Modules

Big-time-functional programming

15-150 M21

Lecture 0712
12 July 2021

0 Packaging Data Together

```
Int.toString : int -> string
Int.compare : int * int -> order
Int.min : int * int -> int
Int.max : int * int -> int
Int.abs : int -> int
```

```
String.concat : string list -> string
String.concatWith :
    string -> string list -> string
String.implode : char list -> string
String.explode : string -> char list
String.isPrefix : string -> string -> bool
String.compare : string * string -> order
```

Other examples

```
List.null : 'a list -> bool
List.length : 'a list -> int
List.nth : 'a list * int -> 'a
List.rev : 'a list -> 'a list
```

```
ListPair.zip :
    'a list * 'b list -> ('a * 'b) list
ListPair.unzip :
    ('a * 'b) list -> 'a list * 'b list
```

```
Fn.id : 'a -> 'a
Fn.const : 'b -> 'a -> 'b
Fn.curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
Fn.uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

0712.0 (Crypto.sml)

```
structure Foo =
  struct
    datatype blah = A of int | B of string
    val k = 3
    exception Badness
    fun g 0 = 1
      | g n = 2 + g(n-1)
  end
```

Deep in the bowels of SML

```
structure Int = struct ... end

structure Bool = struct ... end

structure String = struct ... end

structure List = struct ... end

structure Fn = struct ... end
```

1 Signatures and Transparent Ascription

Today's Slogan:

There are some things the user *shouldn't* know

```
(* highly-secure pseudorandom number generator *)
Crypto.prng : int -> int
```

0712.1 (Crypto.sml)

```
structure Crypto =
struct
  fun calc1 n = 1013 +  (389 * n)
  fun calc2 n = n mod 1039
  val prng = calc2 o calc1
end
```

Fix your signature to it

0712.2 (Crypto.sml)

```
signature CRYPTO =
sig
  val prng : int -> int
end
```

0712.3 (Crypto.sml)

```
structure Crypto2 : CRYPTO =
struct
  fun calc1 n = 1013 +  (389 * n)
  fun calc2 n = n mod 1039
  val prng = calc2 o calc1
end
```

Can go in a signature:

- Types
- Datatypes
- Values
- Exceptions
- Structures

Notes:

- `type` declarations in the signature can be fulfilled by `datatypes` in the structure.
- If the `datatype` is given in the signature, it must be copied identically in the structure.
- `type`s can also be given *concretely* or *abstractly* in the signature.

Concrete

```
signature SIGCo =
  sig
     type t = int
  end
```

Abstract

```
signature SIGAb =
  sig
     type t
  end
```

```
signature SIGAb = sig type t end
```

```
structure StructName1 : SIGAb = struct
    type t = int
  end
```

Even if the type is left abstract in the signature, the user knows what it is implemented as.

```
structure StructName2 :> SIGAb= struct
    type t = int
  end
```

If the type is left abstract in the signature, the user *has no idea* what it is implemented as. Remember to give them a method for making values of that type!

0712.4 (Crypto.sml)

```
signature PRINTABLE =
sig    type t
       val toString : t -> string
end
structure IOP : PRINTABLE =
struct
  type t = int option
  fun toString NONE = "NONE"
     | toString (SOME x) =
        "SOME(" ^ (Int.toString x) ^ ")"
end
```

Check Your Understanding

Why do we want transparent in this case?

5-minute break

2 Opaque Ascription

Is there a type of natural numbers in SML?

0712.5 (Nat.sml)

```
fun fact (0 : nat) : nat = 1
  | fact n = n * fact(n-1)
```

```
type nat = int
```

0712.6 (Nat.sml)

```
signature NAT =
sig
  type nat
  val zero : nat
  val succ : nat -> nat
  val abs : int -> nat
  val asInt : nat -> int
  exception Negative
  val ? : int -> nat
end
```

0712.7 (Nat.sml)

```
structure Nat:>NAT =
struct
  type nat = int
  val zero = 0
  fun succ x = x + 1
  val abs = Int.abs
  val asInt = Fn.id
  exception Negative
  fun ? n =  if n<0 then raise Negative else n
end
```

0712.8 (Nat.sml)

```
fun smartfact (n : Nat.nat):int =
  let
    fun fact 0 = 1
      | fact (k:int):int = k * fact(k-1)
  in
    fact(Nat.asInt n)
  end
```

The `Nat` structure maintained the **invariant** that every integer which the user can obtain of type `Nat.nat` must be nonnegative.

By ascribing opaquely, the user isn't able to construct any values of the abstract type "on their own". So they must use the methods supplied by the structure, which can be guaranteed to maintain the invariant.

Queues are a sequential, first-in first-out data structure.

We can start out with an empty queue

```
>
```

Then *insert* or *enqueue* an element

```
> 3
```

Then another

```
> 3 4
```

Then if we *dequeue*, it returns the first one we put in

```
> 4
```

0712.9 (QUEUE.sig)

```
signature QUEUE =
sig
  type 'a queue
  val emp : 'a queue
  val ins : 'a * 'a queue -> 'a queue
  val rem : 'a queue -> ('a * 'a queue) option
end
```

Single-Stack

0712.10 (queue.sml)

```
structure LQ :> QUEUE =
struct
    type 'a queue = 'a list
    val emp = []
    fun ins (n,l) = l @ [n]
    fun rem [] = NONE
      | rem (y::ys) = SOME (y, ys)
end
```

Invariant `L:'a LQ.queue` lists the elements in the queue from first-in to last-in.

0712.11 (queue.sml)

```
structure LLQ :> QUEUE =
struct
  type 'a queue = ('a list) * ('a list)
  val emp = ([], [])
  fun ins (n,(front, back)) = (front,n::back)
  fun rem ([],[]) = NONE
    | rem (y::ys, back) = SOME (y,(ys,back))
    | rem ([], back) = rem (List.rev back,[])
end
```

Invariant If `(f,b):'a LLQ.queue`, then `f@(rev b)` lists the elements in the queue from first-in to last-in.

Outside scope of the course:

Amortized Analysis

Homework:

Prove that a user cannot tell the difference between `LQ` and `LLQ`

- Can use structures to package data together
- Can use signatures to provide interfaces for uses
- The modules system allows you to hide information behind abstraction boundaries

Next Time

- Typeclasses and algebraic structures
- Functors
- Sets

Thank you!