Discussion 07

Functors

Kenneth Fang (kwf37), Newton Ni (cn279)

Feb. 13, 2019

A1 Debrief: Pipelining

```
let cipher_char config c =
  (map_plug config.plugboard
    (inv_index
      (map_rotors_l_to_r config.rotors
        (map_refl config.refl
          (map_rotors_r_to_l config.rotors
            (index
              (map_plug config.plugboard c)))))))
```

A1 Debrief: Pipelining

```
let cipher_char config c =
    let v1 = map_plug config.plugboard c in
    let v2 = index v1 in
    let v3 = map_rotors_r_to_l config.rotors v2 in
    let v4 = map_refl config.refl v3 in
    let v5 = map_rotors_l_to_r config.rotors v4 in
    let v6 = inv_index v5 in
    map_plug config.plugboard v6
```

A1 Debrief: Pipelining

```
let cipher_char config c =
  c |> map_plug config.plugboard
    |> index
    |> map_rotors_r_to_l config.rotors
    |> map_refl config.refl
    |> map_rotors_l_to_r config.rotors
    |> inv_index
    |> map_plug config.plugboard
```

Key Concept

- Modules are to classes as functors are to **generic** classes

Functors

- Parameterize modules on other modules

Functors

- Parameterize modules on other modules
- Similar to higher-order functions at a module level

Functors

- Parameterize modules on other modules
- Similar to higher-order functions at a module level
- Another mechanism for code reuse

Functors: Map

```
(** Represents a type with a equality relation. *)
module type Equatable = sig

  (** The type of equatable values. *)
  type t

  (** [eq a b] is [true] if [a = b] else [false]. *)
  val eq: t -> t -> bool

end
```

Functors: Map

```
(** Represents a key-value mapping. *)
module type Map = sig
  type 'a t
  type key
  val empty: 'a t
  val add: key -> 'a -> 'a t -> 'a t
  val get: key -> 'a t -> 'a option
end
```

Functors: Map

```
module MakeList (K : Equatable)
  : (Map with type key = K.t) =
struct
  type 'a t = (* ... *)
  type key = (* ... *)
  let empty = (* ... *)
  let add k v map = (* ... *)
  let rec get k map = (* ... *)
end
```

Functors: Map

```
module MakeList (K : Equatable)
  : (Map with type key = K.t) =
struct
  type 'a t = 'a list
  type key = (* ... *)
  let empty = (* ... *)
  let add k v map = (* ... *)
  let rec get k map = (* ... *)
end
```

Functors: Map

```
module MakeList (K : Equatable)
  : (Map with type key = K.t) =
struct
  type 'a t = 'a list
  type key = K.t
  let empty = (* ... *)
  let add k v map = (* ... *)
  let rec get k map = (* ... *)
end
```

Functors: Map

```
module MakeList (K : Equatable)
  : (Map with type key = K.t) =
struct
  type 'a t = 'a list
  type key = K.t
  let empty = []
  let add k v map = (* ... *)
  let rec get k map = (* ... *)
end
```

Functors: Map

```
module MakeList (K : Equatable)
  : (Map with type key = K.t) =
struct
  type 'a t = 'a list
  type key = K.t
  let empty = []
  let add k v map = (k, v) :: map
  let rec get k map = (* ... *)
end
```

Functors: Map

- `MakeList (Key)` makes maps that can compare `Key.t`

Functors: Map

- `MakeList (Key)` makes maps that can compare `Key.t`
- `MakeList (Int)` (see `equal.ml`) compares `Int.t = int`

Functors: Tests

- See `test.ml` for testing multiple implementations of maps

Functors: Tests

- See `test.ml` for testing multiple implementations of maps
- Same functionality, different implementations

Functors: Tests

- See `test.ml` for testing multiple implementations of maps
- Same functionality, different implementations
- Reuse test logic with functors!

Functors: Experiments

- Sealing and constraints
  - `module Map = Map.MakeList (Equal.Int)`
  - `module Map = Map.MakeList (Equal.IntSealed)`
  - `module Map = Map.MakeList (Equal.IntConstrained)`

Functors: Experiments

- Sealing and constraints
  - `module Map = Map.MakeList (Equal.Int)`
  - `module Map = Map.MakeList (Equal.IntSealed)`
  - `module Map = Map.MakeList (Equal.IntConstrained)`
- Other map implementations
  - `MakeFun` functor