Signature review: collect *declarations*

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

    val empty : 'a queue
    val put : 'a * 'a queue -> 'a queue
    val get : 'a queue -> 'a * 'a queue  (* raises Empty *)

    (* LAWS: get(put(a, empty)) == (a, empty) *)
    ...
    (*)
end
Structure: collect definitions

structure Queue :> QUEUE = struct   (* opaque seal *)
    type 'a queue = 'a list
    exception Empty

    val empty = []
    fun put (x,q) = q @ [x]
    fun get [] = raise Empty
      | get (x :: xs) = (x, xs)

    (* LAWS: get(put(a, empty)) == (a, empty) *)
    ...
Dot notation to access components

fun single x = Queue.put (Queue.empty, x)
val _ = single : 'a -> 'a Queue.queue
What interface with what implementation?

Maybe mixed together, extracted by compiler!
  • CLU, Haskell

Maybe matched by name:
  • Modula-3, Modula-3, Ada

Best: any interface with any implementation:
  • Java, Standard ML

But: not “any”—only some matches are OK
Signature Matching

Well-formed

\[
\text{structure } \text{Queue} :> \text{QUEUE} = \text{QueueImpl}
\]

if principal signature of QueueImpl matches ascribed signature QUEUE:

- Every type in QUEUE is in QueueImpl
- Every exception in QUEUE is in QueueImpl
- Every value in QUEUE is in QueueImp
  (type could be more polymorphic)
- Every substructure matches, too (none here)
Signature Ascription

Ascription attaches signature to structure

• Transparent Ascription: types are revealed
  
structure strid : sig_exp = struct_exp
  
This method is stupid and broken (legacy)
  (But it’s awfully convenient)

• Opaque Ascription: types are hidden (“sealing”)
  
structure strid :> sig_exp = struct_exp
  
This method respects abstraction
  (And when you need to expose, can be tiresome)

Slogan: “use the beak”
Opaque Ascription

Recommended

Example:

```
structure Queue :> QUEUE = struct
  type 'a queue = 'a list
  exception Empty

  val empty = []
  fun put (x, q) = q @ [x]
  fun get [] = raise Empty
     | get (x :: xs) = (x, xs)
end
```

Not exposed: 'a Queue.queue = 'a list
  • Respects abstraction
How opaque ascription works

Outside module, no access to representation
• Protects invariants
• Allows software to evolve
• Type system limits interoperability

Inside module, complete access to representation
• Every function sees rep of every argument
• Key distinction abstract type vs object
Abstract data types and your homework

Natural numbers

- Funs /+/, /−/, /*/ see both representations
- Makes arithmetic relatively easy
- But type nat works only with type nat (no “mixed” arithmetic)
Abstract data types and your homework

Two-player games:
- Abstraction not as crisp as “number” or “queue”

Problems abstraction must solve:
- Interact with human player via strings
  (accept moves, display progress)
- Know whose turn it is
- Handle special features like “extra moves”
- Provide API for computer player

Result: a very wide interface
Abstraction design: Computer player

Computer player should work with any game, provided
  • Up to two players
  • Complete information
  • Always terminates

Brute force: exhaustive search

Your turn! What does computer player need?
  • Types?
  • Exceptions?
  • Functions?
Our computer player: AGS

Any game has two key types:

```ml
type config
structure Move : sig
  type move
    ...
    (* string conversion, etc *)
end
```

Key functions use both types:

```
val possmoves : config -> Move.move list
val makemove   : config -> Move.move -> config
```

Multiple games with different `config, move`?
Yes! Using key feature of ML: functor
Game interoperability with functors

functor AgsFun (structure Game : GAME) :> (sig
  structure Game : GAME
  val bestmove : Game.config -> Game.Move.move option
  val forecast : Game.config -> Player.outcome
end
  where type Game.Move.move = Game.Move.move
  and type Game.config = Game.config)
= struct
  structure Game = Game
  ... definitions of 'bestmove', 'forecast' ...
end
Functors: baby steps

A functor abstracts over a module

Formal parameters are declarations:

```ml
functor AddSingle(structure Q:QUEUE) =
  struct
    structure Queue = Q
    fun single x = Q.put (Q.empty, x)
  end
```

Combines familiar ideas:

- Higher-order functions
- type-lambda
Using Functors

Functor applications are evaluated at *compile time*.

```haskell
functor AddSingle(structure Q:QUEUE) = 
  struct
    structure Queue = Q
    fun single x = Q.put (Q.empty, x)
  end

Actual parameters are definitions

structure QueueS = AddSingle(structure Q = Queue)
structure EQueueS = AddSingle(structure Q = EQueue)

where *EQueue* is a more efficient implementation
Refining signature using \texttt{where type}

signature ORDER = sig
  type t
  val compare : t * t -> order
end

signature MAP = sig
  type key
  type 'a table
  val insert : key -> 'a -> 'a table -> 'a table
  ...
end

functor RBTree(structure O:ORD)
  :> (MAP where type key = O.t) =
  struct ... end
Versatile functors

Code reuse. RBTree with different orders

Type abstraction. RBTree with different ordered types

Separate compilation. RBTree compiled independently

functor RBTree(structure O:ORD)
    :> (MAP where type key = O.t) =
    struct
        ...
    end
Functors on your homework

Separate compilation:
  • Unit tests for natural numbers, without an implementation of natural numbers

Code reuse with type abstraction
  • Abstract Game Solver
    (any representation of game config, move)
Trick: Functor instead of function

AGS expects game with **fixed** initial configuration.

What about **family** of games? 3 sticks? 14 sticks? 1000 sticks?

Functor to rescue:

```
functor SticksFun (val N : int) :> GAME =
  struct ... end
```

```
structure S14 = SticksFun(val N = 14)
```
Reusable Abstractions: Extended Example

Error-tracking interpreter for a toy language

(More in 2nd modules recitation, next week)
Classic “accumulator” for errors

signature ERROR = sig
  type error   (* a single error *)
  type summary (* summary of what errors occurred *)

  val nothing : summary  (* no errors *)
  val <+> : summary * summary -> summary  (* combine *)

  val oneError : error -> summary

  (* laws:   nothing <+> s  ==  s
            s <+> nothing  ==  s
            s1 <+> (s2 <+> s3)  ==  (s1 <+> s2) <+> s3
                            // associativity *)

end
structure FirstError :>
  (ERROR where type error = string
   and type summary = string option) =
  struct
    type error      = string
    type summary    = string option

    val nothing = NONE
    fun <+> (NONE,  s) = s
    | <+> (SOME e, _) = SOME e

    val oneError = SOME
  end
structure AllErrors :>

  (ERROR where type error      = string
   and type summary = string list) =

struct
  type error      = string
  type summary    = error list

  val nothing = []
  val <+> = op @
  fun oneError e = [e]
end
Exercise: Simple arithmetic interpreter

(* Given: *)
datatype 'a comp = OK of 'a | ERR of AllErrors.summary

datatype exp = LIT of int | PLUS of exp * exp | DIV of exp * exp

(* Write an evaluation function that tracks errors. *)

val eval : exp -> int comp = ...
Exercise: LIT and PLUS cases

fun eval (LIT n) = OK n
| eval (PLUS (e1, e2)) =
  (case eval e1
    of OK v1 =>
      (case eval e2
        of OK v2 => OK (v1 + v2)
          | ERR s2 => ERR s2)
    | ERR s1 =>
      (case eval e2
        of OK _ => ERR s1
          | ERR s2 => ERR (AllErrors.++ (s1, s2))))

| eval (DIV (e1, e2)) =
  (case eval e1
     of OK v1 =>
         (case eval e2
             of OK 0 => ERR (AllErrors.oneError "Div 0")
             | OK v2 => OK (v1 div v2)
             | ERR s2 => ERR s2)
     | ERR s1 =>
         (case eval e2
             of OK v2 => ERR s1
             | ERR s2 => ERR (AllErrors.<+> (s1, s2))
     | ERR s1 =>
         (case eval e2
             of OK v2 => ERR s1
             | ERR s2 => ERR (AllErrors.<+> (s1, s2))
Combining generic computations

signature COMPUTATION = sig
  type 'a comp (* Computation! When run, results in value of type 'a or error summary. *)
  val succeeds : 'a -> 'a comp

  (* Apply a pure function to a computation. *)
  val <$> : ('a -> 'b) * 'a comp -> 'b comp

  (* Application inside computations. *)
  val <*> : ('a -> 'b) comp * 'a comp -> 'b comp

  (* Computation followed by continuation. *)
  val >>= : 'a comp * ('a -> 'b comp) -> 'b comp
end
Buckets of *generic* algebraic laws

```
succeeds a >>= k  ==  k a                       // identity
comp >>= succeeds  ==  comp                    // identity
comp >>= (fn x => k x >>= h)  ==  (comp >>= k) >>= h  // associativity
succeeds f <*> succeeds x  ==  succeeds (f x)  // success
...```
Environments using "computation"

signature COMPENV = sig

  type 'a env (* environment mapping strings to values of type 'a *)
  type 'a comp (* computation of 'a or an error summary *)

  val lookup : string * 'a env -> 'a comp
end
functor InterpFn(structure Error : ERROR
structure Comp : COMPUTATION
structure Env : COMPENV
val zerodivide : Error.error
val error : Error.error \to \ 'a Comp.comp
sharing type Comp.comp = Env.comp) =

struct
  val (\ast\ast, \$$, \gg\gg) = (Comp.\ast\ast, Comp.\$$, Comp.\gg\gg)

(* Definition of Interpreter *)

end
Definition of interpreter, continued

datatype exp = LIT of int
  | VAR of string
  | PLUS of exp * exp
  | DIV of exp * exp

fun eval (e, rho) =
  let fun ev(LIT n) = Comp.succeeds n
  | ev(VAR x) = Env.lookup (x, rho)
  | ev(PLUS (e1, e2)) = curry op + <$> ev e1 <*> ev e2
  | ev(DIV (e1, e2)) = ev e1 >>= (fn n1 =>
    ev e2 >>= (fn n2 =>
      case n2
        of 0 => error zerodivide
        | _ => Comp.succeeds
          (n1 div n2)))
  in ev e
end
“Computation” abstraction is a “monad”

Supported by special syntax in Haskell:

```haskell
eval :: Exp -> Hopefully Int
eval (LIT v) = return v
eval (PLUS e1 e2) =
  do { v1 <- eval e1
       ; v2 <- eval e2
       ; return (v1 + v2) }
eval (DIV e1 e2) =
  do { v1 <- eval3 e1
       ; v2 <- eval3 e2
       ; if v2 == 0 then Error "div 0"
           else return (v1 `div` v2) }
```
Extend a signature with include

signature ERRORCOMP = sig
    include COMPUTATION
    structure Error : ERROR
datatype 'a result = OK of 'a
                     | ERR of Error.summary
    val run : 'a comp -> 'a result
    val error : Error.error -> 'a comp
end
functor ErrorCompFn(structure Error : ERROR) =>
  (ERRORCOMP where type Error.error = Error.error
   and type Error.summary = Error.summary) =

struct
  structure Error = Error
datatype 'a result = OK of 'a
    | ERR of Error.summary

  type 'a comp = 'a result
  fun run comp = comp

  fun error e = ERR (Error.oneError e)
  fun succeeds = OK

  ...
end
ML module summary

New syntactic category: declaration
  • Of type, value, exception, or module

Signature groups declarations: interface

Structure groups definitions: implementation

Functor enables reuse:
  • Formal parameter: declarations
  • Actual parameter: definitions

Opaque ascription hides information
  • Enforces abstraction