Fork me on GitHub

Lambda Calculus

a crash course

Part 0

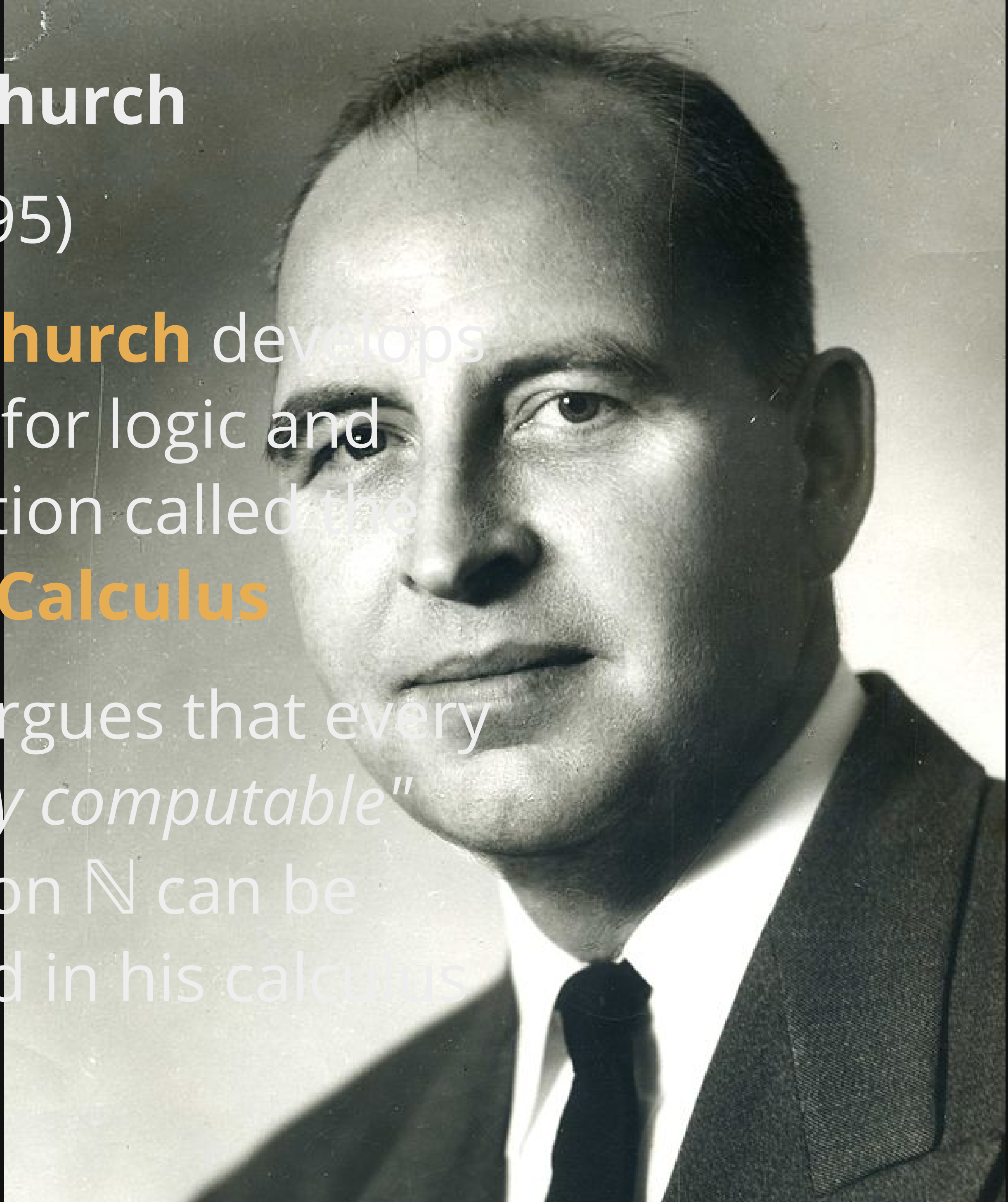
brief historical overview

Alonzo Church

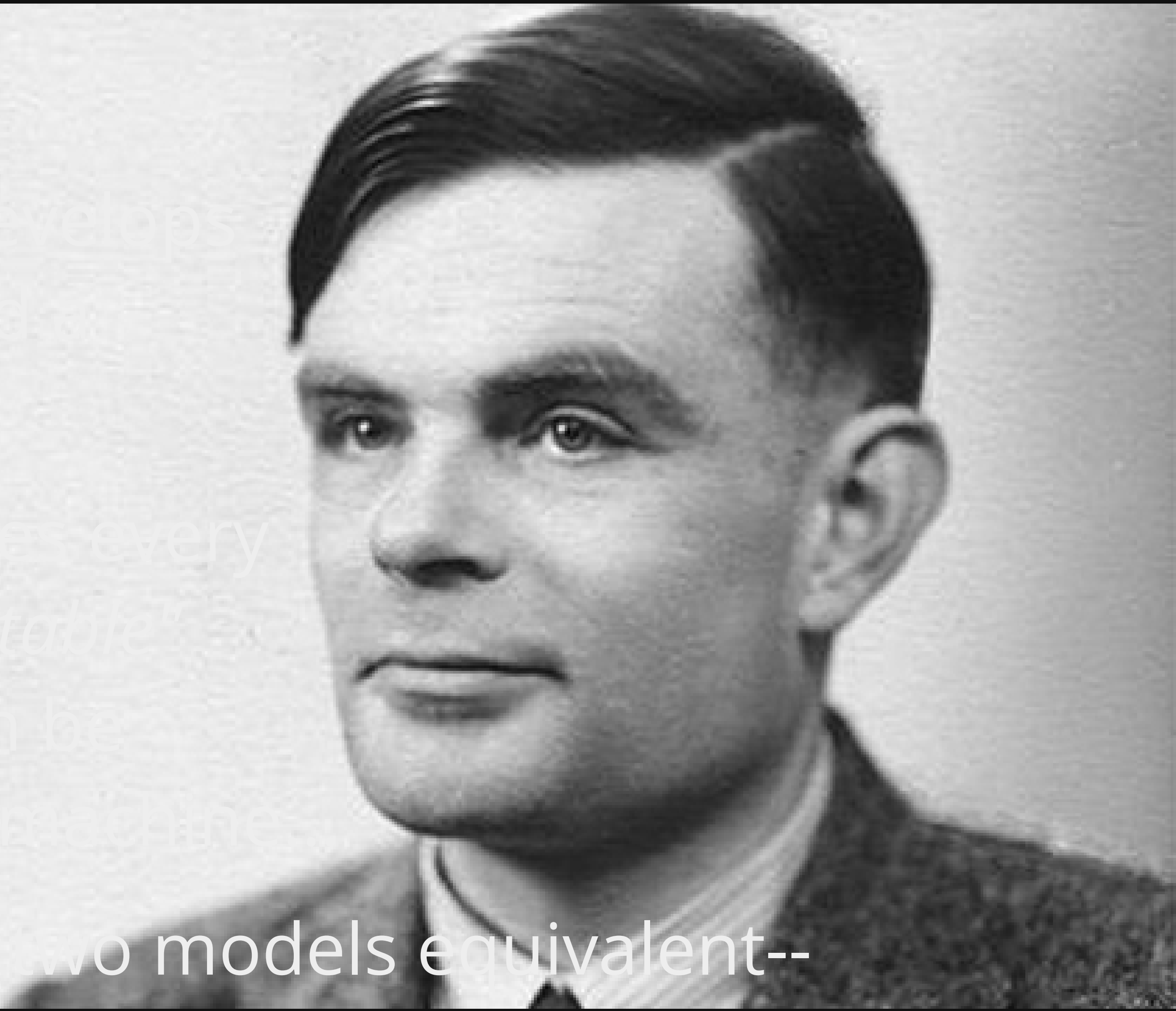
(1903-1995)

(~1930) **Church** develops a system for logic and computation called the **Lambda Calculus**

(~1935) argues that every *"effectively computable"* function on $\mathbb{N}$ can be computed in his calculus



Alan Turing



(1912-1954)

(1935) **Turing** develops what is now called

Turing Machine

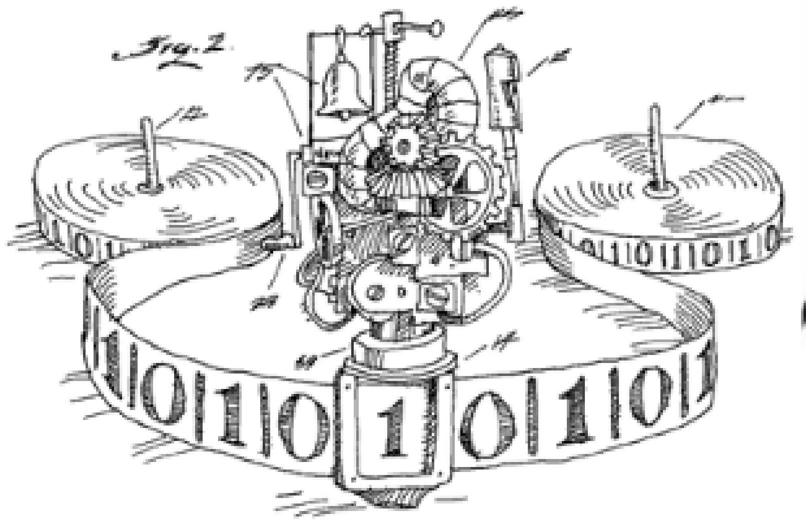
(~1936) Argues every *effectively computable* function on ℕ can be computed by his machine

...and proved the two models equivalent--
compelling evidence for **Church-Turing Thesis**

Church-Turing Thesis

$$(\lambda x.e_1)e_2 \Rightarrow_\beta e_1[e_2/x]$$

$$=$$



Algorithms vs. Languages

- The *Church-Turing Thesis* is one of the most important ideas in computer science.
- The impact of the models of Church and Turing goes well beyond the thesis itself.
- But the two models have impacted two disparate communities.
- Turing Machine ⇝ Algorithms and Complexity
- Lambda Calculus ⇝ Programming Languages

Efficiency vs. Structure

The impact and separation is not accidental.

Two sources of beauty in programs:

efficiency

structure

Turing Machine ⇜⇝ Algorithms

efficiency

Turing Machines are good at measuring resources

- **complexity theory**
  P vs. NP, polynomial hierarchy, P-space
- **asymptotic analysis** of algorithms
- **cryptography** based on how hard it is to solve certain problems
- **learning theory** based on learning power of Turing machines

λ-Calculus $\leftrightsquigarrow$ Languages

structure

λ-Calculus is good at composition and abstraction

- **lambda abstractions, higher-order-functions** (recently even in C++ and Java!)
- **denotational semantics** and **type theory** the "theory of abstraction"
- **proof assistants** (e.g. Agda, Coq, NuPRL, Isabelle)
- **languages** (e.g., Lisp, SML, Haskell, Scala...
  ...and now Java, Python, JavaScript)

The λ-Calculus

Why study it?

1. It encodes every "feasible" computation
2. It encodes logic (we'll see how later)
3. It is the foundation for functional programming (because it directly supports functional abstraction, application, and composition, it captures the essential features of most languages $\rightsquigarrow$ a more natural model of universal computation than Turing's)

The λ-Calculus

syntax 1

A λ-calculus term is either

- a **variable** $x \in \mathsf{Var}$, where Var is a countably infinite set of variable symbols, or
- an **application**, a function M applied to an argument N, usually written MN or $M(N)$, or
- a **lambda abstraction**, an expression $\lambda x.\, e$ that represents a function with input x and body e.

The λ-Calculus

syntax 2

Where a mathematician writes $x \mapsto x^2$

or an SML programmer writes `fn x => x*x`

in the λ-calculus we write $\lambda x.\, x^2$

The λ-Calculus

syntax 3

Grammar

$$e = x \mid \lambda x.\, e \mid e(e)$$

Computation

repeat single rule called β-reduction:

$$\lambda x.\, [\ldots x \ldots x \ldots](e_2) \Rightarrow [\ldots e_2 \ldots e_2 \ldots]$$

Finished

when no terms of the form $e(e)$ remain

The λ-Calculus

examples

1. $(\lambda x.\,(2 \times x + 1))7$
2. $((\lambda f.\,\lambda x.\,(f(f x)))\lambda x.\,(x + 3))2$
3. $\lambda x.\,x(x)(\lambda x.\,x(x))$

The λ-Calculus

encoding logic

Represent **TRUE** by the first projection:

$$\text{true} = \lambda x.\, \lambda y.\, x$$

Represent **FALSE** by the second projection:

$$\text{false} = \lambda x.\, \lambda y.\, y$$

Then **NOT** is defined by

$$\neg = \lambda b.\, b \text{ false true}$$

We won't prove this here, but let's check

$\neg\mathsf{true} = \mathsf{false}$

$$\neg\mathsf{true} = (\lambda b.\, b\ \mathsf{false}\ \mathsf{true})\ \mathsf{true}$$

β-reduction

$$[\mathsf{true}/b](b\ \mathsf{false}\ \mathsf{true}) = \mathsf{true}\ \mathsf{false}\ \mathsf{true}$$

η-expansion

$$(\lambda x.\, \lambda y.\, x)\mathsf{false}\ \mathsf{true} = \mathsf{false}$$

Turing was way ahead of his time

Church was way *way* ahead of his time

- **Virtue:** λ-calculus does not define a reduction order, so it is inherently parallel!
- **Problem:** no obvious cost model since number of steps depends on reduction order and some orders are very inefficient

Proposal of Acar, Bleloch, Harper, Reppy

1. Fix an order that is parallel and cheap.
2. Base a cost model on it.
3. Bound cost when mapped to standard models.

Once we have an order, we can:

- count number of reductions (work)
- count number of parallel steps (depth or span)

Their Conclusion

- **Next 50 years:** need to integrate Complexity/Algorithms and Programming Language Theory.
- **Cost models:** should be based on languages, not machines. Particularly important for parallelism.
- **Other opportunities:** Verification, type-theory and complexity, probabilistic programming, programs-as-data, cryptography and PL, game-theory and PL.

End of Part 1

Links to further reading on

Parallelism and Cost Semantics

https://www.cs.cmu.edu/~rwh/papers.html

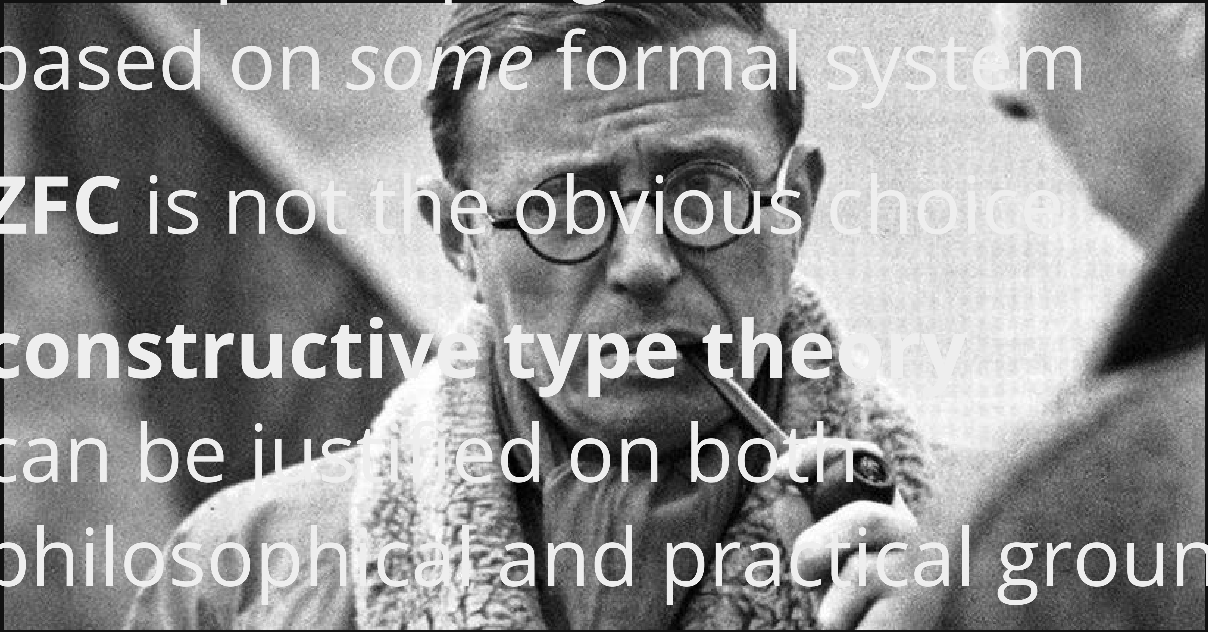
(time for a break)

Part 1

Homily on Constructive Math

- What is a good language for writing proofs?
- What kind of math do we want to do?
- In principle all math can be formalized in ZFC.
- Usually a much weaker theory is sufficient (PA suffices for much of Number Theory) (Analysis can be formalized in PA2)
- In fact, we don't need to commit, as long as our proofs use standard techniques that we believe are formalizable in *some* system.

- But to do math on a computer
  we must make a choice!
- A computer program must be based on *some* formal system
- **ZFC** is not the obvious choice
- **constructive type theory**
  can be justified on both philosophical and practical grounds



Question: Why do math on a computer?

- Because computers can check whether proofs are correct? No, the peer review process works.
- Because computers can prove many things humans can't? No, at least not anytime soon.
- Because computers are really good at computing? Yes!!

Nobody would question the utility of computer programs on the grounds that we can write those programs on a piece of paper faster and more easily in pseudo-code. This would be silly, since *programs written on paper cannot be executed*

The objection that formalizing math on computer is pointless because we can more easily write it down on a piece of paper can be disputed on similar grounds. But...

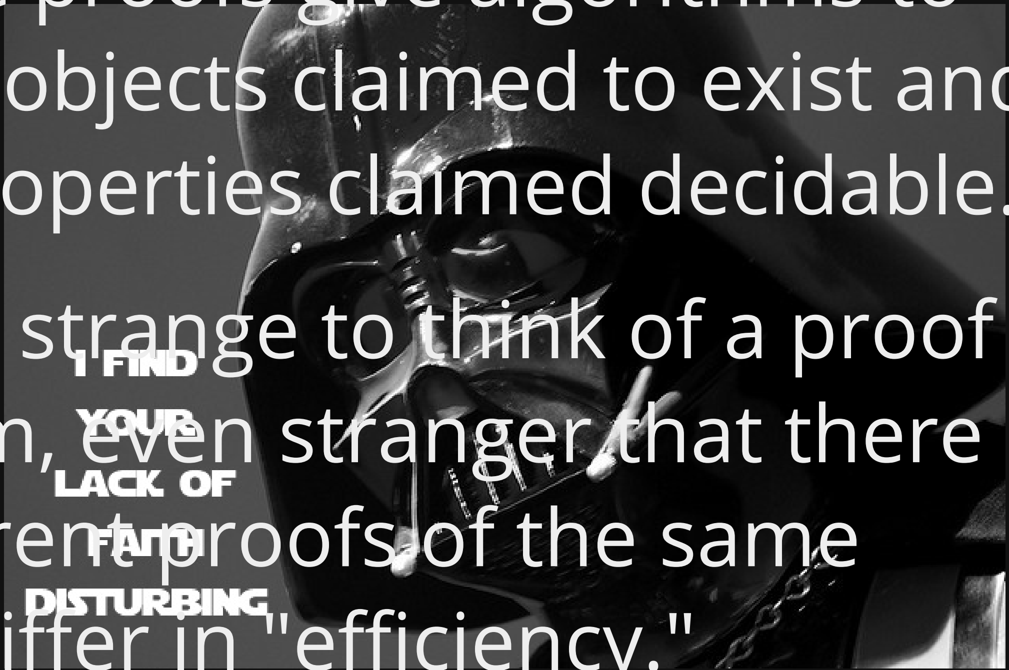
proofs of math theorems cannot be executed

...or can they?

Classical proofs cannot always be executed, but *constructive* proofs can, in a sense.

Constructive proofs give algorithms to compute all objects claimed to exist and decide all properties claimed decidable.

It may seem strange to think of a proof as a program, even stranger that there can be different proofs of the same result that differ in "efficiency."



A Change of Tack

Instead of discussing ways to formalize math, let's consider ways to extend programming languages, e.g. richer data types, new paradigms/techniques.

We will consider a high level functional language and see how it makes programming easier;
some classical algorithms become easy or obvious;
previously inconceivable programs are possible.

We don't mention logic and math at first.

Curry-Howard Correspondence

Eventually, we see *programs as proofs* of theorems and **constructive math** as a subsystem of the programming language.

The most important advantage:
programs are guaranteed correct
by virtue of the their inherent logical content!

Part II

Lambda Calculus

Intro and Quick Review

- λ-calculus is a small language based on some common mathematical idioms.
- It was invented by **Alonzo Church** in 1936, but his version was *untyped*, making the connection with mathematics rather problematic.
- In this course we'll be looking at a *typed* version.

The Impact of λ-Calculus

λ-calculus...

- the basis of functional programming languages (e.g. Haskell, SML, OCaml, Lisp, Erlang, Scala, etc)
- used to give semantics for programming languages; (1965) Peter Landin describes semantics of Algol-60 using λ-calculus.
- closely corresponds to a special logic, called intuitionistic logic, via the *Curry-Howard isomorphism*.

Notation for Sets

- **natural numbers** $\mathbb{N} = \{0, 1, 2, \ldots\}$
- **integers** $\mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$
- **booleans** $\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$
- **cartesian product**
$$R \times S = \{(x, y) \mid x \in R, y \in S\}$$
- **disjoint union**
$$R + S = \{\mathsf{inl}\, x \mid x \in R\} \cup \{\mathsf{inr}\, y \mid y \in S\}$$

Here inl and inr are "tags"; if you prefer, let $\mathsf{inl}\, x = \langle 0, x\rangle$ and $\mathsf{inr}\, y = \langle 1, y\rangle$

More Notation

- **function space** $R \to S$ = functions from R to S (often denoted S^R)
- **unit** 1 = the set containing the empty tuple $\langle\ \rangle$.
- **empty set** = 0.

These operations on sets correspond to familiar operations on natural numbers.

Some ways to describe integers

- **Arithmetic**. Here's an integer

$$3 + (7 \times 2)$$

- **Conditionals.** Here's another

$$\text{case } (7 > 5) \text{ of } \{\text{true. } 20 + 3, \text{false. } 53\}$$

(an "if ... then ... else" construction)

- **Local definitions.** Another integer:

$$\text{let } y \text{ be } (2 \times 18) + (3 \times 102) \text{ in } (y + 17 \times y)$$

This is shorthand for $y + 17 \times y$, with y set to $(2 \times 18) + (3 \times 102)$.

Exercise 1.

What integer is...

1. $(2+5) \times 8$

2.

case (case $1 > 8$ of {true. $5 > 2 + 4$, false. $3 > 2$
of {true. 3×7, false. 100}

3. let y be (let x be $3+2$. $x \times (x+3)$). $y + 15$

4.

let x be $(5+7)$. case $x > 3$ of {true. 12, false. $3 +$

Part 2

Cartesian Product

Disjoint Union

Function Space

Projections

- If $p = (x, y)$ is an ordered pair
  - $\pi_\ell p = x$ is the **first component** of p
  - $\pi_r p = y$ is the **second component** of p
- For example, here's another int
  let x be $(3, 7 + 2)$. $(\pi_\ell x) \times (\pi_r x) + (\pi_r x)$
- It's sometimes convenient to let
  $\mathsf{fst}\ (x, y) = x$ and $\mathsf{snd}\ (x, y) = y$
  denote the 1st and 2nd components of (x, y).

Pattern-matching

We can pattern-match an ordered pair.

$$\text{let } x \text{ be } (3, 7+2).\ \text{case } x \text{ of } (y, z).\ y \times z + z$$

Pattern-match is often a more convenient notation than projection.

Exercise 2

Identify the following integers

1. let y be $(7,$ let x be $3.\ x + 7).\ \pi_\ell y +$ (case y of $(u,$
2. case $(\pi_\ell(7, 357 \times 128) > 2)$ of $\{$true. $13,$ false. 2
3. let x be $(5, (2,$ true$))$. fst x + fst (case x of (y, z).

Disjoint Union

Recall that $R + S$ is the set of ordered pairs $\mathsf{inl} x$, $x \in \mathbb{R}$, and ordered pairs $\mathsf{inr} y$, $y \in S$.

We can pattern-match an element of $R + S$, e.g. $\mathsf{let}\ x\ \mathsf{be}\ \mathsf{inl}\ 3.\ \mathsf{let}\ y\ \mathsf{be}\ 7.\ \mathsf{case}\ x\ \mathsf{of} \{ \mathsf{inl} z.\ z + y, \mathsf{inr}\ w.$

Since x is defined here to be $\mathsf{inl}\ 3$, it matches the pattern $\mathsf{inl}\ z$, so in the body z is 3.

Exercise 3

Identify the following integers

1. case (case $(3 < 7)$ of $\{$true. inr $(8 + 1)$, false. inl
   of $\{$inl $u.\, u + 8$, inr $v.\, v + 3\}$

2. let z be $(3$, inr$(7$, true$))$. fst z + case snd z
   of $\{$inl $y.\, y + 2$, inr y. let 4be $x.\, ((x +$ fst $y) +$ fst

λ-abstraction

Recall S^R denotes the set of functions from $R to S$.

- **λ-abstraction**
  "$\lambda x_R.$" means "the function that takes $x \in R$ to "
- **Example** $\lambda x_{\mathbb{Z}}.(2 \times x + 1)$
  is the function taking $x : \mathbb{Z}$ to $2 \times x + 1$

application

Let $f : R \to S$ be a function and $x \in R$, then
$f x$ means f applied to x

Example

$$(\lambda x_{\mathbb{Z}}.\,(2 \times x + 1))7$$

And that completes our notation!

Exercise 4.

Identify the following integers

1. $[(\lambda f : \mathbb{Z} \to \mathbb{Z} \,.\, \lambda x : \mathbb{Z} \,.\, (f(f x)))\ \lambda x : \mathbb{Z}.\ (x + 3)]$
2. let f be $\lambda x : (\mathbb{Z} + \mathbb{B})$. case x of $\{$inl y . $y + 3$, inr y . $7\}$. (f inl 5) + (f inr false)
3. let f be $\lambda x : (\mathbb{Z} \times \mathbb{Z})$. case x of (y, z) . $(2 \times y +$ f(let u be 4. $u + 1, 8)$

Part 3

A Calculus For Integers and Booleans

A Calculus of Integers

We want to turn the above notations into a calculus. Typically, calculi are defined inductively.

As an example, here is a little calculus of *integer expressions*:

- $\underline{n}$ is an *int expr* for every $n \in \mathbb{Z}$.
- If M is an *int expr*, and N is an *int expr*, then $M + N$ is an *int expr*.
- If M is an *int expr*, and N is an *int expr*, then $M \times N$ is an *int expr*.

Thus an *int expr* is a finite string of symbols.

Don't confused *int expr* $\underline{3} + \underline{4}$ with *integer* $3 + 4$ (which is 7)

Actually, I lied: an *int expr* isn't really a finite string of symbols, it's a finite *tree* of symbols.

So $(\underline{3} + \underline{4}) \times \underline{2}$ and $\underline{3} + \underline{4} \times \underline{2}$ represent different expressions.

But $\underline{3} + \underline{4} \times \underline{2}$ and $\underline{3} + (\underline{4} \times \underline{2})$ are the same expression (i.e. same tree)

This inductive definition describes a category, where

- an object is an *algebra* consisting of a set X equipped with an element $\underline{n} \in X$, for each $n \in \mathbb{Z}$, and two binary operations $+$ and $\times$.
- A morphism is an *algebra homomorphism* i.e. a function that commutes with the operations.

The set of int expressions (trees of symbols) is an *initial algebra*, i.e. an initial object in this category.

Let us write $\vdash M : \mathsf{int}$ to mean "M is an int expr"

Then the above inductive definition can be written

$$\frac{}{\vdash \underline{n} : \mathtt{int}}\; n \in \mathbb{Z}$$

$$\frac{\vdash M : \mathtt{int} \qquad \vdash N : \mathtt{int}}{\vdash M + N : \mathtt{int}} \qquad\qquad \frac{\vdash M : \mathtt{int} \qquad \vdash N : \mathtt{int}}{\vdash M \times N : \mathtt{int}}$$

The two expressions above can be written as "proof trees," this time with the root at the bottom (like in botany).

$$\dfrac{\dfrac{\dfrac{}{\vdash 3 : \mathtt{int}} \qquad \dfrac{}{\vdash 4 : \mathtt{int}}}{\vdash 3 + 4 : \mathtt{int}} \qquad \dfrac{}{\vdash 2 : \mathtt{int}}}{\vdash (3 + 4) \times 2 : \mathtt{int}}$$

and

$$\dfrac{\dfrac{}{\vdash 3 : \mathtt{int}} \qquad \dfrac{\dfrac{}{\vdash 4 : \mathtt{int}} \qquad \dfrac{}{\vdash 2 : \mathtt{int}}}{\vdash 4 \times 2 : \mathtt{int}}}{\vdash 3 + 4 \times 2 : \mathtt{int}}$$

Calculus of Integers and Booleans

Next we want to make a calculus of integers and booleans. We define the set of types to be $\{\text{int}, \text{bool}\}$

We write $\vdash M : A$ to mean "M has type A"

To the above rules we add:

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

$$\frac{\vdash M : \texttt{int} \quad \vdash N : \texttt{int}}{\vdash M > N : \texttt{bool}} \qquad \frac{\vdash M : \texttt{bool} \quad \vdash N : B \quad \vdash N' : B}{\vdash \texttt{case}\ M\ \texttt{of}\ \{\texttt{true}.\ N, \texttt{false}.N'\} : B}$$

Local Definitions

We next want to add local definitions to our calculus, but this presents a problem. On the one hand $\mathsf{let}\ x\ \mathsf{be}\ 3\ .\ \ x+4$ should definitely be an int expr. If we type it into the computer, we get `Answer: 7`

So we want $\mathsf{let}\ x\ \mathsf{be}\ 3\ .\ \ x+4\ :\ \mathsf{int}$

But $x+4$ is not a valid int expr. Typing it in yields

Error: you haven't defined x

So we don't want $\vdash\ x + 4\ :\ \mathsf{int}$

How then can we define the calculus? We have a valid expression with a subterm that is not syntactically valid! The solution is to write

$$x\ :\ \mathsf{int} \vdash\ x + 4\ :\ \mathsf{int}$$

This means: "once we know x is an \mathsf{int}, then $x + 4$ is an int expr"

Exercise 5.

Which of the following would you expect to be correct statements?

1. $x : \text{int} \vdash x + y : \text{int}$
2. $x : \text{int} \vdash \text{let } y \text{ be } 3.\, x + y : \text{int}$
3. $x : \text{int}, y : \text{int} \vdash x + y : \text{int}$
4. $x : \text{int}, y : \text{int} \vdash x + 3 : \text{int}$