15-150 Spring 2018
Lecture 16

Modular Programming

Signatures and Structures
modularity principle

- A large program should be organized as a collection of small components or units
  - more manageable size
  - easier to maintain
- Give an interface for each unit
  - other units should rely only on interface
- This will facilitate separate development
language support

• Signatures
  • interfaces

• Structures
  • implementations

• Functors
  • ways to combine structures...
signatures

A signature is a collection of type names and value names, each with a specified type

```
signature SIGNAME =
sig
  · type names
  · and value headers
  ·
end
```
Implementing Queues
Queue

First-in-first-out (FIFO) data structure

We will use a signature to describe its interface
Signature for a Queue

signature QUEUE =
sig
  type 'a queue (* abstract type *)

  val empty : 'a queue
  val enq : 'a queue * 'a -> 'a queue
  val null : 'a queue -> bool
  val deq : 'a queue -> 'a * 'a queue

  exception Empty
  (* deq (q) raises Empty if q is empty *)

end
Two implementations

- Use a single list
  - Elements stored in arrival order: oldest element at the front, youngest at the end

- Use a pair of lists
  - We will see how to use them to have a more efficient implementation
Abstraction function (first implementation)

The list consists of the elements currently in the queue in arrival order (so oldest element at the head of the list, youngest at the end).
First implementation

structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list

  val empty = nil

  fun enq (q,x) = q @ [x]

  fun null (nil) = true
  | null ( _ ) = false

  exception Empty

  (* deq (q) raises Empty if q is empty *)
  val deq : 'a queue -> 'a * 'a queue
end
Signature ascription

```
structure Queue1 : QUEUE =
 struct
   type 'a queue = 'a list

   val empty = nil

   fun enq (q,x) = q @ [x]

   fun null (nil) = true
     | null ( _ ) = false

 exception Empty

   fun deq (x::q) = (x,q)
     | deq (nil) = raise Empty
end
```

Transparent ascription:
user could say 1::Queue.empty and get back [1]

Is that a good thing?
Abstraction function
(second implementation)

Lists front and back such that front @ (rev back) consists of the elements currently in the queue in arrival order (so oldest element at the head of the list, youngest at the end).
Second implementation

**structure** Queue2 : QUEUE =

```
struct
  type 'a queue = 'a list * 'a list

  val empty = (nil, nil)

  fun enq ((front, back), x) = (front, x::back)

  fun null (nil,nil) = true
  | null _ = false

  exception Empty

  (* deq (q) raises Empty if q is empty *)
  val deq : 'a queue -> 'a * 'a queue
end
```
With opaque ascription

**structure** Queue2 : > QUEUE =

```plaintext
struct
  type 'a queue = 'a list * 'a list

  val empty = (nil, nil)
  fun enq ((front, back), x) = (front, x::back)
  fun null (nil,nil) = true
    | null _ = false

  exception Empty

  (* deq (q) raises Empty if q is empty *)
  val deq :'a queue -> 'a * 'a queue

  fun deq (x::front, back) = (x, (front, back))
    | deq (nil, nil) = raise Empty
    | deq (nil, back) = deq(rev back, nil)

end
```
Dictionaries

signature DICT =

sig

  type key = string (* concrete type *)
  type 'a entry = key * 'a (* concrete type *)

  type 'a dict (* abstract type *)

  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict

end
signature DICT =
sig
  type key = string          (* concrete type *)
  type 'a entry = key * 'a  (* concrete type *)
  type 'a dict               (* abstract type *)
val empty : 'a dict
val lookup : 'a dict -> key -> 'a option
val insert : 'a dict * 'a entry -> 'a dict
end

structure BinarySearchTree : DICT =
struct
  type key = string
  type 'a entry = key * 'a

datatype 'a tree = Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree
val empty = Empty
fun lookup tree key =
  let fun lk (Empty) = NONE
       | lk (Node(left, (key1,value1), right)) =
         (case String.compare(key,key1)
          of EQUAL => SOME(value1)
           | LESS => lk left
           | GREATER => lk right)
  in
   lk tree
  end
fun insert (tree, entry as (key,_)) =
  let fun ins (Empty) = Node(Empty, entry, Empty)
       | ins (Node(left, entry1 as (key1,_), right)) =
         (case String.compare(key,key1)
          of EQUAL => Node(left, entry, right)
           | LESS => Node(ins left, entry1, right)
           | GREATER => Node(left, entry1, ins right))
  in
   ins tree
  end
end

(*
Abstraction function:
The collection of entries in the binary tree constitutes
the dictionary.

Representation Invariant:
The tree is sorted:
  For every node  Node(left, (key1,value1), right),
every key in left is LESS than key1, and
every key in right is GREATER than key1.
*)
Exam practice
(* COMPOSE : (int -> int) list -> int -> int
   REQUIRES: true
   ENSURES : COMPOSE [f_1, ..., f_n] == f_1 ◦ ... ◦ f_n
     COMPOSE [] == (fn x => x)
*)

val COMPOSE : (int -> int) list -> int -> int = foldr (op ◦) (fn x => x)