We are Language Designers!

What have we done?

▶ Define a programming language
  ▶ we were fairly formal
  ▶ still pretty close to OCaml if you squint real hard

▶ Define a type system
  ▶ outlaw bad programs that “get stuck”
  ▶ sound: no typable programs get stuck
  ▶ incomplete: knocked out some OK programs too, ohwell

Elsewhere in the Universe (or the other side of campus)

What do logicians do?

▶ Define formal logics
  ▶ tools to precisely state propositions
Elsewhere in the Universe (or the other side of campus)

What do logicians do?
- Define formal logics
  - tools to precisely state propositions
- Define proof systems
  - tools to figure out which propositions are true

Turns out, we did that too!

Punchline
We are accidental logicians!

The Curry-Howard Isomorphism
- Proofs : Propositions :: Programs : Types
  - proofs are to propositions as programs are to types
Woah. Back up a second. Logic?!

Let’s trim down our (explicitly typed) simply-typed λ-calculus to:

\[
\begin{align*}
  e & ::= x \mid \lambda x. e \mid e \cdot e \\
     & \quad \mid (e, e) \mid e.1 \mid e.2 \\
     & \quad \mid A(e) \mid B(e) \mid \text{match } e \text{ with } A \cdot e \mid B \cdot e
\end{align*}
\]

\[
\begin{align*}
  \tau & ::= b \mid \tau \to \tau \mid \tau \cdot \tau \mid \tau + \tau
\end{align*}
\]

▶ Lambdas, Pairs, and Sums
▶ Any number of base types \( b_1, b_2, \ldots \)
▶ No constants (can add one or more if you want)
▶ No fix

What good is this?!

Well, even sans constants, plenty of terms type-check with \( \Gamma = \cdot \)
\( \lambda x : b. \ x \) has type \( b \rightarrow b \)

\( \lambda x : b_1. \lambda f : b_1 \rightarrow b_2. \ f \ x \) has type \( b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2 \)

\( \lambda x : b_1. \lambda y : b_2. \lambda z : b_1. \ x \ z \ y \) has type \( (b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3 \)
\[ \lambda x : b_1 \rightarrow b_2 \rightarrow b_3. \lambda y : b_2. \lambda z : b_1. x \ z \ y \]

has type

\[ (b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3 \]

---

\[ \lambda x : b_1. \ (A(x), A(x)) \]

has type

\[ b_1 \rightarrow ((b_1 + b_7) \ast (b_1 + b_4)) \]

---

\[ \lambda f : b_1 \rightarrow b_3. \lambda g : b_2 \rightarrow b_3. \lambda z : b_1 + b_2. \]

\( (\text{match } z \text{ with } A x . f \ x \mid B x . g \ x) \)

has type

\[ b_1 \rightarrow (b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3 \]
Let $\lambda f : b_1 \to b_3. \lambda g : b_2 \to b_3. \lambda z : b_1 + b_2. (\text{match } z \text{ with } A . f \, x | B . g \, x)$

has type

$$(b_1 \to b_3) \to (b_2 \to b_3) \to (b_1 + b_2) \to b_3$$

Let $\lambda x : b_1 \ast b_2. \lambda y : b_3. ((y, x.1), x.2)$

has type

$$(b_1 \ast b_2) \to b_3 \to ((b_3 \ast b_1) \ast b_2)$$

Empty and Nonempty Types

Just saw a few "nonempty" types

- $\tau$ nonempy if closed term $e$ has type $\tau$
- $\tau$ empty otherwise
Empty and Nonempty Types

Just saw a few “nonempty” types

▶ \( \tau \) nonempty if closed term \( e \) has type \( \tau \)
▶ \( \tau \) empty otherwise

Are there any empty types?

Sure! \( b_1 \) \( b_1 \rightarrow b_2 \) \( b_1 \rightarrow (b_2 \rightarrow b_1) \rightarrow b_2 \)

What does this one mean?

\( b_1 + (b_1 \rightarrow b_2) \)

I wonder if there’s any way to distinguish empty vs. nonempty...
Empty and Nonempty Types

Just saw a few “nonempty” types

- \( \tau \) \textit{nonempy} if closed term \( e \) has type \( \tau \)
- \( \tau \) \textit{empty} otherwise

Are there any empty types?

Sure!

\[
\begin{align*}
& b_1 & b_1 \rightarrow b_2 & b_1 \rightarrow (b_2 \rightarrow b_1) \rightarrow b_2 \\
\end{align*}
\]

What does this one mean?

\[
b_1 + (b_1 \rightarrow b_2)
\]

I wonder if there’s any way to distinguish empty vs. nonempty...

Ohwell, now for a \textit{totally irrelevant} tangent!

Propositional Logic

Suppose we have some set \( b \) of basic propositions \( b_1, b_2, \ldots \)

- e.g. “ML is better than Haskell”
Propositional Logic

Suppose we have some set $b$ of basic propositions $b_1, b_2, \ldots$

- e.g. “ML is better than Haskell”

Then, using standard operators $\supset, \land, \lor$, we can define formulas:

$$p ::= b \mid p \supset p \mid p \land p \mid p \lor p$$

- e.g. “ML is better than Haskell” $\land$ “Haskell is not pure”

Some formulas are tautologies: by virtue of their structure, they are always true regardless of the truth of their constituent propositions.

- e.g. $p_1 \supset p_1$

Not too hard to build a proof system to establish tautologyhood.

Proof System

$$\Gamma ::= \cdot \mid \Gamma, p$$

Suppose we have some set $b$ of basic propositions $b_1, b_2, \ldots$

- e.g. “ML is better than Haskell”

Then, using standard operators $\supset, \land, \lor$, we can define formulas:

$$p ::= b \mid p \supset p \mid p \land p \mid p \lor p$$

- e.g. “ML is better than Haskell” $\land$ “Haskell is not pure”

Some formulas are tautologies: by virtue of their structure, they are always true regardless of the truth of their constituent propositions.

- e.g. $p_1 \supset p_1$
Proof System

\[ \Gamma ::= \cdot \mid \Gamma, p \]

\[ \Gamma \vdash p \]

\[ \frac{\Gamma \vdash p_1 \quad \Gamma \vdash p_2}{\Gamma \vdash p_1 \land p_2} \]

\[ \frac{\Gamma \vdash p_1 \land p_2}{\Gamma \vdash p_1} \]

\[ \frac{\Gamma \vdash p_1 \land p_2}{\Gamma \vdash p_2} \]

\[ \frac{\Gamma \vdash p_1 \land p_2}{\Gamma \vdash p_1 \lor p_2} \]
Proof System

\[ \Gamma ::= \cdot | \Gamma, p \]

\[ \Gamma \vdash p \]

\[ \Gamma \vdash p_1 \]

\[ \Gamma \vdash p_2 \]

\[ \Gamma \vdash p \]

\[ \Gamma \vdash p_1 \land p_2 \]

\[ \Gamma \vdash p_1 \land p_2 \]

\[ \Gamma \vdash p_1 \]

\[ \Gamma \vdash p_2 \]

\[ \Gamma \vdash p_1 \lor p_2 \]

\[ \Gamma \vdash p_1 \lor p_2 \]

\[ \Gamma \vdash p_1 \lor p_2 \]

\[ \Gamma, p_1 \vdash p_2 \]

\[ \Gamma, p_1 \vdash p_2 \]

\[ \Gamma, p_2 \vdash p_3 \]

\[ \Gamma, p_1 \vdash p_2 \]

\[ \Gamma, p_2 \vdash p_3 \]

\[ \Gamma \vdash p_3 \]

\[ p \in \Gamma \]

\[ \Gamma \vdash p \]

\[ \Gamma \vdash p_1 \lor p_2 \]

\[ \Gamma \vdash p_1 \lor p_2 \]
Wait a second... ZOMG!

That's exactly our type system! Just erase terms, change each $\tau$ to a $p$, and translate $\to$ to $\supset$, $*$ to $\land$, $+$ to $\lor$.

$$\Gamma \vdash e : \tau$$

$$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e : \tau_1 * \tau_2 \quad \Gamma \vdash e : \tau_1 * \tau_2$$

$$\Gamma \vdash A(e) : \tau_1 + \tau_2 \quad \Gamma \vdash B(e) : \tau_1 + \tau_2$$

$$\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau$$

$$\Gamma \vdash \text{match } e \text{ with } Ax. e_1 \mid By. e_2 : \tau$$

$$\Gamma(x) = \tau \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_1 e_2 : \tau_1$$
What does it all mean? The Curry-Howard Isomorphism.

- Given a well-typed closed term, take the typing derivation, erase the terms, and have a propositional-logic proof
- Given a propositional-logic proof, there exists a closed term with that type
- A term that type-checks is a *proof* — it tells you exactly how to derive the logical formula corresponding to its type

- Constructive (hold that thought) propositional logic and simply-typed lambda-calculus with pairs and sums are *the same thing*.
  - Computation and logic are *deeply* connected
  - $\lambda$ is no more or less made up than implication
- Revisit our examples under the logical interpretation...

\[
\lambda x : b. \ x \\
\text{is a proof that} \\
\quad b \rightarrow b
\]

\[
\lambda x : b_1. \lambda f : b_1 \rightarrow b_2. \ f \ x \\
\text{is a proof that} \\
\quad b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2
\]
\[ \lambda x : b_1 \rightarrow b_2 \rightarrow b_3. \lambda y : b_2. \lambda z : b_1. x \, z \, y \] is a proof that

\[ (b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3 \] is a proof that

\[ \lambda x : b_1. (A(x), A(x)) \]

\[ \lambda f : b_1 \rightarrow b_3. \lambda g : b_2 \rightarrow b_3. \lambda z : b_1 + b_2. (\text{match } z \text{ with } A x. f x | B x. g x) \]

is a proof that

\[ (b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3 \]

is a proof that

\[ \lambda x : b_1 \ast b_2. \lambda y : b_3. ((y, x.1), x.2) \]

\[ \lambda : b_1 \ast b_2. \lambda y : b_3. ((y, x.1), x.2) \]

\[ (b_1 \ast b_2) \rightarrow b_3 \rightarrow ((b_3 \ast b_1) \ast b_2) \]

\[ b_1 \rightarrow ((b_1 + b_7) \ast (b_1 + b_4)) \]
So what?

Because:
- This is just fascinating (glad I’m not a dog)
- Don’t think of logic and computing as distinct fields
- Thinking “the other way” can help you know what’s possible/impossible
- Can form the basis for theorem provers
- Type systems should not be \textit{ad hoc} piles of rules!

So, every typed $\lambda$-calculus is a proof system for some logic...

Is STLC with pairs and sums a complete proof system for propositional logic? Almost...

Classical vs. Constructive

Classical propositional logic has the “law of the excluded middle”:

$$\Gamma \vdash p_1 + (p_1 \rightarrow p_2)$$

(Think “$p + \neg p$” – also equivalent to double-negation $\neg\neg p \rightarrow p$)

STLC does not support this law; for example, no closed expression has type $b_1 + (b_1 \rightarrow b_2)$
Classical vs. Constructive

Classical propositional logic has the “law of the excluded middle”:

\[ \Gamma \vdash p_1 + (p_1 \rightarrow p_2) \]

(Think “\(p + \neg p\)” – also equivalent to double-negation \(\neg\neg p \rightarrow p\))

STLC does not support this law; for example, no closed expression has type \(b_1 + (b_1 \rightarrow b_2)\)

Logics without this rule are called constructive. They’re useful because proofs “know how the world is” and “are executable” and “produce examples”

Classical vs. Constructive, an Example

Theorem: There exist irrational numbers \(a\) and \(b\) such that \(a^b\) is rational.

Classical Proof:

Let \(x = \sqrt{2}\). Either \(x^x\) is rational or it is irrational.

If \(x^x\) is rational, let \(a = b = \sqrt{2}\), done.

If \(x^x\) is irrational, let \(a = x^x\) and \(b = x\). Since

\[ \left(\sqrt{2}^\sqrt{2}\right)^\sqrt{2} = \sqrt{2}\left(\sqrt{2} \cdot \sqrt{2}\right) = \sqrt{2}^2 = 2, \text{ done.} \]
Classical vs. Constructive, an Example

Theorem: There exist irrational numbers \(a\) and \(b\) such that \(a^b\) is rational.

Classical Proof:

Let \(x = \sqrt{2}\). Either \(x^x\) is rational or it is irrational.

If \(x^x\) is rational, let \(a = b = \sqrt{2}\), done.

If \(x^x\) is irrational, let \(a = x^x\) and \(b = x\). Since
\[
\left(\sqrt{2}^{\sqrt{2}}\right)^2 = \sqrt{2}^{(\sqrt{2} \cdot \sqrt{2})} = \sqrt{2}^2 = 2, \text{ done.}
\]

Well, I guess we know there are some \(a\) and \(b\) satisfying the theorem... but which ones? LAME.

Constructive Proof:

Let \(a = \sqrt{2}, \quad b = \log_2 9\).

Since \(\sqrt{2}^{\log_2 9} = 9^\log_2 \sqrt{2} = 9^{\log_2(2^{0.5})} = 9^{0.5} = 3, \text{ done.}\)
Constructive logic allows us to distinguish between things that classical logic lumps together.

Consider “$P$ is true.” vs. “It would be absurd if $P$ were false.”

$P$ vs. $\neg\neg P$

Those are different things, but classical logic can’t tell.

Our friends Gödel and Gentzen gave us this nice result:

$P$ is provable in classical logic iff $\neg\neg P$ is provable in constructive logic.
Fix

A “non-terminating proof” is no proof at all.

Remember the typing rule for fix:

\[ \Gamma \vdash e : \tau \rightarrow \tau \]
\[ \Gamma \vdash \text{fix } e : \tau \]

That let’s us prove anything! Example: \text{fix } \lambda x : b. x has type \( b \)

So the “logic” is \textit{inconsistent} (and therefore worthless)

Related: In ML, a value of type \('a\) never terminates normally (raises an exception, infinite loop, etc.)

let rec f x = f x
let z = f 0

Last word on Curry-Howard

It’s not just STLC and constructive propositional logic

\textit{Every} logic has a corresponding typed \( \lambda \) calculus (and no consistent logic has something as “powerful” as \textit{fix}).

If you remember one thing: the typing rule for function application is \textit{modus ponens}