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)*
   ...
   *)
end
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

structure Queue :> QUEUE = 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
  ```markdown
  structure strid : sig_exp = struct_exp
  ```
  This method is stupid and broken (legacy)
  (But it’s awfully convenient)

- **Opaque Ascription:** types are hidden ("sealing")
  ```markdown
  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:

```plaintext
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 \texttt{nat} works only with type \texttt{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:

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
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:

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.

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

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

Separate compilation:
- Heap sort without a heap
- 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)
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