Modules II

15-150 M21

Lecture 0714
14 July 2021

0 Typeclasses and Functors

Reminder: where the base case goes when folding

```
foldr (op^) "!" ["H","E","L","L","O"]
 ⟹ "H"^(foldr (op^) "!" ["E","L","L","O"])
 ⟹ "H"^("E"^(foldr (op^) "!" ["L","L","O"]))
 ⟹ "H"^("E"^("L"^(foldr (op^) "!" ["L","O"])))
 ⟹ "H"^("E"^("L"^("L"^(foldr (op^) "!" ["O"]))))
 ⟹ "H"^("E"^("L"^("L"^("O"^(foldr (op^) "!" []))))))
 ⟹ "H"^("E"^("L"^("L"^("O"^"!"))))
 ⟹ "HELLO!"
```

0714.0 (Tree.sml)

```
  fun inord Empty = []
    | inord (Node(L,x,R)) =
         (inord L)@(x::inord R)
```

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b
```

REQUIRES: g is total

ENSURES:

$$\texttt{foldr g z T} \cong \texttt{List.foldr g z (inord T)}$$

0714.1 (Tree.sml)

```
  fun foldr g z Empty = z
    | foldr g z (Node(L,x,R)) =
        foldr g (g(x,foldr g z R)) L
```

Problem: Cannot make recursive calls in parallel!

Associativity

If `(inord T)` $\cong$ `[x1,x2,x3]`, then

`foldr g z T` $\cong$ `g(x1,g(x2,g(x3,z)))`

in order to make this more parallel, we need this to be the same as

`g(g(x1,x2),g(x3,z))`

.(note this constrains the type!)

Defn. A function `g : t * t -> t` is said to be **associative** if

`g(g(x,y),z)` $\cong$ `g(x,g(y,z))`

for all `x,y,z : t`. (Examples: `op+`, `Int.max`. Non-example: `op-`)

Semigroup

0714.2 (TYPECLASSES.sig)

```
signature SEMIGROUP =
sig
  type t

  (* INVARIANT: cmb is associative *)
  val cmb : t * t -> t
end
```

0714.3 (Typeclasses.sml)

```
structure IntMaxSemi : SEMIGROUP =
struct
  type t = int
  val cmb = Int.max
end
structure IntMinSemi : SEMIGROUP =
struct
  type t = int
  val cmb = Int.min
end
```

0714.4 (TYPECLASSES.sig)

```
signature FOLDABLE =
sig
  type 'a T
  structure S : SEMIGROUP

  val fold : S.t -> S.t T -> S.t
end
```

```
functor F (S : SIG1):SIG2 = struct ... end
```

```
functor F (structure S1 : SIG1
           structure S2 : SIG2) : SIG3 =
    struct ... end
```

```
functor F (type t
           val g : t -> int)
           : SIG' =
    struct ... end
```

0714.5 (Typeclasses.sml)

```
functor mkListFold (S : SEMIGROUP):FOLDABLE =
struct
  type 'a T = 'a list
  structure S = S

  val fold = List.foldr S.cmb
end
```

0714.6 (Typeclasses.sml)

```
functor mkTreeFold (S : SEMIGROUP):FOLDABLE =
struct
  type 'a T = 'a Tree.tree
  structure S = S

  val fold = Tree.foldr S.cmb
end
```

```
fun foldr g z Empty = z
  | foldr g z (Node(L,x,R)) =
        g(foldr g z L,g(x,foldr g z R))
```

Does this work?

$$\texttt{foldr g z T} \overset{?}{\cong} \texttt{List.foldr g z (inord T)}$$

No:

```
foldr op^ "!"
(Node(Node(Empty,"y",Empty),"x",Empty))
```

$\cong$

```
"!y!x!"
```

Defn. A value `z:t` is said to be an **identity** for `g:t*t->t` if

$$\texttt{g(x,z)} \quad \cong \quad \texttt{x} \quad \cong \quad \texttt{g(z,x)}$$

E.g.

- 0 for `op+`
- 1 for `op*`
- `""` for `op^`

0714.7 (TYPECLASSES.sig)

```
signature MONOID =
sig
  type t

  (* INVARIANT: cmb is associative *)
  val cmb : t * t -> t

  (* INVARIANT: z is an identity for cmb *)
  val z : t
end
```

0714.8 (Typeclasses.sml)

```
structure IntPlusMonoid : MONOID =
struct
  type t = int
  val cmb = op+
  val z = 0
end
```

0714.9 (Typeclasses.sml)

```
structure StringMonoid : MONOID =
struct
  type t = string
  val cmb = op^
  val z = ""
```

0714.10 (Typeclasses.sml)

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

0714.11 (TYPECLASSES.sig)

```
signature REDUCIBLE =
sig
  type 'a T
  structure M : MONOID

  val reduce : M.t T -> M.t
end
```

0714.12 (Typeclasses.sml)

```
functor mkListReduce (M : MONOID):REDUCIBLE =
struct
  type 'a T = 'a list
  structure M = M

  val reduce = List.foldr M.cmb M.z
end
```

0714.13 (Tree.sml)

```
  fun reduce g z Empty = z
    | reduce g z (Node(L,x,R)) =
        g(reduce g z L,g(x,reduce g z R))
```

0714.14 (Typeclasses.sml)

```
functor mkTreeReduce (M : MONOID):REDUCIBLE =
struct
  type 'a T = 'a Tree.tree
  structure M = M

  val reduce = Tree.reduce M.cmb M.z
end
```

5-minute break?

1 Sets

In SML, we have the notion of **equality types**, which are types that can be compared with =.

Examples:

- `int`
- `string list`
- `bool * bool option`

Non-examples:

- `real`
- `int -> int`
- `bool -> bool`

A type variable with two apostrophes, e.g. `''a`, is constrained to be an equality type. We can specify an equality type in a signature using the `eqtype` keyword.

0714.15 (SEARCH.sig)

```
signature EQ =
sig
  type t
  (* INVARIANT: equal is a reasonable notion of
   equality *)
  val equal : t -> t -> bool
end
```

0714.16 (Search.sml)

```
functor mkEq (eqtype t) : EQ =
struct
  type t = t
  val equal = Fn.equal
end
```

0714.17 (Search.sml)

```
structure IntEq = mkEq(type t = int)
structure BoolEq = mkEq(type t = bool)
structure StringEq = mkEq(type t = string)
structure IntListEq = mkEq(type t = int list)
structure IntTreeEq = mkEq(type t = int Tree.
  tree)
```

0714.18 (SEARCH.sig)

```
signature ORD =
sig
  type t

  (* INVARIANT: cmp is a comparison function *)
  val cmp : t * t -> order
end
```

0714.19 (Search.sml)

```
structure IntOrd : ORD =
struct
  type t = int
  val cmp = Int.compare
end
structure stringOrd : ORD =
struct
  type t = string
  val cmp = String.compare
end
```

0714.20 (Search.sml)

```
functor cmpEqual (K : ORD):EQ =
struct
  type t = K.t
  fun equal x y = K.cmp(x,y) = EQUAL
end
```

0714.21 (SEARCH.sig)

```
signature SET =
sig
  structure Elt : EQ

  type Set

  val empty : Set

  val insert : Elt.t * Set   -> Set
  val lookup : Set -> Elt.t -> Elt.t option
```

Set signature, continued

0714.22 (SEARCH.sig)

```
  val overwrite : Elt.t * Set   -> Set
  val remove : Elt.t * Set -> Set

  val union : Set -> Set -> Set

  val toString : (Elt.t -> string) -> Set -> 
 string
end
```

Lecture ended here on 14 July 2021. You're not expected to know anything past here.

Code Review:

`OrdListSet`

Code Review:

`OrdTreeSet`

Demonstration:

Proving Representation Independence

Summary

- We can use signatures (with invariants) to explicitly codify typeclasses with specific properties
- We can use structures as input to functors producing more elaborate structures, making our code more modular and maintainable
- We can maintain more complex invariants, and prove the equivalence of different structures ascribing to the same signature.

Next Time

- Sets and Dictionaries
- Balance invariants and Red-Black Trees

Thank you!