Category Theory (for Programmers)

Hype for Types

October 10, 2020

What is a category?

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
```

Examples of Monoids

There are many monoids. For example:

- Natural numbers with zero, addition
- Natural numbers with one, multiplication
- Strings with empty string, string concatenation
- Lists with empty list, appending
- Sets with empty set, union

Categories

Definition

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

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

Examples of Categories

There are many categories. For example:

- Objects are sets, arrows are functions
- Objects are groups, arrows are group homomorphisms
- Objects are "numbers", arrows are for $\leq$
- Objects are propositions, arrows are implications
- Objects are SML types, arrows are (total) functions

We'll focus on the last one.

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?

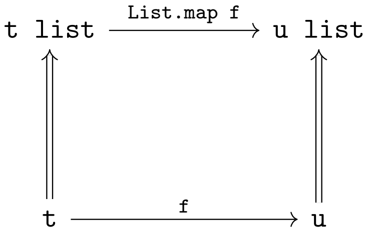
We must:

- 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



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:

```
map_arr Fn.id [1,2,3] =?= [1,2,3]

map_arr List.length o map_arr Int.toString
                =?=
map_arr (List.length o Int.toString)
```

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(i);

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

then it must be equivalent to:[2]

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

i.e., (almost) anything polymorphic.

Conclusion

It's a useful abstraction.

Monads

Descent into partial madness

Partial functions return options:

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

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 opt
```

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 opt
```

If only...

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

Another kind of compose

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

Ta-da!

```
fun f <=< g =
  (fn NONE => NONE | SOME x => f x) o g
```

More than a composition

Some useful versions of common tools

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

Compose

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

Apply

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

Flatten

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

```
bind : 'a t * ('a -> 'b t) -> 'b t
```

```
type 'a t = 'a option
fun x >>= f = case x of SOME x => f x
                       | NONE => NONE

type 'a t = 'a list
fun xs >>= f = List.concat (List.map f xs)

type 'a t = 'a * string
fun (x,a) >>= f = let (y,b) = f x
                  in (y,a^b) end

type 'a t = unit -> 'a
fun x >>= f = fn () => f (x()) ()

datatype 'a t = Ret of 'a | Err of exn
fun x >>= f = case x of Ret a => f x
                       | Err x => Err x
```

Programming with Monads

```
readInput      : stream -> string option
parseUsername  : string -> string option
getUserFromId  : string -> user option
getAvatar      : user   -> image option

SOME TextIO.stdIn
  >>= readInput
  >>= parseUsername
  >>= getUserFromId
  >>= getAvatar
```

Parallel: Imperative Programming

```
inString <- SOME TextIO.stdIn
userId <- parseUsername inString
user <- getUserFromId userId
avatar <- getAvatar user
```

Useful pattern!

Key Idea

Monads are a useful programming tool!

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