# Category Theory (for Programmers)

Hype for Types

April 13, 2021

# Overview

- Lots of patterns appear in math and programming.
- Let's try to codify them!
- We'll end up with some cool abstractions and tricks that make programming simpler.

**Big Idea**

Category theory is the study of composition.

## What is a category?

# Some Algebraic Structures

What do these things have in common?

- Addition on natural numbers
- Multiplication on natural numbers
- String concatenation
- Appending lists
- Union on sets

Some observations:

- Binary operations
- Associative
- Identity element

# Monoids

## Definition

A *monoid* `M` is the data:

- type `t`
- value `z : t`
- value `f : t -> t -> t`
- upholds `f x z` = `f z x` = `x`
- upholds `f x (f y z)` = `f (f x y) z`

Ths abstraction is handy! e.g.:

```
Seq.reduce M.f M.z : t seq -> t
```

# Another Kind of Structure

What do *these* have in common?

- Functions on sets
- Monoid homomorphisms
- The $\leq$ relation on natural numbers
- Implications between propositions
- (Total) functions in SML

Some observations:

- "Things"
- Directed correspondences between the things
  - "Reflexive"
  - Compositional/ "transitive"

# Categories

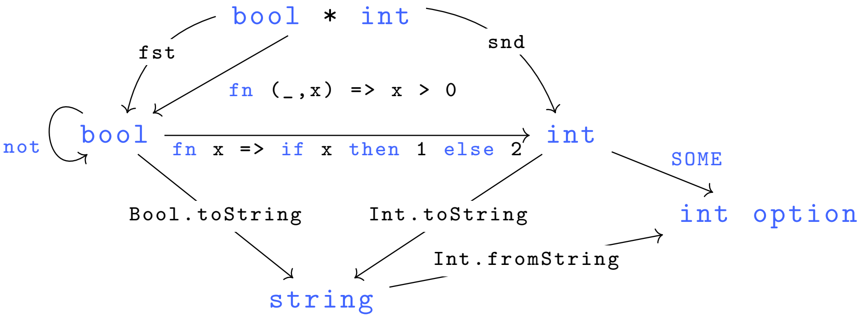
## Definition

A *category* $\mathcal{C}$ is the data:

- a collection of objects, $\mathrm{Ob}(\mathcal{C})$
- a collection of arrows, $\mathrm{Arr}(\mathcal{C})$
- for every arrow, a source $x \in \mathrm{Ob}(\mathcal{C})$
- for every arrow, a target $y \in \mathrm{Ob}(\mathcal{C})$
- for every object $x \in \mathrm{Ob}(\mathcal{C})$, an arrow $\mathrm{id}_x : x \to x$
- for every arrow $u : x \to y$ and $v : y \to z$, an arrow $u \circ v : x \to z$
- for every arrow $f : w \to x$, $g : x \to y$, $h : y \to z$, $f \circ (g \circ h) = (f \circ g) \circ h$

We'll focus on the category of SML types, with total functions as the arrows.

# The Category of SML Types




By convention, we omit:

- Identity arrows (self-loops at types)
- Compositions of arrows

# Mappables[1]

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

# From Category to Category

What would a transformation from category to category look like?

We must:

- turn objects into objects
- turn arrows into arrows

How about:

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

# Visualizing Lists

$$
\begin{tikzcd}
\texttt{t list} \arrow[r, "\texttt{List.map f}"] & \texttt{u list} \\
\texttt{t} \arrow[u, Rightarrow] \arrow[r, "\texttt{f}"] & \texttt{u} \arrow[u, Rightarrow]
\end{tikzcd}
$$

# Mappables?

## 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?

What if we picked:

```
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
```

Problems:

$$\texttt{map\_arr id [1,2,3]} \stackrel{?}{=} \texttt{[1,2,3]}$$

$$\texttt{map\_arr rev o map\_arr tl} \stackrel{?}{=} \texttt{map\_arr (rev o tl)}$$

# Mappables

## Definition

A *mappable* `M` is the data:

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

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!

# Option Map

What does an option look like as a mappable?

```
structure Option : MAPPABLE =
struct
  type 'a t = 'a option

  val map3 = fn f => fn
    NONE   => NONE
  | SOME x => SOME (f x)
end
```

Notice: this satisfies the desired identity and composition properties!

[3] This is built-in to SML as `Option.map`!

# Some More Example Mappables

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

i.e., (almost) anything polymorphic.

**Conclusion**

It's a useful abstraction.

# Monads

# Culinary Composition

We're used to a few combinators for composition:

- `op |> : 'a * ('a -> 'b) -> 'b`
  "pipe"
- `op >>> : ('a -> 'b) * ('b -> 'c) -> ('a -> 'c)`
  "forward composition"[4]

```
val getAvatar = fn token =>
  readLine ()
  |> parseInput
  |> requestData token
  |> toAvatar

val showAvatar =
  getAvatar >>> saveImage
```

Life is good.

[4] Flipped arguments from `op o`; arguably, more "natural"/easier to work with.

# Attack of the Real World

Here, we assumed:

```
val readLine    : unit -> string
val parseInput  : string -> packet
val requestData : token -> packet -> userData
val toAvatar    : userData -> image
val saveImage   : image -> filename
```

However, some of these steps could *fail*.

```
val readLine    : unit -> string option
val parseInput  : string -> packet option
val requestData : token -> packet -> userData option
val saveImage   : image -> filename option
```

```
val getAvatar = fn token =>
  readLine ()   (* string option *)
  |> parseInput (* string -> packet option *)
(* ↑ type error! *)
```

# First Attempt: Pain

```
val getAvatar = fn token =>
  case readLine () of NONE => NONE
  | SOME x1 => (
      case parseInput x1 of NONE => NONE
      | SOME x2 => (
          case requestData token x2 of NONE => NONE
          | SOME x3 => SOME (toAvatar x3)
        )
    )

val showAvatar =
  getAvatar >>> (fn NONE          => NONE
                  | SOME image => saveImage image)
```

## Observation

This is horrible! So much “plumbing” to propagate `NONE`. Before, the core logic was clearly present; now, it’s obscured.

# A New Kind of Composition

Let's reimagine our combinators as if everything produced an option.

```
val op |>  :  (* "pipe" *)
  'a          * ('a -> 'b        ) -> 'b
val op >>= :  (* "bind" *)
  'a option * ('a -> 'b option) -> 'b option
```

```
fun (x : 'a        ) |>  (f : 'a -> 'b        )
  : 'b         =
  f x
fun (x : 'a option) >>= (f : 'a -> 'b option)
  : 'b option =
  case x of
    NONE    => NONE
  | SOME y => f y
```

# A New Kind of Composition

```
val op >>> :
  ('a -> 'b        ) * ('b -> 'c        ) ->
    ('a -> 'c        )
val op >=> :
  ('a -> 'b option) * ('b -> 'c option) ->
    ('a -> 'c option)
```

```
fun (f : 'a -> 'b        ) >>> (g : 'b -> 'c        )
  : ('a -> 'c        ) =
  fn x => g (f x)

fun (f : 'a -> 'b option) >=> (g : 'b -> 'c option)
  : ('a -> 'c option) =
  fn x =>
    case f x of
      NONE   => NONE
    | SOME y => g y
```

# No more plumbing!

```
>>= : 'a option * ('a -> 'b option) -> 'b option
>=> : ('a -> 'b option) * ('b -> 'c option)
        -> ('a -> 'c option)
```

```
val readLine    : unit -> string option
val parseInput  : string -> packet option
val requestData : token -> packet -> userData option
val saveImage   : image -> filename option
```

```
val getAvatar = fn token =>
  readLine ()
  >>= parseInput
  >>= requestData token
  >>= (toAvatar >>> SOME)  (* wrap via SOME *)

val showAvatar =
  getAvatar >=> saveImage
```

# Formalizing Burritos

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

As usual, there are some other invariants - the "monad laws"[5] - which make `return` and `>>=` behave "in the expected way".

```
structure Option : MONAD =
  struct
    type 'a t = 'a option
    val return = SOME
    fun x >>= f = case x of NONE => NONE | SOME y
    => f y
  end
```

[5] https://wiki.haskell.org/Monad_laws

# Some examples...

## Big Idea

Lots of common types ascribe to `MONAD`.

| `type 'a t =` | For when your functions can produce... |
|---|---|
| `'a option` | "failure" via `NONE` |
| `('a, string) either` | "failure" with an error string |
| `unit -> 'a` | a "lazy" output |
| `'a list` | multiple results |
| `'a * string` | a log string (always) |
| `state -> state * 'a` | an updated state, given a state |

# Log Monad

```
structure LogMonad : MONAD =
  MkMonad (
    type 'a t = 'a * string

    fun return (x : 'a) : 'a t = (x, "")

    fun ((x, log) : 'a t) >>= (f : 'a -> 'b t)
      : 'b t =
      let
        val (y, log') = f x
      in
        (y, log ^ log')
      end
  )
```

# What about >=>?

Turns out, we can define it in terms of >>= (and vice versa).

In fact, given `return` and one of the following three functions, the other two can be derived:

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

## Theorem

Every `MONAD` is a `MAPPABLE`.

Given `return` and any of the previous three functions, we can implement

```
val map : ('a -> 'b) -> ('a t -> 'b t)
```

with the desired properties.

# Aside: Imperative Programming

Monads look like a *generalization* of imperative programming.

```
val getAvatar = fn token =>
  readLine >>= (fn input =>
    parseInput input >>= (fn parsed =>
      requestData token parsed >>= (fn data =>
        return (toAvatar data)
      )
    )
  )  (* looks like CPS! *)
```

```
val getAvatar = fn token =>
  do
    input  <- readLine ()
    parsed <- parseInput input
    data   <- requestData token parsed
    return (toAvatar data)
```

This syntactic sugar isn't present in SML, but it "might as well be". (It's in Haskell!)

# Conclusion

# Conclusion

- Category theory lets us think abstractly about a variety of mathematical structures.
- As programmers/type theorists, we can take advantage of category theoretic "signatures" to **reduce boilerplate code**.
- Most common parameterized types are `MAPPABLE`. Just like `List.map` is handy, so are other `map` functions!
- Many parameterized types which are `MONAD`s. We can use this to get helper functions for free, letting us **focus on the "business logic"** rather than peripheral implementation details.