = (add x (rem (add y q)))
(length create) = 0
(length (add x q)) = 1+(length q)
(length (clear q)) = 0
```

Files d’attentes
implantation I

```
type 'a queue = 'a list

let create() = []

let clear () = []

let add x q = x::q

let rec take q =
 match q with
   [] -> failwith "Empty"
 | [x] -> x
 | _::q -> take q

let rec rem q =
 match q with
   [] -> failwith "Empty"
 | [x] -> []
 | x::q -> x::(rem q)

let length = List.length
```

Files d'attentes
implantation II

```
type 'a queue = 'a list

let create() = []

let clear () = []

let add x q = q@[x]

let take q =
 match q with
   [] -> failwith "Empty"
 | x::_ -> x

let rem q =
 match q with
   [] -> failwith "Empty"
 | _::q -> q

let length = List.length
```

Files d'attentes

implantation III

d'aprés Objective Caml/stdlib

```
type 'a queue_cell =
    Nil
  | Cons of 'a * 'a queue_cell ref

type 'a t =
  { mutable head: 'a queue_cell;
    mutable tail: 'a queue_cell }

let create () = { head = Nil; tail = Nil }

let clear q = q.head <- Nil; q.tail <- Nil

let add x q =
  match q.tail with
    Nil (* if tail = Nil then head = Nil *)
    -> let c = Cons(x, ref Nil) in q.head <- c; q.tail <- c
  | Cons(_, newtailref)
    -> let c = Cons(x, ref Nil) in newtailref := c; q.tail <- c

let take q =
  match q.head with
    Nil -> raise Empty
  | Cons(x, _) -> x

let rem q =
  match q.head with
    Nil -> raise Empty
  | Cons(x, rest) ->
      q.head <- !rest;
      (match !rest with Nil -> q.tail <- Nil |  _  -> ()):
      q

let rec length_aux = function
    Nil -> 0
  | Cons(_, rest) -> succ (length_aux !rest)

let length q = length_aux q.head
```

Structure de donnée impérative

module `Array`

```
val length : 'a array -> int

      Return the length (number of elements) of the given array.

val get: 'a array -> int -> 'a

      Array.get a n returns the element number n of array a. The first
      element has number 0. The last element has number Array.length a
      - 1. Raise Invalid_argument "Array.get" if n is outside the
      range 0 to (Array.length a - 1). You can also write a.(n)
      instead of Array.get a n.

val set: 'a array -> int -> 'a -> unit

      Array.set a n x modifies array a in place, replacing element
      number n with x. Raise Invalid_argument "Array.set" if n is
      outside the range 0 to Array.length a - 1. You can also write
      a.(n) <- x instead of Array.set a n x.

val make: int -> 'a -> 'a array
val create: int -> 'a -> 'a array

      Array.make n x returns a fresh array of length n, initialized
      with x. All the elements of this new array are initially
      physically equal to x (in the sense of the ==
      predicate). Consequently, if x is mutable, it is shared among
      all elements of the array, and modifying x through one of the
      array entries will modify all other entries at the same time.
```

valeurs : `set (set (set (make 3) 0 x1) 1 x2) 2 x3`

Vecteurs

équations conditionnelles

Sort: `vect`

Uses: `int, elt`

Symbols:

`length` : `vect` → `int`
`get` : `vect, int` → `elt`
`set` : `vect, int, elt` → `vect`
`make` : `int, elt` → `vect`

Axioms: ∀ `v:vect; e:elt; n,i,j:int`

`(length (make n e))` = `n`

`(length (set v i e))` = `(length v)`

`(0` ≤ `i), (i` < `n)` ⇒ `(get (make n e) i)` = `e`

`(0` ≤ `i), (i` < `(length v))` ⇒ `(get (set v i e) i)` = `e`

`(0` ≤ `i), (i` < `(length v)), (0` ≤ `j), (j` < `(length v)), (i` ≠ `j)`
⇒ `(get (set v i e) j)` = `(get v j)`

Spécification abstraite d'une valeur
l'élément maximal (le plus petit indice de)

Abréviation : `(indom i v)` $\hat{=}$ `(0 ≤ i) & (i < (length v))`

```
Extends: vect
Assume:
  (<) : elt, elt → bool
  (≤) : elt, elt → bool
with: ∀ e,e1,e2,e3:elt
  (e < e) = true
  (e1 < e2) ⇒ (e2 < e1) = false
  (e1 < e2), (e2 < e3) ⇒ (e1 < e3) = true
  (e1 ≤ e2) = (not (e2 < e1))
Symbols:
  imax : vect → int

Axioms: ∀ v:vect; i,m:int; e:elt
  (indom (imax v) v) = true
  (indom i v) ⇒ (get v i) ≤ (get v (imax v))
  (indom i v), ((get v (imax v)) = (get v i)) ⇒ (imax v) ≤ i
```

Calcul d'une valeur

définition récursive et correction

```
Extends: vect
Symbols:
  loop : int, int, vect → int
  find_imax : vect → int
Axioms: ∀ v:vect; i,m:int; e:elt
  (loop m (length v) v) = m
  (indom i v), (indom m v), (get(v,m) <get(v,i))
    ⇒ (loop m i v) = (loop i i+1 v)
  (indom i v), (indom m v), (get(v,i) ≤ get(v,m))
    ⇒ (loop m i v) = (loop m i+1 v)
  ((length v)≠ 0) ⇒ (find_max v) = (loop 0 1 v)
```

Correction : `((length v)≠ 0) ⇒ (find_max v) = (imax v)`

Programme pour `find_max`

implantation fonctionnelle

Implantation littérale :

```
let rec loop m i v =
 if i = Array.length v then m
 else if v.(m) < v.(i) then loop i (i+1) v
 else loop m (i+1) v

let find\_max v =
 if Array.length v = 0 then raise Not_found
 else loop 0 1 v
```

Optimisation :

```
let find_max v =
 let len = Array.length v in
 let rec loop m i =
  if i = len then m
  else loop (if v.(m) < v.(i) then i else m) (i+1)
 in
  if len = 0 then raise Not_found else loop 0 1
```

Correction d'un calcul

L'invariant

```
Extends: vect
Symbols:
  bimax : int → vect → int

Axioms: ∀ v:vect; i,j,m:int
  (0 <i) ⇒ (indom (bimax i v) v) = true
  (0 ≤ j), (j <i) ⇒ (get v j) ≤ (get v (bimax i v))
  (0 ≤ j), (j <i), ((get v (bimax i v)) = (get v j))
    ⇒ (bimax i v) ≤ j
```

Lemme : ∀ `v:vect; i,m:int ((length v)≠ 0), (indom i v), (m = (bimax i v))`
`⇒ (loop m i v) = (imax v)`

```
let find_max v =
 let len = Array.length v in
  if len = 0 then raise Not_found
  else begin
   let m = ref 0 in
   let i = ref 1 in                     (* -- !m = (bimax !i v)              *)
    while !i < len do                   (* -- (!i < len) & (!m = (bimax !i v)) *)
     if v.(!m) < v.(!i) then m := i;    (* -- !m = (bimax !i+1 v)            *)
     incr i                             (* -- !m = (bimax !i v)              *)
    done;                               (* -- (!i = len) & (!m = (bimax !i v)) *)
    !m                                  (* -- !m = (imax v)                  *)
  end ;;
```