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 -> Playeroutcome
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 : \( t \times t \rightarrow \text{order} \)
end

signature MAP = sig
  type key
  type '\a\ table'
  val insert : key \rightarrow \'a \rightarrow '\a\ table \rightarrow '\a\ table
  ...
end

functor RBTree(structure O:ORD)
  \( \rightarrow \) 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:
- 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:

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

```ocaml
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