Category Theory (for Programmers)

Hype for Types

October 31, 2023

What is a category?

Categories

Definition

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

- a collection of objects, $\text{Ob}(\mathcal{C})$

Categories

Definition

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

- a collection of objects, $\text{Ob}(\mathcal{C})$
- a collection of arrows, $\text{Arr}(\mathcal{C})$

Categories

Definition

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

- a collection of objects, $\text{Ob}(\mathcal{C})$
- a collection of arrows, $\text{Arr}(\mathcal{C})$
- for every arrow, a source $x \in \text{Ob}(\mathcal{C})$

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 $y \in \mathrm{Ob}(\mathcal{C})$

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 $\mathrm{id}_x : x \to x$

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 $v : x \to y$ and $u : y \to z$, an arrow $u \circ v : x \to z$

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 $v : x \to y$ and $u : y \to z$, an arrow $u \circ v : x \to z$
- for every arrow $f : y \to z$, $g : x \to y$, $h : w \to x$, $f \circ (g \circ h) = (f \circ g) \circ h$

Examples of Categories

There are many categories. For example:

Examples of Categories

There are many categories. For example:

- Objects are sets, arrows are (total function) functions

Examples of Categories

There are many categories. For example:

- Objects are sets, arrows are (total function) functions
- Objects are groups, arrows are group homomorphisms

Examples of Categories

There are many categories. For example:

- Objects are sets, arrows are (total function) functions
- Objects are groups, arrows are group homomorphisms
- Objects are natural numbers, arrows are for $\leq$

Examples of Categories

There are many categories. For example:

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

Examples of Categories

There are many categories. For example:

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

Examples of Categories

There are many categories. For example:

- Objects are sets, arrows are (total function) functions
- Objects are groups, arrows are group homomorphisms
- Objects are natural 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 $\mathcal{C}_1$ to category $\mathcal{C}_2$ look like?

From Category to Category

What would a transformation from category $\mathcal{C}_1$ to category $\mathcal{C}_2$ look like?

We must:

- turn objects in $\mathcal{C}_1$ into objects in $\mathcal{C}_2$
- turn arrows in $\mathcal{C}_1$ into arrows in $\mathcal{C}_2$

From Category to Category

What would a transformation from category $\mathcal{C}_1$ to category $\mathcal{C}_2$ look like?

We must:

- turn objects in $\mathcal{C}_1$ into objects in $\mathcal{C}_2$
- turn arrows in $\mathcal{C}_1$ into arrows in $\mathcal{C}_2$

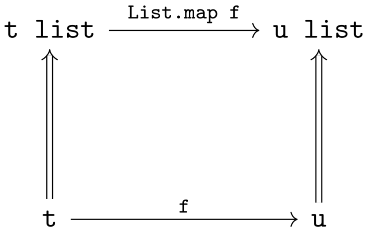
Suppose we have the category of SML types and functions. How about we define a transformation as:

```
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 an arrow `f : t -> u` and turn it into an arrow `List.map f : t list -> u list`

Visualizing Lists



Mappables?

Definition?

A *mappable* `M` is the data:

- `type 'a t`

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`

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
```

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

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` $=_{\texttt{'a t} \to \texttt{'a t}}$ `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 opt`

Descent into partial madness

Partial functions return options:

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

Descent into partial madness

Partial functions return options:

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

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`

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
```

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)
```

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
```

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
```

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
```

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 t)
```

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 t)
```

Apply

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

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 t)
```

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
```

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)
```

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
```

```
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()) ()
```

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
```

EXAMPLE: ERRORS

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

fun divide(x : int, y : int) : int t =
  if y = 0 then NONE
  else SOME (x div y)

val _: string t =
  bind (divide (10, 3), fn x =>
  bind (Int.fromString "0", fn y =>
  bind (divide (x, y), fn z =>
  SOME (Int.toString (x + y + z)))))
```

EXAMPLE: PRINTING

```
type 'a t = string * 'a
fun bind (e : 'a t, f : 'a -> 'b t) : 'b t =
  let
    val (s1, a) = e
    val (s2, b) = f a
  in
    (s1 ^ s2, b)
  end
```

EXAMPLE: PRINTING

```
fun print (s : string) : unit t =
  (s, ())
fun add (x : int, y : int) : int t =
  ("adding", x + y)
val _ : int t =
  bind (print "hi", fn () =>
  bind (add (20, 22), fn n =>
  ("done", n)))
(* result : ("hiaddingdone", 42) : int t *)
```

Programming with Monads

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

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

Monads are like burritos

A monad is a special kind of a functor. A functor F takes each type T and maps it to a new type FT. A burrito is like a functor: 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 functor 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 possess a **join** *function that takes a ridiculous burrito of burritos and turns them into a regular burrito. Here the obvious join function is to remove the outer tortilla, then unwrap the inner burritos and transfer their fillings into the outer tortilla, and throw away the inner wrappings.*

Monads are like burritos

The **map**, **join**, *and* **return** *functions must satisfy certain laws. For example, if* **B** *is already a burrito, and not merely a filling for a burrito, then* **join(return(B))** *must be the same as* **B**. *This means that if you have a burrito, and you wrap it in a second tortilla, and then unwrap the contents into the outer tortilla, the result is the same as what you started with.*