Type Theory (via Agda)

@ClocksSugars

06/29/26

clockssugars.blog/appliuni/prelims/proptypes.html
clockssugars.blog/articles/0326-homotopy-type-theory/

The Plan

I am not a computer scientist! I'm not even a type theorist! I'm a mathematician who learned all of this to study homotopy type theory! This talk will reflect that!

This is going to be very forgiving (or boring!)

1. What is an (inductive) type (and why we want them)
2. How do inductive types link to mathematical concepts
3. How can types reflect propositional logic
4. Proofs as Functions (obvious edition)
5. Dependent types and their role in propositional logic
6. Agda's hole-based development vs tactics (live coding)

What is a type

Let's start with some things we think are close to what a type should represent.

- Set
- Category
- ?

It's hard not to say a 'type' should be a type *of things*.
This is a *bad* way to think about types!

What is a type

Example: Something that is *not* a type, a group!

- A *set* of objects G with a binary operation $\square \times \square$

- (Closure) The binary operation can't escape G, so $g, h \in G \implies g \cdot h \in G$ (so it is *closed*)

- (Identity) G has an object which does nothing under the operation, so $e \in G$ satisfies $ge = eg = g$ for all $g \in G$

- (Associativity) $(gh)k = ghk = g(hk)$

- (Inverses) Every object $g \in G$ has an opposite object $g^{-1} \in G$ so that $gg^{-1} = g^{-1}g = e$

If I have a group G, what do I *know*? I can multiply, invert, change order of operations

If I know $g \in G$, what data do I know about g?

What is a type

Example: C-style enum

```
enum Alphabetic {
    A,
    B,
    C,
    ...
    Z
}
```

If I tell you I'm going to give you a value x : `Alphabetic`, what do you know?

For the purposes of this talk, the *value* of a type is that it tells you what a thing *is* if you know its type. We expect to get this out of *inductive types*.

What is a type

A type is *not* what the type is *supposed* to represent. It is the ways you make its members (and its names (unless you take the univalence axiom))

When I tell you I have an inductive type, you should expect I have *some* way to *concretely* enumerate the members of that type.

Then what are some non-inductive types?

Type of types?

Function types?

If I knew all the types, I could enumerate the function types using $\to$ as a constructor

$$\square \to \square : \texttt{Type} \to \texttt{Type} \to \texttt{Type}$$

What is a type

A type is *not* what the type is *supposed* to represent. It is the ways you make its members (and its names (unless you take the univalence axiom))

When I tell you I have an inductive type, you should expect I have *some* way to *concretely* enumerate the members of that type.

Then what are some non-inductive types?

Type of types?

Type of functions?

Maybe if I knew all the types, I could enumerate the function types using $\to$ as a constructor (right-associative)

$$\square \to \square : \Big(\texttt{Type} \to \big(\texttt{Type} \to \texttt{Type}\big)\Big)$$

What is a type

Example: Option/Maybe types from Rust/Haskell are (generic) inductive types

```rust
enum Option<T> {
    None,
    Some(T)
}
```

```haskell
data Maybe T = Nothing | Just T
```

If I tell you I have an element a: `Option<Alphabetic>` you know it has to be one of `None`, `Some(A)`, `Some(B)`, ..., `Some(Z)`

This means you can write a function from `Option<Alphabetic>` to another type by first asking *how the input was made*.

What is a type

Some things about inductive types, since a thing's *type* tells you what it *is*:

- A thing always belongs to only **one** type, which tells how it's made

- There are no subtypes or parent types (but we can make injections into smaller types)

- In light of the above, $x \in X$ is a *proposition*, not a type assignment. We write $x : X$ to say x is *of* type X, and this is not falsifiable like $x \notin X$.

Inductive types vs Categories

We are *not* going to spend long on category theory!
Three things:

1. Categories are always defined not just with objects (groups, sets, vector spaces) but with functions which preserve the properties of the objects (homomorphisms, functions, linear maps)

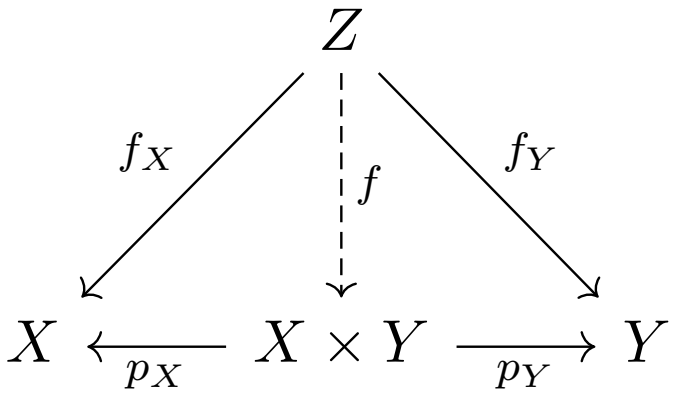
2. Category theory allows us to define common mathematical patterns as *diagrams* of *arrows* between objects

3. Category theory tells us that we often get a meaningful object simply by taking one of those diagrams and *flipping* the arrows. These *dual* patterns are usually prefixed with *co-* (covectors, cohomology, coproduct, etc.)

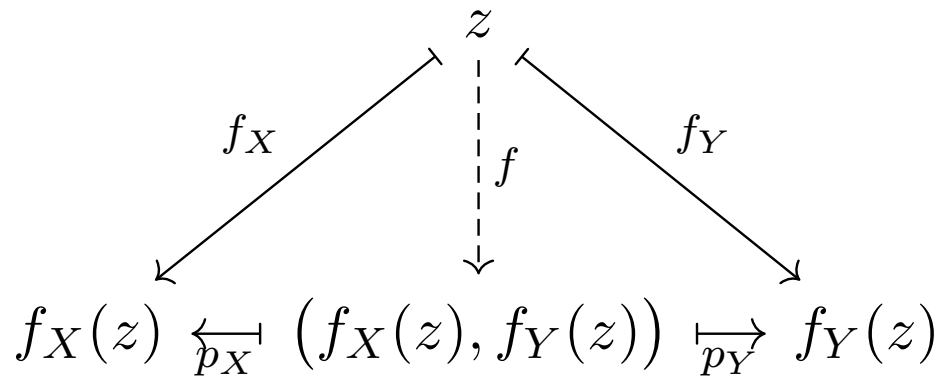
Inductive types vs Categories

Example: (Cartesian) Product

Object level



Element level



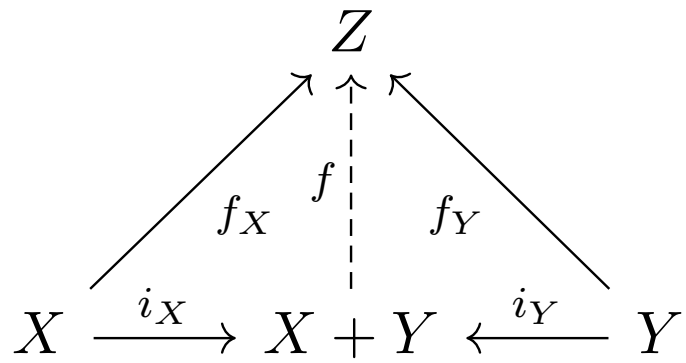
If f_1, f_2, p_X, p_Y exist, f exists and $X \times Y$ is a product object.

Inductive types vs Categories

```
1  enum Pair<X,Y> { pair(X,Y) }
2
3  fn P_X<X,Y>( pair: Pair<X,Y> ) -> X {
4      match pair { Pair::pair(x,y) => x }
5  }
6
7  fn P_Y<X,Y>( pair: Pair<X,Y> ) -> Y {
8      match pair { Pair::pair(x,y) => y }
9  }
10
11 fn merge<X,Y,Z>(
12     f_X: fn(Z) -> X,
13     f_Y: fn(Z) -> Y
14 ) -> (impl Fn(Z) -> Pair<X,Y>) {
15     fn f( z: Z ) -> Pair<X,Y> {
16         Pair::pair( f_X(z), f_Y(z) )
17     };
18     return f
19 }
```

Inductive types vs Categories

Duality time! Example: Co-product



If i_X, i_Y, f_X, f_Y exist, f exists and $X + Y$ is the coproduct, or *sum* object.

What is $X + Y$? What does it contain?

Inductive types vs Categories

You may know this as the `Either`/`Result` type from Haskell/Rust!

```
enum Sum<X,Y> {
    inl(X),
    inr(Y)
}

fn merge<X,Y,Z>(
    f_X: fn(X) -> Z,
    f_Y: fn(Y) -> Z
) -> (impl Fn(Sum<X,Y>) -> Z) {
    fn f( xory : Sum<X,Y> ) {
        match xory {
            inl(x) => f_X(x),
            inr(y) => f_Y(y)
        }
    };
    f
}
```

Inductive types vs Categories

The point:

- Mathematics defines things **restrictively** (e.g. a set G which obeys axioms x,y,z,...)

- Inductive types define things **prescriptively** (e.g. a type `Coprod` has *constructors* or *introduction rules* `inl`, `inr`)

- Knowing exactly how the elements of an inductive type are made means we get to write *elimination rules* for all the ways you would *destruct* a thing and make something else

Proposition Types

What do we need for propositional logic? True, False, `OR`, `AND`, `NOT`, logical implication, ...

A proposition type is a type which is true when it is *inhabited*.

We call members of a proposition type a *witness* or *evidence* or *proof*.

```
enum True { // Also called Unit type, or 𝟙, or () in
    Rust
    Star
}
```

What would a type be if it were defined as false?

```
enum False {} // Also called Empty or 0
```

Proposition Types

How can we use what we've seen so far to construct logical operations?

- P `AND` Q should be inhabited if *both* P and Q can find a witness... $[P \wedge Q] = [P \times Q]$
- P `OR` Q should be inhabited if *either* P or Q can find a witness... $[P \vee Q] = [P + Q]$
- P `implies` Q should be inhabited if having an element of P gives you an element of Q... $[P \Rightarrow Q] = [P \to Q]$
- `NOT` P should be only inhabited if there are no inhabitants of P...

A function must associate an output to every input. What happens if the input is `Empty`? What happens if the *output* is `Empty`? $[\neg P] = [P \to \mathbf{0}]$.

Proofs as Functions

We can now write very simple proofs as if they were programs, such as P or Q and not P implies Q

```
fn trivial_func<Q>(x: False) -> Q { match x {} }

fn theorem<P,Q>(
    s: Sum<P,Q>,
    g: fn(p: P) -> False
) -> Q
{
    match s {
        Sum::inl(p) => trivial_func(g(p)),
        Sum::inr(q) => q
    }
}
```

This function is a witness of $P + Q \to (\neg P) \to Q$
In fact, rust accepts the above as valid code!

Proofs as Functions

Here's the same thing in haskell!

```
1  data Sum a b = Inl a | Inr b
2
3  data False
4  type Neg p = p -> False
5
6  triv :: False -> q
7  triv x = case x of
8
9  theorem :: Sum p q -> Neg p -> q
10 theorem s g = case s of
11    Inl p -> triv ( g p )
12    Inr q -> q
```

Look closely because Agda's syntax is very close to haskell!

Dependent Types

We never finished our list of logical operations. How do we represent

- Conditions $P(x)$ for some $x : X$?
- Universal Quantifiers "for all $x\colon X$, $P(x)$"
- Existential Quantifiers "there exists $x\colon X$ with $P(x)$"

We can't do these without **dependent types**! These are

- Type families $P\colon X \to \texttt{Type}$
- Dependent functions f with input type X and output type $P(x)$ changing depending on the input $x\colon X$
- Dependent pairs (x, p) with the type of the second entry $p\colon P(x)$ changing depending on which $x\colon X$ is in the first entry.

Dependent Types

Dependent functions are called Π-types, like big-product, like big-product-type, or big-logical-`AND`. e.g.

$$f : \prod_{x : X} P(x)$$

f must assign a proof to every member of the $P : X \to \texttt{Type}$ type family in order to fulfill its obligations as a *function*.

Dependent pairs are called Σ-types, like big-sums, like big-sum-types, or big-logical-`OR`. e.g.

$$(x, p) : \sum_{x : X} P(x)$$

(x, p) is an *example* of a proof for *at least one* member of the type family $P : X \to \texttt{Type}$, and which one it proved.

Martin-Lof Type Theory language for Propositions

$\top := \mathbf{1}$ read as 'true' or 'top' or 'tautology'

$\bot := \mathbf{0}$ read as 'false' or 'bottom' or 'contradiction'

$\wedge := \times$ read as 'and',

$\vee := +$ read as 'or',

$\forall := \Pi$ read as 'for all...'

$\exists := \Sigma$ read as 'there exists...'

$\Rightarrow := \rightarrow$ read as 'implies', e.g. $P \Rightarrow Q$ is the type of functions taking witnesses $p\colon P$ and producing some witness $q\colon Q$, along with $(P \Leftarrow Q) = (Q \rightarrow P)$,

$\neg P := (P \rightarrow \bot)$ read as 'negation' or 'not...'

$(P \Leftrightarrow Q) := (P \rightarrow Q) \times (Q \rightarrow P)$, that is, we say P 'if and only if' Q