Modules II

Typeclasses and Functors

Review

17.0

```
signature THING =
sig
  type t
  val k : int * t
end
```

17.1

```
structure Thing1 :> THING =
struct
  type t = bool
  val b = false
  val k = (3,b)
end
```

Abstract structures

17.3

```
signature QUEUE =
sig
  type 'a queue
  val emp : 'a queue
  val ins : 'a * 'a queue -> 'a queue
  val rem : 'a queue -> ('a * 'a queue) option
end
```

- Structures opaquely ascribe to hide type implementation
- Can have multiple equivalent structures ascribing to the signature
- Structures may have different runtime properties

Section 1

Transparent Ascription

The idea

Remember: transparent means that the user can see (and use) the implementation of abstract types

17.0

```
signature THING =
sig
  type t
  val k : int * t
end
```

17.2

```
structure Thing2 : THING =
struct
  type t = bool
  val b = false
  val k = (3,b)
end
```

The idea

Remember: transparent means that the user can see (and use) the implementation of abstract types.

So we use the fact that it's transparent to package particular values with given (known) types.

Order!

17.4

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

17.5

```
structure IntStr_Ordered : ORDERED =
struct
  type t = int * string
  fun lex (EQUAL,Y) = Y
    | lex (X,_) = X
  fun compare ((i,s),(i',s')) =
    lex(Int.compare(i,i'),
        String.compare(s,s'))
end
```

Print!

17.6

```
signature PRINTABLE =
sig
  type t
  val toString : t -> string
end
```

17.7

```
structure B2I_Printable : PRINTABLE=
struct
  type t = bool -> int
  fun toString f =
    "fn true => "
    ^ (Int.toString (f true))
    ^ " | false => "
    ^ (Int.toString (f false))
end
```

Semigroup

17.8

```
signature SEMIGROUP =
sig
  type t

  (* INVARIANT: cmb is associative:
   *     for all x,y,z,
   * cmb(cmb(x,y),z) == cmb(x,cmb(y,z))
   *)
  val cmb : t * t -> t
end
```

17.9

```
structure Str_Semigroup : SEMIGROUP =
struct
  type t = string
  val cmb = op ^
end

type anyTypeYouWant = int * (string -> bool) list

structure Projection_Semigroup : SEMIGROUP =
struct
  type t = anyTypeYouWant
  fun cmb(x,y) = x
end
```

Section 2

Functors

Idea

Recall *lambda abstraction*

```
fun foo  (x : inputType) : outputType =  e
```

This is declares a "parametrized" version of `e`: it can take on numerous values, depending on the value of `x`.

Let's do this with structures!

```
functor Foo  (X : inputSig) : outputSig =  E
```

```
functor Foo'  (X :  inputSig) :> outputSig = E
```

Example: Trivial semigroups

17.10

```
signature TYPE = sig type t end

functor Fst_Semigroup (T : TYPE) : SEMIGROUP =
struct
  type t = T.t
  fun cmb(x,y) = x
end
```

Syntax

- Functors can take a single structure:

```
functor A (X : inputSig) : outputSig = E
```

- Or several:

```
functor B (structure X : inputSig1
           structure Y : inputSig2)
           : outputSig = E
```

Note: no comma between structures!

- Or just some data

```
functor C (type t1
           val t1 -> t1)
           : outputSig = E
```

Note: no comma between data!

Example

17.11

```
functor Proj1_Semigroup(type t): SEMIGROUP =
struct
  type t = t
  fun cmb(x,y) = x
end


structure IntFstSemi =
    Proj1_Semigroup(type t=int)
structure BoolFstSemi =
    Proj1_Semigroup(type t=bool)
```

Example: Ignoring structure

17.12

```
signature MONOID =
sig
  type t

  (* INVARIANT: cmb is associative:
   *     for all x,y,z,
   * cmb(cmb(x,y),z) == cmb(x,cmb(y,z))
   *)
  val cmb : t * t -> t

  (* INVARIANT: z is an identity for cmb:
   *     for all x,
   * cmb(x,z) == x == cmb(z,x)
   *)
  val z : t
end
```

Example: Ignoring structure

17.13

```
functor asSemi (M : MONOID) : SEMIGROUP =
struct
  type t = M.t
  val cmb = M.cmb
end

functor toMonoid (structure S : SEMIGROUP
                  val z : S.t) : MONOID =
struct
  type t = S.t
  val cmb = S.cmb
  val z = z
end
```

Map!

17.14

```
signature MAPPABLE =
sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end

structure TreeMappable =
struct
  datatype 'a t = Empty | Node of 'a t * 'a *
   'a t

  fun map f Empty = Empty
    | map f (Node(L,x,R)) =
       Node(map f L, f x, map f R)
end
```

Fold!

17.15

```
signature FOLDABLE =
sig
  structure Elt : SEMIGROUP
  type 'a t
  val fold : Elt.t -> Elt.t t -> Elt.t
end

functor MapFold (E : SEMIGROUP) : FOLDABLE =
struct
  structure Elt = E
  type 'a t = 'a list
  val fold = List.foldr E.cmb
end
```

Reduce!

17.16

```
signature REDUCIBLE =
sig
  structure Elt : MONOID
  type 'a t
  val reduce : Elt.t t -> Elt.t
end

functor TreeReducible(E : MONOID):REDUCIBLE =
struct
  structure Elt = E
  datatype 'a t = Empty
                | Node of 'a t * 'a * 'a t
  fun reduce Empty = E.z
    | reduce (Node(L,x,R)) =
       E.cmb(E.cmb(reduce L, x), reduce R)
end
```

Section 3

Example: Sets

Idea

A **set** is a data structure representing a finite unordered collection of elements.

- There is a set $\varnothing$
- If I have some element x, I can form the set $\{x\}$
- For each element x and each set S, I can ask whether $x \in S$ or $x \notin S$
- For each set S, there is a natural number $|S|$, representing how many distinct x are such that $x \in S$
- Given a set S and an element x, I can insert x into S and get some S'.

  $$y \in S' \qquad \iff \qquad y \in S \text{ or } y = x$$

  If $x \in S$, then S' should be the same as S and $|S| = |S'|$. If $x \notin S$, then $|S'| = |S| + 1$
- If S and S' are sets, $S \cup S'$, $S \cap S'$, and $S \setminus S'$ should be sets with the appropriate $\in$-behavior.

17.17

```
signature EQ =
sig     type t
        val equal : t * t -> bool     end
```

17.18

```
functor EqTypeToEQ (eqtype t):EQ =
struct
  type t = t
  val equal = op=
end
structure IntEQ = EqTypeToEQ (type t=int)
structure StrEQ = EqTypeToEQ (type t=string)
structure FunEQ : EQ = struct
  type t = bool -> int
  fun equal(f,g) =
    (f true = g true) andalso (f false = g
   false)
end
```

17.19

```
signature SET =
sig

  (* The EQ structure to use for element
   comparison *)
  structure Elt : EQ

  (* The type of the set *)
  type t

  (* These functions give the capability to
   create a set *)
  val empty : t
  val singleton : Elt.t -> t
  val fromList : Elt.t list -> t
```

17.20

```
  (* These functions give information about a set (destructors) *)
  val size : t -> int
  val toList : t -> Elt.t list

  (* Element related functions *)
  val insert : t -> Elt.t -> t
  val remove : t -> Elt.t -> t
  val member : t -> Elt.t -> bool

  (* Bulk Operations *)
  val union : t * t -> t
  val intersection : t * t -> t
  val difference : t * t -> t
end
```

17.21

```
functor MkSet (Elt : EQ) :> SET where type Elt.t = Elt.t =
struct

  structure Elt = Elt

  (* INVARIANT: the list contains no duplicates *)
  type t = Elt.t list
```

17.22

```
val remove' = fn (x,l) =>
  List.filter (not o (Fn.curry Elt.equal x))
  l
val insert' = fn (x,l) =>
  x :: remove' (x,l)
val member = fn l => fn x =>
  List.exists (Fn.curry Elt.equal x) l
```

Coming up

- Tomorrow: Implementing fast dictionary structures using intelligent invariants (red-black trees)
- Next Week: Implementing sophisticated modules
  - Mon/Tue: Parallel algorithms (the `Sequence` signature)
  - Wed/Thu: Game AIs (the `Game` signature)

Thank you!