Functional Programming

Tyng–Ruey Chuang

2010 Formosan Summer School
on Logic, Language, and Computation
June 28 – July 9, 2010

This course note . . .

- . . . is prepared for the *2010 Formosan Summer School on Logic, Language, and Computation (FLOLAC)* held in Taipei, Taiwan,
- . . . is made available from the FLOLAC '10 web site:

  `http://flolac.iis.sinica.edu.tw/flolac10/`

  (please also check the above site for updated version)
- . . . and is released to the public under a Creative Commons Attribution-ShareAlike 3.0 Taiwan license:

  `http://creativecommons.org/licenses/by-sa/3.0/`

Course outline

Unit 1. Basics of functional programming.

Unit 2. Fold/unfold functions; Parametric modules.

Each unit consists of 2 hours of lecture and 1 hour of lab/tutor. Examples will be given in Objective Caml (O'Caml). Useful online resources about O'Caml:

- Web site: `http://caml.inria.fr/`
- Book: *Developing Applications with Objective Caml.*
  URL: `http://caml.inria.fr/pub/docs/oreilly-book/`

1 Basics of functional programming

1.1 Function, evaluation, and binding

Functions

```
let x = 1
let y = x + 1
let succ n = n + 1
let z = succ y
```

-
  ```
  val x : int = 1
  val y : int = 2
  val succ : int -> int = <fun>
  val z : int = 3
  ```

```
let sum x y = x + y
let five = sum 2 3
```

-
  ```
  val sum : int -> int -> int = <fun>
  val five : int = 5
  ```

```
let plus3 = sum 3
let seven = plus3 4
```

-
  ```
  val plus3 : int -> int = <fun>
  val seven : int = 7
  ```

Anonymous functions

```
let succ = fun n -> n + 1
let one = succ 0
let two = (fun n -> n + 1) one
```

-
  ```
  val succ : int -> int = <fun>
  val one : int = 1
  val two : int = 2
  ```

```
let sum = fun x -> fun y -> x + y
let plus3 = sum 3
```

-
  ```
  val sum : int -> int -> int = <fun>
  val plus3 : int -> int = <fun>
  ```

```
let twice = fun f -> fun x -> f (f x)
let plus6 = twice plus3
let seven = plus6 one
```

-
  ```
  val twice : (’a -> ’a) -> ’a -> ’a = <fun>
  val plus6 : int -> int = <fun>
  val seven : int = 7
  ```

Functions as arguments and as results

```
let compose f g = fun x -> f (g x)
let plus3  n = n + 3
let times2 n = n * 2
let this = compose plus3  times2 1
let that = compose times2 plus3  1
```

-
  ```
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  val plus3 : int -> int = <fun>
  val times2 : int -> int = <fun>
  val this : int = 5
  val that : int = 8
  ```

```
let twice f = compose f f
let what  = twice (fun n -> n + n)
let guess = what 1
```

-
  ```
  val twice : ('a -> 'a) -> 'a -> 'a = <fun>
  val what : int -> int = <fun>
  val guess : int = 4
  ```

Notations in O'Caml

Function application is just juxtaposition, and is left associative. These two definitions are the same:

- `let this = compose plus3 times2 1`
- `let this = ((compose plus3) times2) 1`

Function abstraction is right associative. These two definitions are the same:

- `let sum = fun x -> fun y -> x + y`

  ```
  val sum : int -> int -> int = <fun>
  ```

- `let sum = fun x -> (fun y -> x + y)`

  ```
  val sum : int -> (int -> int) = <fun>
  ```

Evaluation in O'Caml

- Expressions are evaluated before they are passed as arguments to the function body.
- The function body is evaluated only when all the arguments are evaluated.
- Functions can be partially applied.

Binding in O'Caml

- Lexical binding: Expressions are evaluated and bound to the corresponding identifiers in the order they appear in the program text.
- Nested binding: Outer bindings are shadowed by inner bindings.

```
let x = 100
let f y = let  x = x + y  in  x
let x = 10
let z = f x
```

- Simultaneous binding: Several bindings occur at the same time under the same environment.

```
let x = z
and z = x
```

- Recursive binding: Identifiers can be referred to when they are being defined.

```
let rec fac n = if n <= 0 then 1 else n * (fac (n -1))
let six = fac 3
```

Recursive functions: examples

- Expressiveness: Euclid's algorithm for greatest common divisor (gcd), assuming integers $m, n > 0$:

```
let rec gcd m n =
     if m mod n = 0
        then n
        else gcd n (m mod n)

let u = gcd 57 38
let v = gcd 38 59
```

- The danger of non-terminating computation:

```
let rec loop x = loop x
let oops = loop 0
```

1.2 Data types

Built-in data types in O'Caml

type int `0`, `-1`, ...

type char `'a'`, `'\''`, ...

type string `"\"O'Caml\" is a fine language.\n"`, ...

type float `3.14159`, `0.314159e1`, ...

type unit = ()

type bool = false | true

type 'a list = [] | :: of 'a * 'a list `[]`, `true::false::[]`, `[1; 2; 3]`, ...

type 'a option = None | Some of 'a `None`, `Some 17`, `Some [None; Some true]`, ...

Built-in type operators in O'Caml

Cartesian product

```
type int_pair = int * int

let rec gcd (m, n) =
     if m mod n = 0
        then n else gcd (n, m mod n)


val gcd : int * int -> int = <fun>
```

Function space

```
type int2int2int = int -> int -> int

let rec gcd m n =
     if m mod n = 0
        then n else gcd n (m mod n)


val gcd : int -> int -> int = <fun>
```

Expressions, values, and types

- Well-typed expressions:

  ```
  0, (1 + 2), (sum 2 3), (fun x -> fun y -> x + y), (2, true)
  ```

- Ill-typed expressions:

  ```
  (1 + '2'), (sum 2 3.0), ((fun x -> fun y -> x + y) 0 1 2)
  ```

- All O'Caml values have types:

  ```
  val sum : int -> int -> int = <fun>
  val five : int = 5
  ```

- Some values are polymorphic:

  ```
  val twice : ('a -> 'a) -> 'a -> 'a = <fun>
  val empty_list : 'a list = []
  ```

- Expressions are statically checked to ensure they always evaluate to values.

O'Caml is strict

- O'Caml insists on evaluating the arguments in a function application though the arguments may not be required for the computation in the function body. O'Caml is called a *strict* language.
- Some functional language, e.g., Haskell, will evaluate the function arguments only when they are demanded by the computation in the function body. These languages are *non-strict*.
- What is wrong in this picture (in O'Caml):

  ```
  let oracle () = ...

  let choice this that =
      if oracle () then this else that
  ```

-
  ```
  let rec loop x = loop x
  let oops = choice (loop 0) 0
  ```

Functions to the rescue!

```
let rec loop x = loop x

let choice this that =
    if oracle () then this else that

let new_choice this that =
    if oracle () then this () else that ()

let was = choice (loop 0) 0
let now = new_choice (fun () -> loop 0) (fun () -> 0)
```

```
val choice : 'a -> 'a -> 'a = <fun>
val new_choice : (unit -> 'a) -> (unit -> 'a) -> 'a = <fun>
```

What about variables?

- We can bind values to identifiers; once an identifier is bound, its value never changes. Of course, bindings can be nested hence, for the same identifier, the inner binding may shadow outer binding.
- Can one implement a counter using only functions?
- We can implement *many* counters using only functions!

-
  ```
  let init value = fun () -> value
  let read counter = counter ()
  let step counter more = fun () -> read counter + more
  ```

  ```
  val init : 'a -> unit -> 'a = <fun>
  val read : (unit -> 'a) -> 'a = <fun>
  val step : (unit -> int) -> int -> unit -> int = <fun>
  ```

Counters via functions

```
let init value = fun () -> value
let read counter = counter ()
let step counter more = fun () -> read counter + more

let mem = init 0
let x = step mem 1
let y = step mem 2
let z = step x   100
let x_y_z = (read x, read y, read z)
```

```
val init : 'a -> unit -> 'a = <fun>
val read : (unit -> 'a) -> 'a = <fun>
val step : (unit -> int) -> int -> unit -> int = <fun>
val mem : unit -> int = <fun>
val x : unit -> int = <fun>
val y : unit -> int = <fun>
val z : unit -> int = <fun>
val x_y_z : int * int * int = (1, 2, 101)
```

Programming by pattern-matching

```
type 'a table = (string * 'a) list

let rec lookup key table =
    match table with
          [] -> None
        | (name, value) :: rest ->
          if key = name then Some value
                        else lookup key rest

type color = Red | Yellow | Green
let fruits = [("banana", Yellow); ("guava", Green)]

let this = lookup "guava" fruits
let that = lookup "mango" fruits
```

```
val lookup : 'a -> ('a * 'b) list -> 'b option = <fun>
val fruits : (string * color) list = [("banana", Yellow); ("guava", Green)]
val this : color option = Some Green
val that : color option = None
```

List reversal: two examples

-
  ```
  let rec reverse list =
      match list with
            [] -> []
          | head :: tail -> (reverse tail) @ [head]
  ```

-
  ```
  let reverse list =
      let rec rev rest accumulator =
          match rest with
                [] -> accumulator
              | hd :: tl -> rev tl (hd :: accumulator)
   in
      rev list []
  ```

- Both have type:

```
val reverse : 'a list -> 'a list = <fun>
```

- Which one is better?

Functions over lists

```
let rec filter p list =
    match list with
          [] -> []
        | head :: tail -> if p head then head :: (filter p tail)
                                     else filter p tail

let rec append front rear =
    match front with
          [] -> rear
        | head :: tail -> head :: (append tail rear)

let this = filter (fun n -> n mod 2 = 0) [1; 2; 3]
let that = append [1; 2; 3] [100; 101; 102]
```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
val append : 'a list -> 'a list -> 'a list = <fun>
val this : int list = [2]
val that : int list = [1; 2; 3; 100; 101; 102]
```

User-defined type constructors

```
type 'a tree = Leaf
             | Node of 'a * 'a tree * 'a tree
```

- `tree` is a type constructor: it construct a type α *tree* whenever given a type α.
- `Leaf` and `Node` are the two value constructors for type α *tree*.

```
Leaf: 'a tree
Node: 'a * 'a tree * 'a tree -> 'a tree
```

- In O'Caml, type constructors start with lower-case letters; value constructors start with upper-case letters.

- In O'Caml, type constructors and value constructors are unary. Type construction uses postfix notation; value construction, prefix.

  ```
  Some (Node (1, Node (0, Leaf, Leaf), Node (2, Leaf, Leaf)))
  ```

  has type

  ```
  int tree option
  ```

Functions over trees

```
let rec swap tree =
    match tree with
          Leaf -> Leaf
        | Node (here, left, right) ->
          Node (here, swap right, swap left)


let rec insert key tree =
    match tree with
          Leaf -> Node (key, Leaf, Leaf)
        | Node (here, left, right) ->
          if key < here
             then Node (here, insert key left, right)
             else Node (here, left, insert key right)
```

```
val swap : 'a tree -> 'a tree = <fun>
val insert : 'a -> 'a tree -> 'a tree = <fun>
```

Functions over trees, continued

```
let rec build f s =
    match f s with
          None -> Leaf
        | Some (a, left, right) ->
          Node (a, build f left, build f right)

let range (low, high) =
    if low > high
       then None
       else let mid = (low + high) / 2 in
            Some (mid, (low, mid - 1), (mid + 1, high))

let tree1to7 = build range (1, 7)
```

```
val build : ('a -> ('b * 'a * 'a) option) -> 'a -> 'b tree = <fun>
val range : int * int -> (int * (int * int) * (int * int)) option = <fun>
val tree1to7 : int tree = Node (4, Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Leaf)),
                                   Node (6, Node (5, Leaf, Leaf), Node (7, Leaf, Leaf)))
```

2 Fold/unfold functions; Parametric modules

2.1 Fold/unfold functions for data types

Functions over lists, re-visited

```
let rec filter p list =
    match list with
          [] -> []
        | head :: tail ->
          if p head then head :: (filter p tail)
                    else filter p tail

let rec append front rear =
    match front with
          [] -> rear
        | head :: tail -> head :: (append tail rear)
```

- Both functions work on lists in a bottom-up manner.
- What is the base case, and what is the inductive step?

Fold function for lists

```
let rec fold (base, step) list =
    match list with
          [] -> base
        | hd :: tl -> step (hd, fold (base, step) tl)

let filter p list =
    let step (hd, acc) = if p hd then (hd :: acc) else acc
 in
    fold ([], step) list

let append front rear =
    fold (rear, fun (hd, acc) -> hd :: acc) front
```

```
val fold : 'a * ('b * 'a -> 'a) -> 'b list -> 'a = <fun>
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
val append : 'a list -> 'a list -> 'a list = <fun>
```

Fold function for trees

```
let rec swap tree =
    match tree with
          Leaf -> Leaf
        | Node (here, left, right) ->
          Node (here, swap right, swap left)

let rec fold (base, step) tree =
    match tree with
          Leaf -> base
        | Node (here, left, right) ->
          step (here, fold (base, step) left,
                      fold (base, step) right)

let swap' tree = fold (Leaf,
    fun (here, left, right) -> Node (here, right, left)) tree
```

```
val fold : 'b * ('a * 'b * 'b -> 'b) -> 'a tree -> 'b = <fun>
```

What is a tree, anyway?

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

- A tree of type α *tree* is a value that can be folded.
- Whenever given a base value of type β, and an inductive function of type $\alpha \times \beta \times \beta \rightarrow \beta$, a tree can be folded into a value of type β.

A new data type for trees

```
type ('a, 'b) t = Leaf
                | Node of 'a * 'b * 'b

type 'a tree = Rec of ('a, 'a tree) t

let rec fold f tree =
    match tree with
          Rec Leaf -> f Leaf
        | Rec (Node (here, left, right)) ->
            f (Node (here, fold f left, fold f right))
```

```
type ('a, 'b) t = Leaf | Node of 'a * 'b * 'b
type 'a tree = Rec of ('a, 'a tree) t
val fold : (('a, 'b) t -> 'b) -> 'a tree -> 'b = <fun>
```

A new `swap` function

```
let swap tree =
    let f t = match t with Leaf -> Rec Leaf
                         | Node (here, left, right) ->
                      Rec (Node (here, right, left))
 in
     fold f tree

let tree123 = Rec (Node (2, Rec (Node (1, Rec Leaf, Rec Leaf)),
                            Rec (Node (1, Rec Leaf, Rec Leaf))))
let tree321 = swap tree123
```

```
val swap : 'a tree -> 'a tree = <fun>
val tree123 : int tree =
  Rec (Node (2, Rec (Node (1, Rec Leaf, Rec Leaf)),
                Rec (Node (1, Rec Leaf, Rec Leaf))))
val tree321 : int tree =
  Rec (Node (2, Rec (Node (1, Rec Leaf, Rec Leaf)),
                Rec (Node (1, Rec Leaf, Rec Leaf))))
```

Look at a tree this way!

```
type ('a, 'b) t = Leaf | Node of 'a * 'b * 'b

type 'a tree = Rec of ('a, 'a tree) t

val fold : (('a, 'b) t -> 'b) -> 'a tree -> 'b = <fun>
```

- Type constructor $(\alpha, \beta)\ t$ defines (the only) two forms of a tree node.
- Type constructor $\alpha\ tree$ defines a tree as a recursive structure via type constructor $(\alpha, \beta)\ t$. The recursion occurs at the second type argument to t.
- A function of type $(\alpha, \beta)\ t \rightarrow \beta$ comprises both the base case and the inductive step necessary for folding a value of type $\alpha\ tree$ to a value of type β.

A new data type for trees, continued

```
type ('a, 'b) t = Leaf
                | Node of 'a * 'b * 'b

type 'a tree = Rec of ('a, 'a tree) t

let rec unfold g seed =
    match g seed with
          Leaf -> Rec Leaf
        | Node (here, left, right) ->
     Rec (Node (here, unfold g left, unfold g right))
```

```
type ('a, 'b) t = Leaf | Node of 'a * 'b * 'b
type 'a tree = Rec of ('a, 'a tree) t
val unfold : ('a -> ('b, 'a) t) -> 'a -> 'b tree = <fun>
```

We saw this before!

```
let rec build f s =
    match f s with
          None -> Leaf
        | Some (a, left, right) ->
          Node (a, build f left, build f right)

let range (low, high) =
    if low > high
       then None
       else let mid = (low + high) / 2 in
            Some (mid, (low, mid - 1), (mid + 1, high))

let tree1to7 = build range (1, 7)
```

```
val build : ('a -> ('b * 'a * 'a) option) -> 'a -> 'b tree = <fun>
val range : int * int -> (int * (int * int) * (int * int)) option = <fun>
val tree1to7 : int tree = Node (4, Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Leaf)),
                                   Node (6, Node (5, Leaf, Leaf), Node (7, Leaf, Leaf)))
```

Rewrite it using unfold

```
let rec unfold g seed =
    match g seed with
          Leaf -> Rec Leaf
        | Node (here, left, right) ->
     Rec (Node (here, unfold g left, unfold g right))

let range (low, high) =
    if low > high
       then Leaf
       else let mid = (low + high) / 2 in
            Node (mid, (low, mid - 1), (mid + 1, high))

let balanced = unfold range
let tree1to7 = balanced (1, 7)
```

```
val unfold : ('a -> ('b, 'a) t) -> 'a -> 'b tree = <fun>
val range : int * int -> (int, int * int) t = <fun>
val balanced : int * int -> int tree = <fun>
val tree1to7 : ...
```

Look at a tree the other way!

```
type ('a, 'b) t = Leaf | Node of 'a * 'b * 'b

type 'a tree = Rec of ('a, 'a tree) t

val unfold : ('b -> ('a, 'b) t) -> 'b -> 'a tree = <fun>
```

- Type constructor $(\alpha, \beta)\ t$ defines (the only) two forms of a tree node.
- Type constructor $\alpha\ tree$ defines a tree as a recursive structure via type constructor $(\alpha, \beta)\ t$. The recursion occurs at the second type argument to t.
- A function of type $\beta \rightarrow (\alpha, \beta)\ t$ comprises the co-inductive step necessary for unfolding a value of type β to a value of type $\alpha\ tree$.

Fold and unfold for trees

```
let rec fold f tree =
    match tree with
          Rec Leaf -> f Leaf
        | Rec (Node (here, left, right)) ->
            f (Node (here, fold f left, fold f right))

let rec unfold g seed =
    match g seed with
          Leaf -> Rec Leaf
        | Node (here, left, right) ->
     Rec (Node (here, unfold g left, unfold g right))
val   fold : (('a, 'b) t -> 'b) -> 'a tree -> 'b = <fun>
val unfold : ('a -> ('b, 'a) t) -> 'a -> 'b tree = <fun>
```

Functions `fold` and `unfold` look strangely similar to each other!

Fold and unfold for trees, the third round (I)

```
type ('a, 'b) t = Leaf
               | Node of 'a * 'b * 'b

let map (f, g) t =
    match t with Leaf -> Leaf
              | Node (h, l, r) -> Node (f h, g l, g r)

type 'a tree = Rec of ('a, 'a tree) t

let down (Rec t) =     t
let up         t  = Rec t
```

```
val map : ('a->'b) * ('c->'d) -> ('a,'c) t -> ('b,'d) t = <fun>
val down : 'a tree -> ('a, 'a tree) t = <fun>
val up : ('a, 'a tree) t -> 'a tree = <fun>
```

Fold and unfold for trees, the third round (II)

```
type ('a, 'b) t = Leaf
                | Node of 'a * 'b * 'b

type 'a tree = Rec of ('a, 'a tree) t

let id x = x

let rec   fold f tree =  f (map (id,   fold f) (down tree))

let rec unfold g seed = up (map (id, unfold g) (   g seed))

val id : 'a -> 'a = <fun>
val fold : (('a, 'b) t -> 'b) -> 'a tree -> 'b = <fun>
val unfold : ('a -> ('b, 'a) t) -> 'a -> 'b tree = <fun>
```

Fold and unfold for trees — ever more functional!

Fold and unfold are functions that each takes in a (basis) function as the argument and return a (tree) function as the result.

```
let ($) f g x = f (g x)

let rec   fold f tree =  (f $ map (id,   fold f) $ down) tree
let rec unfold g seed = (up $ map (id, unfold g) $    g) seed

let this =   fold up
let that = unfold down


val ( $ ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val fold : (('a, 'b) t -> 'b) -> 'a tree -> 'b = <fun>
val unfold : ('a -> ('b, 'a) t) -> 'a -> 'b tree = <fun>
val this : 'a tree -> 'a tree = <fun>
val that : 'a tree -> 'a tree = <fun>
```

What is `this`, and what is `that`?

Functional diagram for fold

In the diagram, functions are arrows, and types are objects.

$$\begin{tikzcd}
\alpha\ tree \arrow[r, "\text{down}"] \arrow[d, "\text{fold } f"'] & (\alpha, \alpha\ tree)\ t \arrow[d, "\text{map }(\text{id}, \text{fold } f)"] \\
\beta & (\alpha, \beta)\ t \arrow[l, "f"']
\end{tikzcd}$$

```
let rec fold f tree = (f $ map (id, fold f) $ down) tree
```

Functional diagram for unfold

In the diagram, functions are arrows, and types are objects.

$$\begin{tikzcd}
\beta \arrow[r, "g"] \arrow[d, "\text{unfold } f"'] & (\alpha, \beta)\ t \arrow[d, "\text{map }(\text{id}, \text{unfold } f)"] \\
\alpha\ tree & (\alpha, \alpha\ tree)\ t \arrow[l, "\text{up}"']
\end{tikzcd}$$

```
let rec unfold g seed = (up $ map (id, unfold g) $ g) seed
```

Let's not forget lists!

```
type ('a, 'b) t = Null
                | Cons of 'a * 'b

type 'a list = Rec of ('a, 'a list) t

let rec   fold f list =  (f $ map (id,   fold f) $ down) list
let rec unfold g seed = (up $ map (id, unfold g) $    g) seed
```

$$\begin{tikzcd}
\alpha\ list \arrow[r, "\text{down}"] \arrow[d, "\text{fold } f"'] & (\alpha, \alpha\ list)\ t \arrow[d, "\text{map }(\text{id}, \text{fold } f)"'] \\
\beta & (\alpha, \beta)\ t \arrow[l, "f"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\beta \arrow[r, "g"] \arrow[d, "\text{unfold } f"'] & (\alpha, \beta)\ t \arrow[d, "\text{map }(\text{id}, \text{unfold } f)"'] \\
\alpha\ list & (\alpha, \alpha\ list)\ t \arrow[l, "\text{up}"']
\end{tikzcd}$$

2.2 Parametric Modules

Modules

- A module, also called *structure*, packs together related definitions (types, values, and even modules).
- The module name acts as a "name space" to avoid name conflicts.

```
module MyStack =
struct
  type 'a t = 'a list
  let empty = []
  let push elm stack = elm :: stack
  let pop  stack =
      match stack with
            [] -> None
          | head :: tail -> Some (head, tail)
end

let whatever = MyStack.push 1 []
```

Module interfaces

- A module interface, also called *signature*, specifies which components of a structure are accessible from the outside, and with which type.
- It acts as a contract between the user and the implementer of a module. Interface checking is always enforced in O'Caml.

```
module type STACK =
sig
  type 'a t
  val empty: 'a t
  val push: 'a -> 'a t -> 'a t
  val pop: 'a t -> ('a * 'a t) option
end

module S: STACK = MyStack
let whatever = S.push 1 S.empty
```

Parametric modules

- A parametric module, also called *functor*, is a structure parameterized by other structures.

- Type sharing and structure sharing constraints can be used to relate the arguments and the result.

```
module type QUEUE = STACK
module type S2Q = functor (S: STACK) -> QUEUE

module MakeQueue: S2Q = functor (S: STACK) ->
struct
  type 'a t = 'a S.t * 'a S.t
  let empty = (S.empty, S.empty)
  let push elm (front, rear) = (front, S.push elm rear)
  let pop (front, rear) =
      match S.pop front with
            Some (e, s) -> Some (e, (s, rear))
          | None ->  ...
end
```

Tree folding

```
type ('a, 'b) t = Leaf
                | Node of 'a * 'b * 'b

let map (f, g) t =
    match t with Leaf -> Leaf
               | Node (h, l, r) -> Node (f h, g l, g r)

type 'a tree = Rec of ('a, 'a tree) t
```

$$\begin{tikzcd}
\alpha\ \mathit{tree} \arrow[r, "\text{down}"] \arrow[d, "\text{fold } f"'] & (\alpha, \alpha\ \mathit{tree})\ t \arrow[d, "\text{map }(\text{id}, \text{fold } f)"] \\
\beta & (\alpha, \beta)\ t \arrow[l, "f"']
\end{tikzcd}$$

List folding

```
type ('a, 'b) t = Null
                | Cons of 'a * 'b

let map (f, g) t =
    match t with Null -> Null
```

```
                | Cons (hd, tl) -> Cons (f hd, g tl)

type 'a list = Rec of ('a, 'a list) t
```

$$\begin{tikzcd}
\alpha\ list \arrow[r, "\text{down}"] \arrow[d, "\text{fold } f"'] & (\alpha, \alpha\ list)\ t \arrow[d, "\text{map } (\text{id}, \text{fold } f)"] \\
\beta & (\alpha, \beta)\ t \arrow[l, "f"']
\end{tikzcd}$$

A fold for all seasons?

- Wanted: A way to describe the derivation of a unary type constructor by recursing over a binary type constructor, and to define the accompanying fold function at the same time.
- This is exactly what a parametric module can do!
- Input: a module with a binary type constructor and its map function.
- Output: a module with a unary type constructor, its map function, and its fold and unfold functions.

Module interfaces `FUN` and `FIX`

```
module type FUN =
sig
  type ('a, 'u)  t
  val map: ('a -> 'b) * ('u -> 'v) -> ('a, 'u) t -> ('b, 'v) t
end

module type FIX =
sig
  module Base: FUN
  type 'a t = Rec of ('a, 'a t) Base.t
  val down: 'a t -> ('a, 'a t) Base.t
  val up:   ('a, 'a t) Base.t -> 'a t

  val map:  ('a -> 'b) -> 'a t -> 'b t
  val fold: (('a, 'x) Base.t -> 'x) -> 'a t -> 'x
end
```

`Mu`, the fixed-pointing module

```
module type MU = functor (B: FUN) -> FIX with module Base = B

module Mu: MU = functor (B: FUN) ->
struct
  module Base = B
  type 'a t = Rec of ('a, 'a t) Base.t
  let down (Rec t) =     t
  let up        t  = Rec t

  let rec fold f (Rec t) = f   (Base.map (id, fold f) t)
  let rec map  f (Rec t) = Rec (Base.map (f,  map  f) t)
end
```

Module `Tree`

```
module T =
struct
  type ('a, 'b) t = Leaf
                  | Node of 'a * 'b * 'b

  let map (f, g) t =
      match t with Leaf -> Leaf
                 | Node (  h,   l,   r) ->
                   Node (f h, g l, g r)
end

module Tree = Mu(T)
```

Module List

```
module L =
struct
  type ('a, 'b) t = Null
                  | Cons of 'a * 'b
  let map (f, g) t =
      match t with Null -> Null
                 | Cons (  hd,   tl) ->
                   Cons (f hd, g tl)
end

module List = Mu(L)
```