Monads

Hype for Types

March 24, 2025

Mappables[1]

[1] Well, "functors", but that's already a thing in SML...

Shmategory Weary

Suppose we have some type `'a`, and we consider all of the functions it is "equipped" with (i.e. all functions of type `'a -> 'b` for some type `'b`).

Shmategory Weary

Suppose we have some type `'a`, and we consider all of the functions it is "equipped" with (i.e. all functions of type `'a -> 'b` for some type `'b`).

We can think of any functions `'a -> 'b` as being a relationship between the types `'a` and `'b`.

Shmategory Weary

Suppose we have some type `'a`, and we consider all of the functions it is "equipped" with (i.e. all functions of type `'a -> 'b` for some type `'b`).

We can think of any functions `'a -> 'b` as being a relationship between the types `'a` and `'b`.

Suppose we also wanted to "transform" the type `'a` into the type `'a list`.

Shmategory Weary

Suppose we have some type `'a`, and we consider all of the functions it is "equipped" with (i.e. all functions of type `'a -> 'b` for some type `'b`).

We can think of any functions `'a -> 'b` as being a relationship between the types `'a` and `'b`.

Suppose we also wanted to "transform" the type `'a` into the type `'a list`.

Question

How would this affect the function `'a -> 'b`? How do we perform the transformation such that the relationship between `'a` and `'b` is preserved?

From Types to Types

Consider the following transformation:

```
type 'a map_obj   = 'a list
fun     map_arr f = List.map f
```

where we

From Types to Types

Consider the following transformation:

```
type 'a map_obj    = 'a list
fun     map_arr f = List.map f
```

where we

- take a type `t` and turn it into type `t list`

From Types to Types

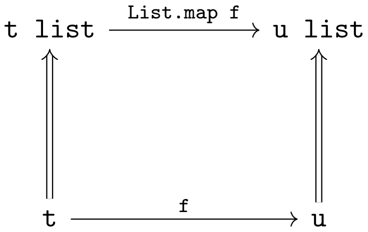
Consider the following transformation:

```
type 'a map_obj    = 'a list
fun     map_arr f = List.map f
```

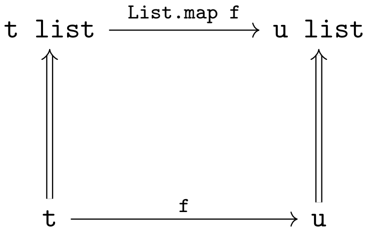
where we

- take a type `t` and turn it into type `t list`
- take a function `f : t -> u` and turn it into a function `List.map f : t list -> u list`

Visualizing Lists



Visualizing Lists



Key Idea

Even though the types 'a and 'b are now different, the relationship between them has been preserved by the transformation.

Mappables?

Why stop at lists? How would we define a transformation for any arbitrary destination type?

Mappables?

Why stop at lists? How would we define a transformation for any arbitrary destination type?

Definition?

A *mappable* `M` is the data:

- `type 'a t`

Mappables?

Why stop at lists? How would we define a transformation for any arbitrary destination type?

Definition?

A *mappable* `M` is the data:

- type `'a t`
- value `map : ('a -> 'b) -> 'a t -> 'b t`

Mappables?

Why stop at lists? How would we define a transformation for any arbitrary destination type?

Definition?

A *mappable* `M` is the data:

- type `'a t`
- value `map : ('a -> 'b) -> 'a t -> 'b t`

In other words:

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

Which map?

Let's go back to our list transformation.

Which map?

Let's go back to our list transformation.

There are countless functions we could have chosen for our transformation that have type `('a -> 'b) -> 'a list -> 'b list`:

Which map?

Let's go back to our list transformation.

There are countless functions we could have chosen for our transformation that have type `('a -> 'b) -> 'a list -> 'b list`:

```
type 'a map_obj    = 'a list

fun map_arr1 f =
  fn _ => []
fun map_arr2 f =
  fn l => List.map f (List.rev l)
fun map_arr3 f =
  fn []      => []
   | _::xs => List.map f xs
```

Not this map

If we want our transformation to maintain the relationship between the types, then some of those suggestions, while having the right type, are problematic.

Not this map

If we want our transformation to maintain the relationship between the types, then some of those suggestions, while having the right type, are problematic.

We would want our transformation on functions to maintain the following:

Not this map

If we want our transformation to maintain the relationship between the types, then some of those suggestions, while having the right type, are problematic.

We would want our transformation on functions to maintain the following:

- The identity function `id : 'a -> 'a` is transformed into the identity function `id' : 'a t -> 'a t` for the new type

Not this map

If we want our transformation to maintain the relationship between the types, then some of those suggestions, while having the right type, are problematic.

We would want our transformation on functions to maintain the following:

- The identity function `id : 'a -> 'a` is transformed into the identity function `id' : 'a t -> 'a t` for the new type
- For any functions `f : 'a -> 'b` and `g : 'b -> 'c`, `map (f o g) = (map f) o (map g)`

Mappables

Definition

A *mappable* `M` is the data:

- type `'a t`
- value `map : ('a -> 'b) -> 'a t -> 'b t`

Mappables

Definition

A *mappable* `M` is the data:

- type `'a t`
- value `map : ('a -> 'b) -> 'a t -> 'b t`
- upholds `map id` $=_{\texttt{'a t} \to \texttt{'a t}}$ `id`

Mappables

Definition

A *mappable* `M` is the data:

- type `'a t`
- value `map : ('a -> 'b) -> 'a t -> 'b t`
- upholds `map id` $=_{\texttt{'a t} \rightarrow \texttt{'a t}}$ `id`
- upholds `map f o map g` $=$ `map (f o g)`

Mappables

Definition

A *mappable* `M` is the data:

- type `'a t`
- value `map : ('a -> 'b) -> 'a t -> 'b t`
- upholds `map id` $=_{\texttt{'a t} \to \texttt{'a t}}$ `id`
- upholds `map f o map g` $=$ `map (f o g)`

In other words:

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

Optimization: Loop Fusion!

If we have:

```
int[n] arr;

for (int i = 0; i < n; i++)
  arr[i] = f(arr[i]);

for (int i = 0; i < n; i++)
  arr[i] = g(arr[i]);
```

[2] Not just for lists - any data structure with a "sensible" notion of map works!

Optimization: Loop Fusion!

If we have:

```
int[n] arr;

for (int i = 0; i < n; i++)
  arr[i] = f(arr[i]);

for (int i = 0; i < n; i++)
  arr[i] = g(arr[i]);
```

then it must be equivalent to:[2]

```
for (int i = 0; i < n; i++)
  arr[i] = g(f(arr[i]));
```

[2] Not just for lists - any data structure with a "sensible" notion of map works!

Some Example Mappables

- Lists
- Options
- Trees
- Streams
- Functions `int -> 'a`
- ...

Some Example Mappables

- Lists
- Options
- Trees
- Streams
- Functions `int -> 'a`
- ...

i.e., (almost) anything polymorphic.

Conclusion

It's a useful abstraction!

Monads

Descent into partial madness

Partial functions return options

Descent into partial madness

Partial functions return options:

- `sqrt : int -> int option`

Descent into partial madness

Partial functions return options:

- `sqrt : int -> int option`
- `div : (int * int) -> int option`

Descent into partial madness

Partial functions return options:

- `sqrt : int -> int option`
- `div : (int * int) -> int option`
- `head : 'a list -> 'a option`

Descent into partial madness

Partial functions return options:

- `sqrt : int -> int option`
- `div : (int * int) -> int option`
- `head : 'a list -> 'a option`
- `tail : 'a list -> 'a list option`

Descent into partial madness

Partial functions return options:

- `sqrt : int -> int option`
- `div : (int * int) -> int option`
- `head : 'a list -> 'a option`
- `tail : 'a list -> 'a list option`

How would we write the partial version of `tail_3`?

```
(* tail_3 : 'a list -> 'a list *)
fun tail_3 (_::_::_::L) = L
```

Composing partial functions

How would we write the partial version of `tail_3`?

```
tail_3 : 'a list -> 'a list option
```

Composing partial functions

How would we write the partial version of tail_3?

```
tail_3 : 'a list -> 'a list option
```

Partial madness!

```
fun tail_3 L0 =
  case tail L0 of
    NONE => NONE
  | SOME L1 =>
    (case tail L1 of
      NONE => NONE
    | SOME L2 => tail L2)
```

Composing partial functions

How would we write the partial version of `tail_3`?

```
tail_3 : 'a list -> 'a list option
```

Partial madness!

```
fun tail_3 L0 =
  case tail L0 of
    NONE => NONE
  | SOME L1 =>
    (case tail L1 of
      NONE => NONE
    | SOME L2 => tail L2)
```

What about `tail_5`?

Composing partial functions (again)

How would we write the partial version of tail_5?

```
tail_5 : 'a list -> 'a list option
```

Composing partial functions (again)

How would we write the partial version of `tail_5`?

```
tail_5 : 'a list -> 'a list option
```

If only...

```
val tail_5 = tail o tail o tail o tail o tail
```

However, `tail : 'a list -> 'a list option`, so we can't compose them like this.

Composing partial functions (again)

How would we write the partial version of `tail_5`?

```
tail_5 : 'a list -> 'a list option
```

If only...

```
val tail_5 = tail o tail o tail o tail o tail
```

However, `tail : 'a list -> 'a list option`, so we can't compose them like this.

Let's consider another kind of compose:

```
o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
<=< : ('b -> 'c opt) * ('a -> 'b opt) -> ('a -> 'c opt)
```

Composing partial functions (again)

How would we write the partial version of `tail_5`?

```
tail_5 : 'a list -> 'a list option
```

If only...

```
val tail_5 = tail o tail o tail o tail o tail
```

However, `tail : 'a list -> 'a list option`, so we can't compose them like this.

Let's consider another kind of compose:

```
o :  ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
<=< :  ('b -> 'c opt) * ('a -> 'b opt) -> ('a -> 'c opt)
```

Ta-da!

```
val tail_5 =
      tail <=< tail <=< tail <=< tail <=< tail
```

More than composition

Take a look at these Option functions:

```
type 'a t = 'a option
```

Option.composePartial

```
val <=< :
  ('b -> 'c t) * ('a -> 'b t) -> ('a -> 'c t)
```

More than composition

Take a look at these Option functions:

```
type 'a t = 'a option
```

Option.composePartial

```
val <=< :
  ('b -> 'c t) * ('a -> 'b t) -> ('a -> 'c t)
```

Option.map

```
val >>| : 'a t * ('a -> 'b) -> 'b t
```

More than composition

Take a look at these Option functions:

```
type 'a t = 'a option
```

Option.composePartial

```
val <=< :
  ('b -> 'c t) * ('a -> 'b t) -> ('a -> 'c t)
```

Option.map

```
val >>| : 'a t * ('a -> 'b) -> 'b t
```

Option.mapPartial

```
val >>= : 'a t * ('a -> 'b t) -> 'b t
```

More than composition

Take a look at these Option functions:

```
type 'a t = 'a option
```

Option.composePartial

```
val <=< :
  ('b -> 'c t) * ('a -> 'b t) -> ('a -> 'c t)
```

Option.map

```
val >>| : 'a t * ('a -> 'b) -> 'b t
```

Option.mapPartial

```
val >>= : 'a t * ('a -> 'b t) -> 'b t
```

Option.join

```
val join : 'a t t -> 'a t
```

More than composition

Take a look at these Option functions:

```
type 'a t = 'a option
```

Option.composePartial

```
val <=< :
  ('b -> 'c t) * ('a -> 'b t) -> ('a -> 'c t)
```

Option.map

```
val >>| : 'a t * ('a -> 'b) -> 'b t
```

Option.mapPartial

```
val >>= : 'a t * ('a -> 'b t) -> 'b t
```

Option.join

```
val join : 'a t t -> 'a t
```

Option.SOME

```
val return : 'a -> 'a t
```

Programming with bind

```
fun sum_options (a : int option) (b : int option) =
  a >>= fn a' =>
  b >>= fn b' =>
  SOME (a' + b')
```

Programming with bind

```
fun sum_options (a : int option) (b : int option) =
  a >>= fn a' =>
  b >>= fn b' =>
  return (a' + b')
```

Programming with bind

```
fun sum_options (a : int option) (b : int option) =
  a >>= fn a' =>
  b >>| fn b' =>
  a' + b'
```

Programming with bind

```
fun sum_options (a : int option) (b : int option) =
  a >>= fn a' =>
  b >>| fn b' =>
  a' + b'

fun tail_5 L0 =
  tail L0 >>= fn L1 =>
  tail L1 >>= fn L2 =>
  tail L2 >>= fn L3 =>
  tail L3 >>= fn L4 =>
  tail L4
```

Programming with bind

```
fun sum_options (a : int option) (b : int option) =
  a >>= fn a' =>
  b >>| fn b' =>
  a' + b'

fun tail_5 L0 =
  tail L0 >>= fn L1 =>
  tail L1 >>= fn L2 =>
  tail L2 >>= fn L3 =>
  tail L3 >>= fn L4 =>
  tail L4

fun tail_5 L0 =
  tail L0 >>= tail >>= tail >>= tail >>= tail
```

OCaml loves this!

```
open Option.Let_syntax

let sum_options (a : int option) (b : int option) =
  let%bind a' = a in
  let%map b' = b in
  a' + b'
```

OCaml loves this!

```
open Option.Let_syntax

let sum_options (a : int option) (b : int option) =
  let%bind a' = a in
  let%map b' = b in
  a' + b'

let tail_5 L0 =
  let%bind L1 = tail L0 in
  let%bind L2 = tail L1 in
  let%bind L3 = tail L2 in
  let%bind L4 = tail L3 in
  tail L4
```

More than just options: Or_error

```
open Or_error.Let_syntax

type 'a t = Ok of 'a | Error of Error.t

let divide (x : int) (y : int) : int Or_error.t =
  if y = 0
    then Or_error.error_string ":("
    else Or_error.return (x div y)

let _ : string Or_error.t =
  let%bind x = divide 10 3 in
  let%map y = divide x 0 in
  string_of_int (x + y)
```

More than just options: Deferred

```
open Async

type 'a t = 'a Deferred.t

let is_same (f1 : string) (f2 : string)
    : bool Deferred.t =
  let%bind contents1 = Reader.file_contents f1 in
  let%bind contents2 = Reader.file_contents f2 in
  return (String.equal contents1 contents2)
```

Useful pattern!

Key Idea

Monads are a useful programming tool!

```
signature MONAD =
  sig
    type 'a t
    val bind : 'a t * ('a -> 'b t) -> 'b t
    val return : 'a -> 'a t
  end
```

Monads are like burritos

A monad is a special kind of a mappable. A mappable F takes each type T and maps it to a new type FT. A burrito is like a mappable: it takes a type, like meat or beans, and turns it into a new type, like beef burrito or bean burrito.

Monads are like burritos

A mappable must also be equipped with a **map** *function that lifts functions over the original type into functions over the new type. For example, you can add chopped jalapeños or shredded cheese to any type, like meat or beans; the lifted version of this function adds chopped jalapeños or shredded cheese to the corresponding burrito.*

Monads are like burritos

A monad must also possess a **return** *function that takes a regular value, such as a particular batch of meat, and turns it into a burrito. The unit function for burritos is obviously a tortilla.*

Monads are like burritos

Finally, a monad must have a **bind** *function that takes a burrito, tells you how to shuffle the ingredients, and turns it into a new burrito. For example, given a burrito, you can unwrap the tortilla, add cheese, and rewrap it.*

Monads are like burritos

The **map**, **bind**, *and* **return** *functions must satisfy certain laws. For example, if* **B** *is already a burrito, and not merely a filling for a burrito, then* **B** *>>=* **return** *must be the same as* **B**. *This means that if you have a burrito, unwrap the burrito, and rewrap it in a new tortilla, its the same as the original burrito.*