15-150

Principles of Functional Programming

Slides for Lecture 17

Functors

March 20, 2025

Michael Erdmann

Lessons:

- Parameterized Structures
- Type Classes

A *functor*
expects a structure as argument
and produces a structure.

Simile:

abstraction	signature	type
implementation	structure	value
mapping	functor	function

Before we get to functors,
we need to explore some motivations.

Recall:

```
signature DICT =
sig
  type key = string                       (* concrete type *)
  type 'a entry = key * 'a                (* concrete type *)

  type 'a dict                            (* abstract type *)

  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Comment about implementing abstract types

The dictionary signature **`DICT`** specifies the abstract type

```
type 'a dict   (* abstract *)
```

Writing "**`type`**" in the signature merely means one has to provide some type **`'a dict`**, when creating a structure that ascribes to **`DICT`**.

A developer implementing such a structure can do so in a variety of ways, including using "**`datatype`**" declarations:

```
datatype 'a dict = Empty | Node of 'a dict * 'a entry * 'a dict
```

```
datatype 'a tree = Empty | Node of 'a tree * 'a entry * 'a tree
type 'a dict = 'a tree
```

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
type 'a dict = 'a entry tree
```

```
type 'a dict = 'a entry list
```

Representational Independence

- Client does not need to know and should not rely on how an abstract type is implemented.
- There could be different implementations (e.g., our two **QUEUE** implementations).
- All behavior relevant to the client should be specified via a signature.

Recall:

```
signature DICT =
sig
  type key = string                         (* concrete type *)
  type 'a entry = key * 'a                  (* concrete type *)

  type 'a dict                              (* abstract type *)

  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

We had made the dictionary abstract, we allowed the entries to be arbitrary, but we fixed the keys to be strings.

What if we wanted the keys to be integers … or something else?

We could try to make the dictionaries doubly polymorphic:

```
signature DICT =
sig
  type 'a key = 'a                      (* concrete type *)
  type ('a, 'b) entry = 'a key * 'b     (* concrete type *)

  type ('a, 'b) dict                    (* abstract type *)

  val empty : ('a, 'b) dict

  val lookup :
  val insert :

end
```

We could try to make the dictionaries doubly polymorphic:

```
signature DICT =
sig
  type 'a key = 'a                               (* concrete type *)
  type ('a, 'b) entry = 'a key * 'b              (* concrete type *)

  type ('a, 'b) dict                             (* abstract type *)

  val empty : ('a, 'b) dict

  val lookup :
  val insert :

end
```

We could try to make the dictionaries doubly polymorphic:

```
signature DICT =
sig
  type 'a key = 'a                          (* concrete type *)
  type ('a, 'b) entry = 'a key * 'b         (* concrete type *)

  type ('a, 'b) dict                        (* abstract type *)

  val empty : ('a, 'b) dict

  val lookup :
  val insert :

end
```

What goes here?

We realize that we need to be able to compare values of our **key** type.

At the very least the **key** type needs some kind of equality comparison.

Ideally it should have some kind of order comparison so we can implement dictionaries using binary search trees.

How do we model that?

One possibility is to make the comparison function an argument to **`insert`** and **`lookup`**, so:

```
lookup : ('a * 'a -> order) -> ('a, 'b) dict -> 'a -> 'b option

insert : ('a * 'a -> order) -> (('a, 'b) dict * ('a, 'b) entry)
                            -> ('a, 'b) dict
```

Then we could implement BST much as before:

```
structure BST : DICT =
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
     | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty


  fun lookup cmp d k =


  fun insert cmp (d, e) =

end  (* structure BST *)
```

Then we could implement BST much as before:

Remember: These two types were specified concretely in the signature,
so we need to implement them as specified.

```
structure BST : DIC
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
     | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty


  fun lookup cmp d k =


  fun insert cmp (d, e) =

end  (* structure BST *)
```

Then we could implement BST much as before:

```
structure BST : DICT =
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
     | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty


  fun lookup cmp d k =


  fun insert cmp (d, e) =

end  (* structure BST *)
```

The abstract dictionary type is again a tree, but now doubly polymorphic.

(And we wrote it without a separate hidden helper type, but that's not significant.)

Then we could implement BST much as before:

```
structure BST : DICT =
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
     | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty


  fun lookup cmp d k =



  fun insert cmp (d, e) =

end  (* structure BST *)
```

Implement the **`empty`** dictionary as an **`Empty`** tree, as before.

Then we could implement BST much as before:

```
structure BST : DICT =
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
     | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty


  fun lookup cmp d k =


  fun insert cmp (d, e) =

end  (* structure BST *)
```

The bodies of **`lookup`** and **`insert`** are much as before, but they now use **`cmp`** in place of **`String.compare`**.

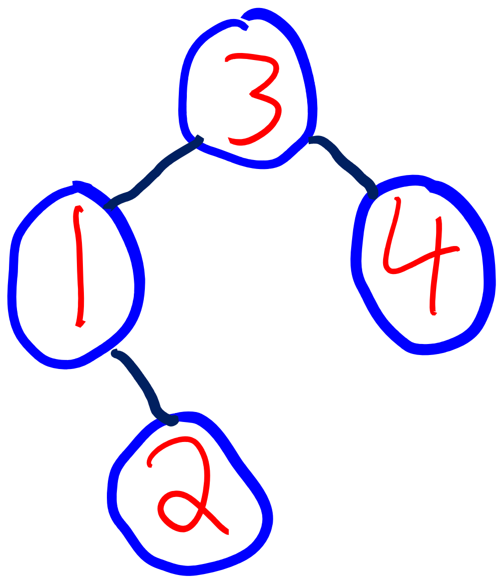
Does this do the trick?

Yes and No.

If we are careful to use the same comparison function `cmp` in `insert` as in `lookup`, and do that consistently for all operations with a given dictionary, then everything is fine.

However, it is easy to make a mistake. (A malicious user might do so intentionally.)

For example, perhaps we have created the following tree using **`Int.compare`**:



If we now binary search for **1**, using **`cmp`** below, we won't find it:

```
fun cmp (x,y) = Int.compare (y,x)
```

Let's take advantage of the type system
to ensure that
all operations
on a given dictionary
use the same comparison function.

A *type class* is a type along with some collection of operations for that type (not necessarily all operations).

Example:

```
signature ORDERED =
sig
  type t  (* parameter *)
  val compare : t * t -> order
end
```

Signature **`ORDERED`** specifies an “ordered type class” to consist of a type **`t`** along with a comparison function **`compare`** for **`t`**.

A *type class* is a type along with some collection of operations for that type (not necessarily all operations).

Example:

```
signature ORDERED =
sig
  type t  (* parameter *)
  val compare : t * t -> order
end
```

Signature **`ORDERED`** specifies an “ordered type class” to consist of a type **`t`** along with a comparison function **`compare`** for **`t`**.

Comment: The signature does not specify **`t`** concretely, but **`t`** need not be abstract. In a given setting, type **`t`** will be some already existing type, so t is a “parameter”. The signature is said to be “descriptive” of what we mean by an “ordered type class”. This is in contrast to our signature for dictionaries, which was “prescriptive”, defining a brand new abstract type along with operations for it.

Perspective on types of Types

- concrete : client and implementation both know what the type is.
- abstract : client does not (and should not) know how the type is implemented – the client's code must work regardless of the implementation.
- parameter : client supplies the type – implementation must work with whatever the client supplies.

Three structures implementing different **ORDERED**s:

Three structures implementing different **`ORDERED`**s:

```
structure IntLt : ORDERED =
struct
  type t = int
  val compare = Int.compare
end
```

Three structures implementing different **ORDERED**s:

```
structure IntLt : ORDERED =
struct
  type t = int
  val compare = Int.compare
end
```

Specify whatever type we care about.

Specify whatever the comparison function we want.

Three structures implementing different **ORDERED**s:

```
structure IntLt : ORDERED =
struct
  type t = int
  val compare = Int.compare
end


structure IntGt : ORDERED =
struct
  type t = int
  fun compare(x,y) = Int.compare(y,x)
end
```

We may want different comparison functions for a given type. Package each up in its own structure.

Three structures implementing different **`ORDERED`**s:

```
structure IntLt : ORDERED =
struct
  type t = int
  val compare = Int.compare
end


structure IntGt : ORDERED =
struct
  type t = int
  fun compare(x,y) = Int.compare(y,x)
end


structure StringLt : ORDERED =
struct
  type t = string
  val compare = String.compare
end
```

Three structures implementing different **ORDERED**s:

(again, now with the signature shown on the left)

```
signature ORDERED =
sig
  type t  (* parameter *)
  val compare : t * t -> order
end
```

```
structure IntLt : ORDERED =
struct
  type t = int
  val compare = Int.compare
end

structure IntGt : ORDERED =
struct
  type t = int
  fun compare(x,y) = Int.compare(y,x)
end

structure StringLt : ORDERED =
struct
  type t = string
  val compare = String.compare
end
```

Let us now redefine the dictionary signature:

```
signature DICT =
sig
  structure Key : ORDERED    (* parameter *)
  type 'a entry = Key.t * 'a  (* concrete *)

  type 'a dict                (* abstract *)

  val empty : 'a dict
  val lookup : 'a dict -> Key.t -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Let us now redefine the dictionary signature:

```
signature DICT =
sig
  structure Key : ORDERED     (* parameter *)
  type 'a entry = Key.t * 'a  (* concrete *)

  type 'a dict                (* abstract *)

  val empty : 'a dict
  val lookup : 'a dict -> Key.t -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Instead of a polymorphic key we have an “ordered” key.

We now implement dictionaries with different keys:

```
structure IntLtDict : DICT =
struct
  structure Key = IntLt
  (* rest of code much as in original BST but now using
     Key.t and Key.compare instead of key and String.compare. *)
end

structure IntGtDict : DICT =
struct
  structure Key = IntGt
  (* ... uses Key.t & Key.compare instead of key & String.compare ... *)
end

structure StringLtDict : DICT =
struct
  structure Key = StringLt
  (* ... uses Key.t & Key.compare instead of key & String.compare ... *)
end
```

We now implement dictionaries with different keys:

```
structure IntLtDict : DICT =
struct
  structure Key = IntLt
  (* rest of code much as in original BST but now using
     Key.t and Key.compare instead of key and String.compare. *)
end

structure IntGtDict : DICT =
struct
  structure Key = IntGt
  (* ... uses Key.t & Key.compare instead of key & String.compare ... *)
end

structure StringLtDict : DICT =
struct
  structure Key = StringLt
  (* ... uses Key.t & Key.compare instead of key & String.compare ... *)
end
```

only difference is the **Key**

A couple points to consider:

(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

(2) Can we avoid rewriting the same code over and over when implementing dictionaries that use different **Key**s?

(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

For instance, could we accidentally insert into a dictionary using `IntLtDict.insert` but then lookup using `IntGtDict.lookup` ?

After all, `IntLtDict.Key.t` and `IntGtDict.Key.t` are both `int`.

(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

Yes!

The types **`IntLtDict.dict`**

and **`IntGtDict.dict`** are different.

Each **`datatype 'a dict = ...`** declaration

creates a brand new type (*Dataype Generativity*).

(Printed representation is the same, but types are not.)

Typechecker will prevent intermingling of dictionaries.

(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

Yes!

The types **`IntLtDict.dict`** and **`IntGtDict.dict`** are different.

Each **`datatype 'a dict = ...`** declaration creates a brand new type (*Dataype Generativity*).

CAUTION: Answer would be NO if we had used association lists.

(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

Yes!

The types **`IntLtDict.dict`** and **`IntGtDict.dict`** are different.

Answer is also YES if we use opaque ascription.

(2) Can we avoid rewriting the same code over and over when implementing dictionaries that use different `Key`s?

Yes!

That's where functors come into the picture.

A functor expects a structure and creates a structure.

Let's write a functor that expects a structure ascribing to **`ORDERED`** and creates a structure ascribing to **`DICT`**.

```
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a

  datatype 'a dict =  ...

  (* code as before -- using
     Key.t and Key.compare   *)
end
```

argument

```
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a

  datatype 'a dict =  ...

  (* code as before -- using
     Key.t and Key.compare    *)
end
```

return

```
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a

  datatype 'a dict =  ...

  (* code as before -- using
     Key.t and Key.compare   *)
end
```

And now can define our earlier dictionaries as:

```
structure IntLtDict = TreeDict(IntLt)
structure IntGtDict = TreeDict(IntGt)
structure StringLtDict = TreeDict(StringLt)
```

If we want to hide the tree implementation of dictionaries, we could use opaque ascription:

```
functor TreeDict (K : ORDERED) :> DICT
= struct ... end
```

However, that also hides the key type in **DICT**. We need that to be known to be the same as the input key type. We therefore use a **where type** clause to expose the key type in **DICT**:

```
functor TreeDict (K : ORDERED)
        :> DICT where type Key.t = K.t
= struct ... end
```

Recall:

```
signature ORDERED =
sig
  type t  (* parameter *)
  val compare : t * t -> order
end


signature DICT =
sig
  structure Key : ORDERED     (* parameter *)

  type 'a entry = Key.t * 'a  (* concrete *)

  type 'a dict                (* abstract *)

  val empty : 'a dict
  val lookup : 'a dict -> Key.t -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Some Syntax Comments

- **`where type`** clauses expose types in a signature. So we could also have defined the following (for instance):

```
structure T = TreeDict(IntLt) :>
                  DICT where type Key.t = int
```

- Multiple **`where type`** clauses are permitted in SML/NJ.

Syntactic Sugar

One can pass multiple structures or even
value declarations to a functor using a more verbose format.
SML will wrap an implicit signature around these arguments.
For instance, the following verbose format:

no comma !

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED
= ... (* code that refers to Ox and Oy *)
```

desugars as (not quite this, but enough for our purposes):

```
functor PairOrder (P : sig
                         structure Ox : ORDERED
                         structure Oy : ORDERED
                       end)
                  : ORDERED
= ... (* code that refers to P.Ox and P.Oy *)
```

Example: 2D Lexicographic Order

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED
=
struct
  type t = Ox.t * Oy.t
  fun compare ((x1,y1), (x2,y2)) =
      (case Ox.compare (x1,x2)
         of EQUAL => Oy.compare (y1,y2)
          | otherwise => otherwise)
end
```

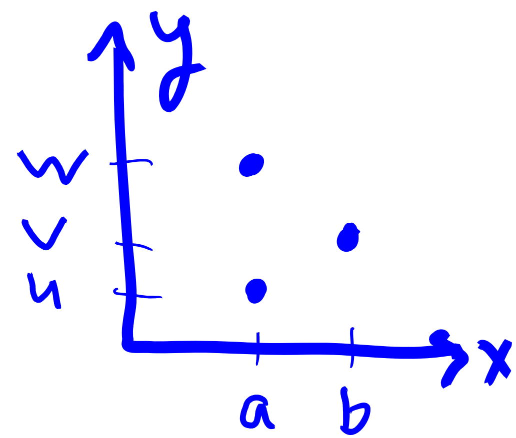
Example: 2D Lexicographic Order

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED
=
struct
  type t = Ox.t * Oy.t
  fun compare ((x1,y1), (x2,y2)) =
      (case Ox.compare (x1,x2)
         of EQUAL => Oy.compare (y1,y2)
          | otherwise => otherwise)
end
```

Example: 2D Lexicographic Order

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED
=
struct
  type t = Ox.t * Oy.t
  fun compare ((x1,y1), (x2,y2)) =
      (case Ox.compare (x1,x2)
         of EQUAL => Oy.compare (y1,y2)
          | otherwise => otherwise)
end
```

$(a,u) < (a,w) < (b,v)$



Now let's put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

```
structure GridOrder =
  PairOrder (structure Ox = StringLt
             structure Oy = IntLt)
```

Notice how we pass arguments in the verbose format:
As if we were defining a structure that contains Ox and Oy as substructures.

With the syntactic sugar we have:

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED =
  ... (* code that refers to Ox and Oy *)

structure GridOrder = PairOrder (structure Ox = StringLt
                                 structure Oy = IntLt)
```

With the syntactic sugar we have:

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED =
  ... (* code that refers to Ox and Oy *)

structure GridOrder = PairOrder (structure Ox = StringLt
                                 structure Oy = IntLt)
```

Without sugaring, we might write something like this:

```
signature PairSig =
sig
   structure Ox : ORDERED
   structure Oy : ORDERED
end

functor PairOrder (P : PairSig) : ORDERED =
  ... (* code that refers to P.Ox and P.Oy *)
```

With the syntactic sugar we have:

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED =
  ... (* code that refers to Ox and Oy *)

structure GridOrder = PairOrder (structure Ox = StringLt
                                 structure Oy = IntLt)
```

Without sugaring, we might write something like this:

```
signature PairSig =
sig
   structure Ox : ORDERED
   structure Oy : ORDERED
end

functor PairOrder (P : PairSig) : ORDERED =
  ... (* code that refers to P.Ox and P.Oy *)

structure Grid : PairSig =
struct
   structure Ox = StringLt
   structure Oy = IntLt
end

structure GridOrder = PairOrder (Grid)
```

With the syntactic sugar we have:

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED =
  ... (* code that refers to Ox and Oy *)

structure GridOrder = PairOrder (structure Ox = StringLt
                                 structure Oy = IntLt)
```

Without sugaring, we might write something like this:

```
signature PairSig =
sig
   structure Ox : ORDERED
   structure Oy : ORDERED
end

functor PairOrder (P : PairSig) : ORDERED =
  ... (* code that refers to P.Ox and P.Oy *)

structure Grid : PairSig =
struct
   structure Ox = StringLt
   structure Oy = IntLt
end

structure GridOrder = PairOrder (Grid)
```

For a given functor, need to be consistent.
Define functor AND call it using sugar.
Or define functor AND call it without sugar.
Cannot mix the syntax for a given functor.

Now let's put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

```
structure GridOrder =
  PairOrder (structure Ox = StringLt
             structure Oy = IntLt)
```

Create a board structure indexed by the grid coordinates:

```
structure Board = TreeDict(GridOrder)
```

Create a board value with something on it:

```
val b = Board.insert (Board.empty,
                      (("A", 1), fn x => x + 1))
```

Now let's put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

```
structure GridOrder =
  PairOrder (structure Ox = StringLt
             structure Oy = IntLt)
```

Create a board structure indexed by the grid coordinates:

```
structure Board = TreeDict(GridOrder)
```

Create a board value with something on it:

```
val b = Board.insert (Board.empty,
                       (("A", 1), fn x => x + 1))
```

Question: What is the type of **b** ?

Now let's put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

```
structure GridOrder =
  PairOrder (structure Ox = StringLt
             structure Oy = IntLt)
```

Create a board structure indexed by the grid coordinates:

```
structure Board = TreeDict(GridOrder)
```

Create a board value with something on it:

```
val b = Board.insert (Board.empty,
                      (("A", 1), fn x => x + 1))
```

Question: What is the type of **b** ?

Answer: `(int -> int) Board.dict` .

That is all.

Please have a good weekend.

See you Tuesday, when we will discuss an approach for maintaining hard-to-satisfy representation invariants in the context of **Red** **Black** Trees.