# Programming for Category Theorists

## Categorical Semantics for Functional Programming Languages

Chris Grossack
(they/them)

November 24, 2020

- This is going to be an entry level talk
- We're going to describe some deep connections between Category Theory and Functional Programming
- This should motivate some of the constructions for people new to the field
- It may also provide a new lens for viewing some constructions you're already familiar with
- Ideally, by the end of the talk, you'll feel like some seemingly abstract constructions arise very naturally in the context of programming
- In the interest of keeping the talk elementary, I've left out the beautiful connection this topic has with proof theory.
- You can find the slides posted on my website grossack.site

# Let's Start!

- Let's start with a toy programming language.
- We'll add features over the course of the talk.
- In the beginning, we have a glorified calculator:

## Toy Language: Mk. I

$$\mathtt{e} ::= \mathtt{T} \mid \mathtt{F} \mid \mathtt{n} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$$

- This is called a grammar, and defines the syntax of our language
- Informally, we can use any of the symbols separated by $|$, and $e_i$ should be interpreted recursively as a pre-existing expression.
- Some examples should show exactly what is meant

## Toy Language: Mk. I

$$\mathtt{e} ::= \mathtt{T} \mid \mathtt{F} \mid \mathtt{n} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$$

- $\mathtt{T}$
- $5$
- $5 + 3$
- $\mathtt{if\ T\ then\ 4\ else}\ 2 + 3$
- $(\mathtt{if\ F\ then\ 3\ else\ 4}) + 0$

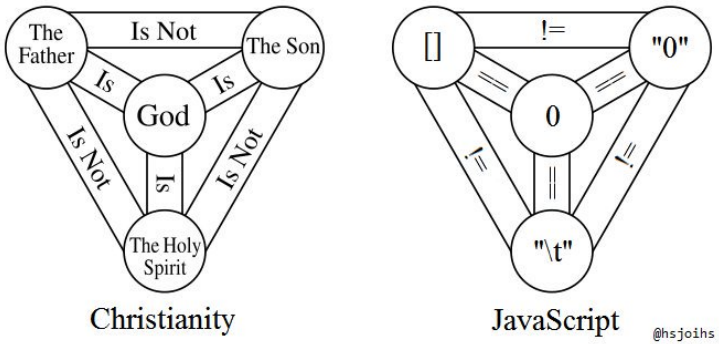
- We currently allow "bad" expressions too
- $\mathtt{F} + (T + 3)$
- $\mathtt{if\ 4\ then\ T\ else\ F}$

## Important Question:

What do we do about "bad" expressions?

## Answer 1:

- Accept the expressions for who they are
- They're not bad... Just misunderstood
- Pick a convention to interpret them: e.g.
  - $F \approx 0$, $T \approx 1$
  - $0 \approx F$, $n \approx T$
- This was popular in the '90s and '00s:
  - Javascript, Python, Perl, PHP, etc.
- Falling out of favor, as it leads to tricky bugs and weird behavior
- This led to a rather famous meme:




(You can find this and other strange behavior at javascriptwtf.com)

## Answer 2:

- Find a way to exclude these expressions.
- They aren't loved. They aren't wanted.
- This is the current trend in programming languages

## Types

- What went wrong?
- $\mathtt{T} + 5$ isn't meaningful because $T$ is the wrong *type* of object.
- Let's add Types to our langauge and only allow expressions whose types are "what we expect"

## Toy Language: Mk. II

$$\mathtt{e} ::= \mathtt{T} \mid \mathtt{F} \mid \mathtt{n} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$$

$$\tau ::= \mathbb{B} \mid \mathbb{N}$$

- We also have to provide *rules* which say what the type of an expression is, and how they interact.
- I'll show these exactly once, for cultural growth. But they quickly get complicated, so I'll omit them in the future and make sure the typing rules are clear from context.

## Toy Language: Mk. II

$$\mathtt{e} ::= \mathtt{T} \mid \mathtt{F} \mid \mathtt{n} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$$

$$\tau ::= \mathbb{B} \mid \mathbb{N}$$

$$\frac{}{\mathtt{T} : \mathbb{B}} \qquad \frac{}{\mathtt{F} : \mathbb{B}} \qquad \frac{}{n : \mathbb{N}}$$

$$\frac{e_1 : \mathbb{N} \quad e_2 : \mathbb{N}}{e_1 + e_2 : \mathbb{N}} \qquad \frac{e_1 : \mathbb{B} \quad e_2 : \tau \quad e_3 : \tau}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

(This is where the beautiful connection with proof theory lies)

- This is a pretty silly programming language
- It's really a calculator that only knows about addition
- We aren't *programming* until we can define functions
- Let's go ahead and add some syntax for that!

# The infamous $\lambda$-calculus

- We first allow expressions to contains variables: e.g.
  - $x + 5$
  - $\texttt{if } b \texttt{ then } y + 1 \texttt{ else } 3 + 2$
- Then we add syntax to bind a variable as the argument of a function:
  - $\lambda(x : \mathbb{N}).x + 5$
  - $\lambda(b : \mathbb{B}).\lambda(y : \mathbb{N}).\texttt{if } b \texttt{ then } y + 1 \texttt{ else } 3 + 2$
- We often want a function to return *multiple* outputs
- So we add syntax for tuples of expressions:
  - $(5, \texttt{T})$

## Toy Language: Mk III.

$$\mathtt{e} ::= \mathtt{T} \mid \mathtt{F} \mid \mathtt{n} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$$
$$\mid \mathtt{x} : \tau \mid \lambda(\mathtt{x} : \tau).\mathtt{e} \mid \mathtt{e}_1(\mathtt{e}_2) \mid (\mathtt{e}_1, \mathtt{e}_2) \mid \pi_{\mathtt{L}}(\mathtt{e}) \mid \pi_{\mathtt{R}}(\mathtt{e})$$

$$\tau ::= \mathbb{B} \mid \mathbb{N} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$$

- $\lambda(x : \mathbb{N}).x : \mathbb{N} \to \mathbb{N}$
- $(\lambda(x : \mathbb{N}).x + 3)(4) : \mathbb{N}$
- $(5, \mathtt{T}) : \mathbb{N} \times \mathbb{B}$
- $\pi_R(5, \mathtt{T}) : \mathbb{B}$
- $\lambda(b : \mathbb{B}).\mathtt{if}\ b\ \mathtt{then}\ (\lambda(x : \mathbb{N}).x + 1)\ \mathtt{else}\ (\lambda(x : \mathbb{N}).x + 2)$
  $: \mathbb{B} \to (\mathbb{N} \to \mathbb{N})$

# Some Pressing Questions

- Ok, I hear you asking
- This is cool and all, but I was promised Category Theory
- More pressingly, how do we *evaluate* one of these programs?
- There's an "obvious" way to do it, but how do we make it formal?
- Why can't $3 + 5 \rightsquigarrow 15$?

These are some very astute questions, Socratic Dialogue partner!

# Syntax and Semantics

- If you hang out with me long enough, you *will* hear me rant about $\sim \star \sim$ Syntax and Semantics $\sim \star \sim$
- Syntax is the rules for writing things down
- Semantics tell you what the syntax *means*
- We've discussed the Syntax of our programming language, but we haven't said anything about the Semantics!

# S'n'S: Pt 2. Enter Categories

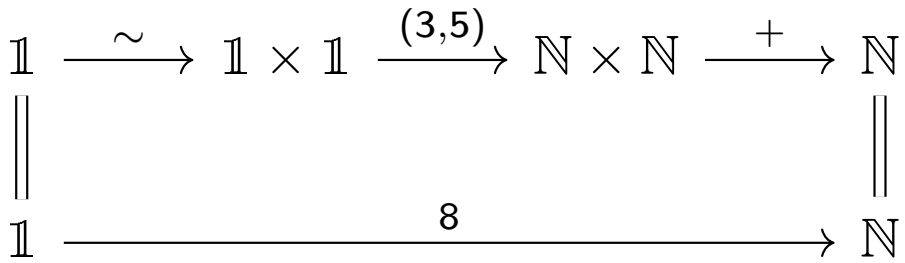
- The important observation is this:
- We can use *Categories* to interpret our programming language
- We will have an object for every type $\tau$
- And we will have an "element" for every program of type $\tau$
  - Here by "element", I mean an arrow $\mathbb{1} \xrightarrow{p} \tau$ from the terminal object
- As we make our programming language more expressive, we will need more structure on the categories in which we can interpret it
- Let's see some examples, though.

Let's start small: How can we interpret some of our calculator programs in Set?

## Semantics in Set

- We first pick a set for each type. Since we're just starting out, take
  - $\mathbb{B} = \{0, 1\}$
  - $\mathbb{N} = \{0, 1, 2, \ldots\}$
- Then we pick an *element* for each value
  - $\mathtt{T} = 1 \in \mathbb{B} \quad \mathtt{F} = 0 \in \mathbb{B}$
  - $\mathtt{n} = n \in \mathbb{N}$
- Then, for each recursively defined term, we pick a function defined on the relevant set
  - $\mathbb{N} \times \mathbb{N} \xrightarrow{+} \mathbb{N}$
  - $\mathbb{B} \times \mathbb{N} \times \mathbb{N} \xrightarrow{\mathtt{if-then-else}_{\mathbb{N}}} \mathbb{N}$
  - $\mathbb{B} \times \mathbb{B} \times \mathbb{B} \xrightarrow{\mathtt{if-then-else}_{\mathbb{B}}} \mathbb{B}$
- Again, let's pick the obvious functions based on the syntax

Now we can interpret programs by composing these arrows! The fact that this diagram commutes expresses the fact that $5 + 3$ evaluates to 8




The top path is the program $5 + 3$, and the bottom is the program 8. Commutativity says exactly that these are actually the same program!

(There is a subtle point here about the difference between equality of programs and evaluation that I'm going to gloss over)

- How do we interpret $\lambda(x : \tau).e$?
- It should be some object whose elements correspond to arrows...
- For instance, if $e = x + 5$, then $e$ is an arrow $\mathbb{N} \xrightarrow{x+5} \mathbb{N}$
- Then we want $\mathbb{1} \xrightarrow{\lambda(x:\mathbb{N}).x+5} \mathbb{N}^{\mathbb{N}}$ to be an element of $\mathbb{N}^{\mathbb{N}} = \{f : \mathbb{N} \to \mathbb{N}\}$
- As an aside, notice $e$ is not an element. The elements in our category correspond with the *closed terms*. That is, the programs with no free variables.
- More to the point – what categorical structure do we need to interpret $\lambda$ terms in this way?

# A suggestive detour

At this point I will, for no apparent reason, recall the notion of a Cartesian Closed Category

## Definition

A Category is called Cartesian Closed whenever it has

- All finite products
  - So if $A$ and $B$ are objects, then there is an object $A \times B$ satisfying some universal property
  - Note "finite" here includes 0. So there is also a Terminal Object $\mathbb{1}$, the empty product.
- "Exponential Objects" $B^A$ for every object $A$ and $B$.
  - The elements of $B^A$ correspond exactly to arrows $A \to B$.
  - $B^A$ also comes equipped with an "evaluation map" $\epsilon : B^A \times A \to B$ which does the obvious thing

- Remarkably, the notion of "programming language" we've built up so far has led us to the central notion of Cartesian Closedness!
- Notice also that we can interpret our programs in *any* Cartesian Closed Category $\mathcal{C}$:
  - First pick objects $B$ and $N$ to represent $\mathbb{B}$ and $\mathbb{N}$
  - We then use cartesian closedness to interpret
    - product types by the product in $\mathcal{C}$
    - function types by the exponential in $\mathcal{C}$
  - Then pick elements $\mathbb{1} \to B$ and $\mathbb{1} \to N$ for `T`, `F`, and every $n$.
  - Lastly, pick arrows to interpret $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $\texttt{if-then-else}_\tau : \mathbb{B} \times \tau \times \tau \to \tau$ (for every type $\tau$)
- Then doing this, all of our programs correspond to arrows in this category, and the programs with no free variables are all given by elements. So running the program amounts to figuring out which element it names!

- We can run this argument in reverse too!
- Take your favorite Cartesian Closed Category $\mathcal{C}$
- We can build a programming language out of $\mathcal{C}$ as follows:
  - Add a type $\tau_A$ for every object $A$ in $\mathcal{C}$
  - Add a value $x : A$ for every element $\mathbb{1} \xrightarrow[x]{A}$
  - Add an expression $e_f$ for every arrow $f : A \to B$
- Then studying interpretations of this programming language in *other* Cartesian Closed Categories helps you understand the interactions between that category and the one you started with
- This programming langauge is called the Internal Logic of $\mathcal{C}$

Ok. That was the big jump in abstraction.

If you stuck with me through that, the rest of the talk is smooth sailing

Even if you didn't, the rest of the talk is easier
(so now is a good time to tune back in)

Before we go on: Are there any questions?

# A Natural Question

- When we defined our semantics, we allowed ourselves to pick any old arrow $\texttt{if} - \texttt{then} - \texttt{else}_\tau : \mathbb{B} \times \tau \times \tau \to \tau$
- But obviously we *want* these arrows to have something to do with each other.
- After all, they have the same *syntax*. That should be reflected somehow in their *semantics*.
- Intuitively, we should be able to give *one* interpretation that works uniformly for all types $\tau$.
- Such an interpretation is called Polymorphic, and it's an extremely useful abstraction when programming
- Following Eilenberg and Mac Lane, before we can rigoriously interpret polymorphism, we have to talk about functors

## Definition

A functor (really an endofunctor) on $\mathcal{C}$ is a map $F$ so that

- for each object $A$ of $\mathcal{C}$, $FA$ is an object of $\mathcal{C}$ too
- each arrow $A \xrightarrow{f} B$ in $\mathcal{C}$ lifts to an arrow $FA \xrightarrow{Ff} FB$ in $\mathcal{C}$ in a way compatible with composition

- The identity functor $I$
  - $IA = A$ for each object $A$
  - $If = f$ for each arrow $f : A \to B$
- The constant functor $C_X$
  - $C_X A = X$ for each object $A$
  - $C_X f = 1_X$ for each arrow $f : A \to B$
- The functor $\texttt{ITE}$
  - $\texttt{ITE}\ A = X \times A \times A$
  - $\texttt{ITE}\ f$ is the map $(x, a_1, a_2) \mapsto (x, fa_1, fa_2)$ from $\texttt{ITE}\ A \to \texttt{ITE}\ B$

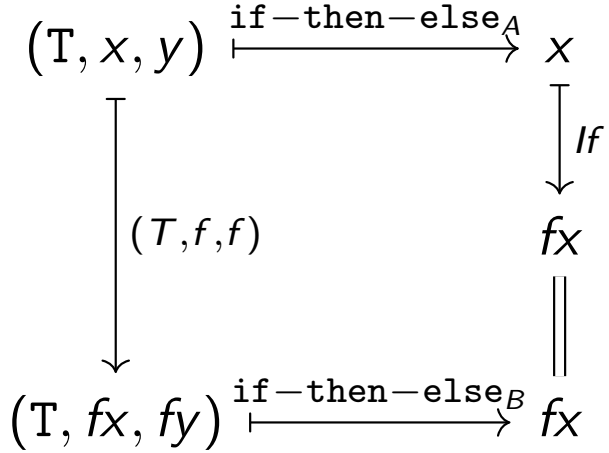
- Next we come to a Natural Transformation between functors
- This makes precise the idea that a construction is *uniform* in all of its possible inputs
- This certainly *sounds* related to our question about the semantics of `if − then − else...`

## Definition

Given two functors $F$ and $G$, a *Natural Transformation* is a collection of arrows $\theta_A : FA \to GA$ for each object $A$ such that the following square always commutes:

$$\begin{array}{ccc} FA & \xrightarrow{\theta_A} & GA \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ FB & \xrightarrow{\theta_B} & GB \end{array}$$

Let's see what happens for the intended interpretation of $\texttt{if} - \texttt{then} - \texttt{else}$:




- The square commutes! (and it does for $\mathtt{F}$ too)
- So, in the intended interpretation, $\texttt{if} - \texttt{then} - \texttt{else}_A$ is a *natural transformation* from $\mathbb{B} \times \tau \times \tau$ to $I\tau$!
- This naturality makes rigorous the idea that we really have one definition which works simultaneously for every type $\tau$
- Conversely, when thinking about natural transformations, my first source of intuition is a kind of polymorphic map.

There's one last example I want to share, but to do it, we need to expand our Toy Language one last time!

## Toy Language: Mk. IV

$$\tau ::= \mathbb{B} \mid \mathbb{N} \mid \texttt{Str} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \texttt{List}(\tau)$$

$$e ::= \ldots \mid s \mid [] : \tau \mid e :: e \mid \texttt{fold}(e_1, e_2, e_3)$$

- We've added a type `Str` of *strings*
  - "`hello`" : `Str`

- We've also added a type $\texttt{List}(\tau)$ which stores a list of elements of type $\tau$
  - [] is the empty list
  - We can add a value `x` to a preexisting list `xs` to get the new list `x :: xs`
  - So the list $[1, 2, 3]$ would be represented by $1 :: 2 :: 3 :: [] : \texttt{List}(\mathbb{N})$
- Finally, given
  - a function $\oplus : \tau \times \tau \to \tau$
  - an initial value $x^* : \tau$
  - and a list $xs = [x_1, \ldots, x_n]$
  - $\texttt{fold}(\oplus, x^*, xs) = x_1 \oplus (x_2 \oplus (\ldots (x_{n-1} \oplus (x_n \oplus x^*))))$

- For instance, we can append two lists together by setting
  - $[]\texttt{@ys} = \texttt{ys}$
  - $(\texttt{x} :: \texttt{xs})\texttt{@ys} = \texttt{x} :: (\texttt{xs@ys})$
- We can implement this idea with the following code:

$$\texttt{xs@ys} = \lambda((\texttt{xs}, \texttt{ys}) : \texttt{List}(\tau) \times \texttt{List}(\tau)).\texttt{fold}(::, \texttt{ys}, \texttt{xs})$$

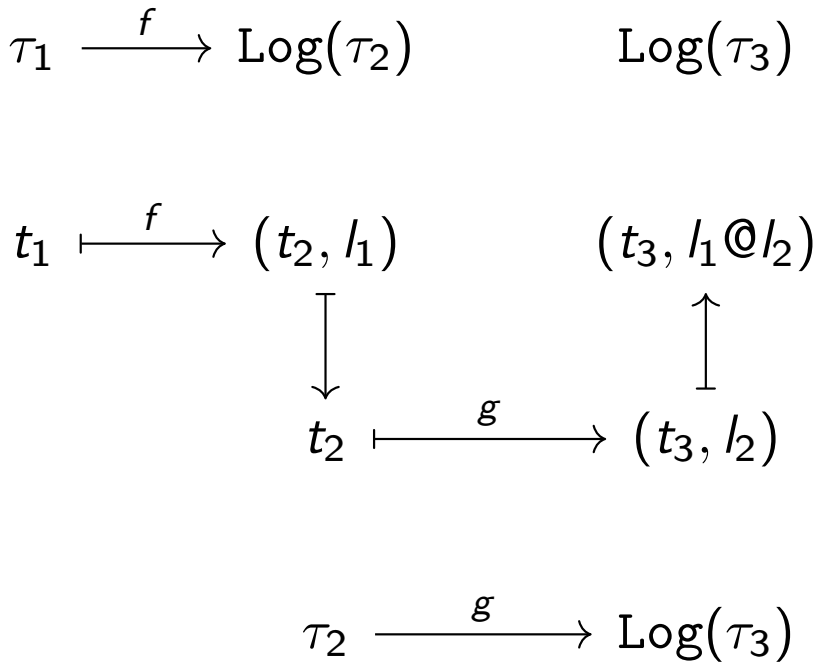
- As a (fun?) exercise, you should verify this!
- While you're at it, can you see why $\tau \mapsto \texttt{List}(\tau)$ is a functor?
- And why $[]_\tau : \mathbb{1} \to \texttt{List}(\tau)$ is actually a natural transformation from $C_{\mathbb{1}}(\tau)$ to $\texttt{List}(\tau)$?

# A Practical Hypothetical

- Now, let's say you want to add logs to your program to make debugging easier.
- You have a bunch of functions of type $\tau_1 \to \tau_2$, and you want to make them keep track of the logfile at the same time.
- You realize a good logging format might be $\texttt{Log}(\tau) = \tau \times \texttt{List}(\texttt{Str})$
- In this way, you keep track of the same output that you used to return, but you also have a list of strings representing the logs that you've saved so far.
- You might write some code like the following to use this idea:

$$\texttt{doubleWithLog}(x) = (x + x, \text{"}\texttt{doubled a number}\text{"}) : \mathbb{N} \to \texttt{Log}(\mathbb{N})$$

- There's one major issue, though
- We've broken compositionality!
- If $f : \tau_1 \to \text{Log}(\tau_2)$ and $g : \tau_2 \to \text{Log}(\tau_3)$, then we can't compose them since $\text{Log}(\tau_2) \neq \tau_2$.
- That's ok, though. We can write our own code to handle composition for us! This new *special* composition will keep track of the logfiles so that we don't have to think about it.
- How should we do this? Well, intuitively...




We can write this formally as

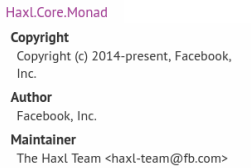
$$(f >=> g) = \lambda(t_1 : \tau_1).(\pi_L(g(\pi_L(f(t_1)))), \pi_R(f(t)) @ \pi_R(g(\pi_L(f(t)))))$$

- The experienced category theorists are probably hopping up and down right about now
- We have a functor $\tau \mapsto \mathrm{Log}(\tau)$
- Equipped with a *fancy* composition on arrows of the form $\tau \to \mathrm{Log}(\tau)$
- This is exactly the data of Kleisli Composition!
- This tells us that our (endo)functor Log is particularly special. It's actually a Monad!
- Again, this is one of my first sources of intuition for monads –
- They tend to act like datastructures where you spruce up an object with some $\sim$ bonus structure $\sim$.

# Real People Care

- A lot of software engineers are starting to get interested in Category Theory
- It's not a niche thing either, if you google "monad tutorial" you'll see just how many people are trying to understand this material
- If nothing else, hopefully this talk gave you another good response to "what's the point in all that abstract nonsense"
- I also think this lens is a good one for newcomers to the field (though I might be biased, since this was my first lens)
- I'll also add that software engineers are interested in a lot of this machinery – I only scratched the surface.





Thank you!
Questions?