

FUNCTORS!

Echo Recitation • श्री • 7 March 2007

- Consider this “abstract” dictionary signature:

```
signature DICT = sig
  type key    (* ABSTRACT TYPE *)
  val compare : key * key -> order
  type 'a dict
  val empty  : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

- Note `key` is left unspecified. Let's try using it...

```
signature DICT = sig
  type key (* ABSTRACT TYPE *)
  val compare : key * key -> order
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

- Does this IntDict definition work for us?

```
structure IntDict :> DICT = struct
  type key = int
  val compare : key * key -> order = Int.compare
  datatype 'a dict =
    Empty | Node of 'a dict * 'a * 'a dict
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

- You need to publicize the key type. Specialize the signature with where:

```
structure IntDict :> DICT where type key = int =
struct
  type key = int
  val compare : key * key -> order = Int.compare
  datatype 'a dict =
    Empty | Node of 'a dict * 'a * 'a dict
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

- Or:

```
signature INT_DICT = DICT where type key = int
structure IntDict :> INT_DICT = ...
```


- Key comparison isn't really a relevant part of dictionaries—really the only things we want are `insert`, `lookup`, etc.
- Idea: separate away these operations.

```
signature ORDER = sig
  type t (* ABSTRACT TYPE *)
  val compare : t * t -> order
end
```

- Key comparison isn't really a relevant part of dictionaries—really the only things we want are `insert`, `lookup`, etc.
- Idea: separate away these operations.

```
signature ORDER = sig
  type t (* ABSTRACT TYPE *)
  val compare : t * t -> order
end
```

```
structure IntOrder :> ORDER where type t = int =
struct
  type t = int
  val compare = Int.compare
end
```

- Key comparison isn't really a relevant part of dictionaries—really the only things we want are `insert`, `lookup`, etc.
- Idea: separate away these operations.

```
signature ORDER = sig
  type t (* ABSTRACT TYPE *)
  val compare : t * t -> order
end
```

```
structure IntOrder :> ORDER where type t = int =
struct
  type t = int
  val compare = Int.compare
end
```

```
structure StringOrder :> ORDER where type t = string =
struct
  type t = string
  val compare = String.compare
end
```



```
signature ORDER = sig
  type t (* ABSTRACT TYPE *)
  val compare : t * t -> order
end

signature PQUEUE = sig
  structure Elt:ORDER (* Parameter *)
  type pqueue
  exception empty
  val empty: pqueue
  val insert: Elt.t * pqueue -> pqueue
  val remove: pqueue -> Elt.t * pqueue
end
```

```
signature ORDER = sig
  type t (* ABSTRACT TYPE *)
  val compare : t * t -> order
end
```

```
signature PQUEUE = sig
  structure Elt:ORDER (* Parameter *)
  type pqueue
  exception empty
  val empty: pqueue
  val insert: Elt.t * pqueue -> pqueue
  val remove: pqueue -> Elt.t * pqueue
end
```

- Now to use an IntOrder structure:

```
structure IntPQueue :> PQUEUE where type Elt.t = int =
struct
  structure Elt:ORDER = IntOrder
  type pqueue = ... a heap based on Elt.compare ...
  exception empty
  val empty = ... the empty heap ...
  val insert = ...
  val remove = ...
end
```

```
signature ORDER = sig
  type t (* ABSTRACT TYPE *)
  val compare : t * t -> order
end
```

```
signature PQUEUE = sig
  structure Elt:ORDER (* Parameter *)
  type pqueue
  exception empty
  val empty: pqueue
  val insert: Elt.t * pqueue -> pqueue
  val remove: pqueue -> Elt.t * pqueue
end
```

- Now to use a `StringOrder` structure:

```
structure StringPQueue :> PQUEUE where type Elt.t = string =
struct
  structure Elt:ORDER = StringOrder
  type pqueue = ... a heap based on Elt.compare ...
  exception empty
  val empty = ... the empty heap ...
  val insert = ...
  val remove = ...
end
```



```
functor PQueue (structure E:ORDER)
    :> PQUEUE where type Elt.t = E.t =
struct
    structure Elt:ORDER = E
    type pqueue = ... a heap based on Elt.compare ...
    exception empty
    val empty = ... the empty heap ...
    val insert = ...
    val remove = ...
end
```

```
functor PQueue (structure E:ORDER)
    :> PQUEUE where type Elt.t = E.t =
struct
    structure Elt:ORDER = E
    type pqueue = ... a heap based on Elt.compare ...
    exception empty
    val empty = ... the empty heap ...
    val insert = ...
    val remove = ...
end
```

- Creating some priority queues with functors:

```
structure IntPQ      = PQueue (structure E = IntOrder)
structure StringPQ  = PQueue (structure E = StringOrder)
```

```
functor PQueue (structure E:ORDER)
    :> PQUEUE where type Elt.t = E.t =
struct
    structure Elt:ORDER = E
    type pqueue = ... a heap based on Elt.compare ...
    exception empty
    val empty = ... the empty heap ...
    val insert = ...
    val remove = ...
end
```

- Creating some priority queues with functors:

```
structure IntPQ      = PQueue (structure E = IntOrder)
structure StringPQ = PQueue (structure E = StringOrder)
```

- What are the signatures?

- Combined dictionary/priority queue abstract datatype:

- Combined dictionary/priority queue abstract datatype:

```
signature ADT = sig
  structure Val : ORDER
  type adt = ...
  ...
end
```

- Combined dictionary/priority queue abstract datatype:

```
signature ADT = sig
  structure Val : ORDER
  type adt = ...
  ...
end
```

- A functor can call other functors:

```
functor Adt(structure V:ORDER) :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  structure D = Dict(structure K = V)
  structure Q = PQueue(structure E = V)
  type adt = ...
  ...
end
```

- Combined dictionary/priority queue abstract datatype:

```
signature ADT = sig
  structure Val : ORDER
  type adt = ...
  ...
end
```

- A functor can call other functors:

```
functor Adt(structure V:ORDER) :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  structure D = Dict(structure K = V)
  structure Q = PQueue(structure E = V)
  type adt = ...
  ...
end
```

- Ensures that $D.Key.t = Q.Elt.t = V.t$, but...

- Add DICT and PQUEUE arguments to the functor:

```
functor Adt(structure V:ORDER and D:DICT and Q:PQUEUE)
  :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  type adt = ...
```

- Trying it out:

```
structure IntDict = Dict(structure K = IntOrder)
structure IntPQ   = PQueue(structure E = IntOrder)
structure A =
  ADT(structure V=IntOrder and D=IntDict and Q=IntPQ)
```

- Problem! It's not guaranteed in general that
 $D.Key.t = Q.Elt.t = V.t$.

- Add DICT and PQQUEUE arguments to the functor:

```
functor Adt(structure V:ORDER and D:DICT and Q:PQUEUE)
  :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  type adt = ...
```

WRONG!

- Here's a variant that works:

```
functor Adt(structure V:ORDER
  and D:DICT where type Key.t = V.t
  and Q:PQUEUE where type Elt.t = V.t)
  :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  type adt = ...
```

- The version from the last slide:

```
functor Adt(structure V:ORDER
            and D:DICTIONARY where type Key.t = V.t
            and Q:PQUEUE where type Elt.t = V.t)
  :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  type adt = ...
```

- A version that uses sharing:

```
functor Adt(structure V:ORDER and D:DICTIONARY and Q:PQUEUE
            sharing type D.Key.t = Q.Elt.t = V.t)
  :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  type adt = ...
```

- The version from the last slide:

```
functor Adt(structure V:ORDER and D:DICTIONARY and Q:PQUEUE
            sharing type D.Key.t = Q.Elt.t = V.t)
  :> ADT where type Val.t = V.t =
struct
  structure Val:ORDER = V
  type adt = ...
```

- A version that uses sharing:

```
functor Adt(structure D:DICTIONARY and Q:PQUEUE
            sharing type D.Key.t = Q.Elt.t)
  :> ADT where type Val.t = D.key.t =
struct
  structure Val:ORDER = D.key
  type adt = ...
```

- <http://www.cs.cmu.edu/~rwh/introsml/modules>
- and especially:
<http://www.cs.cmu.edu/~rwh/introsml/modules/subfun.htm>