Programming Language Theory

Curry-Howard Isomorphism
Curry-Howard Isomorphism

What we did:
▶ Define a programming language
▶ Define a type system to rule out programs that get stuck

What logicians do:
▶ Define a logic (a way to state propositions)
  ▶ Example: Propositional logic \( p ::= b \mid p \land p \mid p \lor p \mid p \rightarrow p \)
▶ Define a proof system (a way to prove propositions)

But it turns out we did that too!

Slogans:
▶ “Propositions are Types”
▶ “Proofs are Programs”
A slight variant

Let’s take the (explicitly-typed) Simply-Typed Lambda Calculus with

- base types $b_1, b_2, \ldots$
- no constants (but, could add one or more)
- pairs
- sums

\[
\begin{align*}
  e &::= x \mid \lambda x:\tau.\ e \mid e\ e \mid (e, e) \mid e.1 \mid e.2 \\
  &\quad \mid L(e) \mid R(e) \mid \text{case } e \text{ of } L(x) \Rightarrow e \mid R(x) \Rightarrow e

  v &::= \lambda x:\tau.\ e \mid (v, v) \mid L(v) \mid R(v)

  \tau &::= b_i \mid \tau \rightarrow \tau \mid \tau \ast \tau \mid \tau + \tau
\end{align*}
\]

Even without constants, plenty of terms type-check with $\Gamma = \cdot \ldots$
Example programs

\[ \lambda x : b_{17}. \ x \]

has type

\[ b_{17} \rightarrow b_{17} \]
Example programs

\[ \lambda x : b_1. \ \lambda f : b_1 \to b_2. \ f \ x \]

has type

\[ b_1 \to (b_1 \to b_2) \to b_2 \]
Example programs

\[ \lambda f : b_1 \to b_2 \to b_3. \ \lambda x : b_2. \ \lambda y : b_1. \ f \ y \ x \]

has type

\[ (b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3 \]
Example programs

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

has type

\[ b_1 \to ((b_1 + b_7) \times (b_1 + b_4)) \]
Example programs

\[ \lambda f : b_1 \rightarrow b_3. \ \lambda g : b_2 \rightarrow b_3. \ \lambda z : b_1 + b_2. \ (\text{case } z \text{ of } L(x) \Rightarrow f \ x \mid R(x) \Rightarrow g \ x) \]

has type

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

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

has type

\[ (b_1 \ast b_2) \rightarrow b_3 \rightarrow ((b_3 \ast b_1) \ast b_2) \]
Empty and Nonempty Types

We have seen several “nonempty” types (closed terms of that type exist):

- $b_{17} \rightarrow b_{17}$
- $b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2$
- $(b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3$
- $b_1 \rightarrow ((b_1 + b_7) \ast (b_1 + b_4))$
- $(b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3$
- $(b_1 \ast b_2) \rightarrow b_3 \rightarrow ((b_3 \ast b_1) \ast b_2)$

But there are also lots of “empty” types (no closed terms of that type exist):

- $b_1$
- $b_1 \rightarrow b_2$
- $b_1 + (b_1 \rightarrow b_2)$
- $b_1 \rightarrow (b_2 \rightarrow b_1) \rightarrow b_2$

And “I” have a “secret” way of knowing whether or not a type will be empty; let me show you propositional logic...
Propositional Logic

With \( \rightarrow \) for implies, \( + \) for (inclusive-)or, and \( * \) for and:

\[
p ::= b | p \rightarrow p | p * p | p + p
\]

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

\[
\Gamma \vdash p
\]

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

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

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

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

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

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

\[
\frac{p \in \Gamma}{\Gamma \vdash p}
\]

\[
\frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \rightarrow p_2}
\]

\[
\frac{\Gamma \vdash p_1 \rightarrow p_2 \quad \Gamma \vdash p_1}{\Gamma \vdash p_2}
\]
But that looks familiar. . .

That’s exactly our type system, obtained by erasing terms and changing every $\tau$ to a $p$:

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ast \tau_2}{\Gamma \vdash e.1 : \tau_1} \quad \frac{\Gamma \vdash e_2 : \tau_1 \ast \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \mathcal{L}(e) : \tau_1 + \tau_2}{\Gamma \vdash \mathcal{L}(e) : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathcal{R}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 + \tau_2 \quad \Gamma, x:\tau_1 \vdash e_1 : \tau \quad \Gamma, y:\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case } e \text{ of } \mathcal{L}(x) => e_1 \mid \mathcal{R}(y) => e_2 : \tau}$$

$$\Gamma(x) = \tau \quad \frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x:\tau_1. \ e : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ e_2 : \tau_1}$$
Curry-Howard Isomorphism

- Given a well-typed closed term, we can take the typing derivation, erase the terms, and have a propositional-logic proof.
- Given a propositional-logic proof, there exists a well-typed closed term with that type.
- A term that type-checks is a proof — it tells you exactly how to derive the logic formula corresponding to its type.
- Intuitionistic (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 examples under the logical interpretation...
Example proofs

\[ \lambda x : b_{17}. \ x \]

is a proof that

\[ b_{17} \rightarrow b_{17} \]
Example proofs

\[
\lambda x: b_1. \lambda f: b \to b_2. f \, x
\]

is a proof that

\[
b_1 \to (b_1 \to b_2) \to b_2
\]
Example proofs

\[ \lambda f : b_1 \to b_2 \to b_3. \lambda x : b_2. \lambda y : b_1. f \ y \ x \]

is a proof that

\[ (b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3 \]
Example proofs

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

is a proof that

\[ b_1 \rightarrow ((b_1 + b_7) \cdot (b_1 + b_4)) \]
Example proofs

\[
\lambda f: b_1 \rightarrow b_3. \lambda g: b_2 \rightarrow b_3. \lambda z: b_1 + b_2. \\
(\text{case } z \text{ of } L(x) \Rightarrow f \ x \mid R(x) \Rightarrow g \ x)
\]

is a proof that

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

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

is a proof that

\[ (b_1 \ast b_2) \rightarrow b_3 \rightarrow ((b_3 \ast b_1) \ast b_2) \]
Why care?

Because:

▶ This is just fascinating!
▶ Don’t think of logic and computing as separate fields.
▶ Thinking “the other way” informs what’s possible/impossible.
▶ Can form the basis for automated theorem provers.
▶ Type systems should not be *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 or not p” — also equivalent to double-negation.)

STLC has no proof for this.

▶ there is no closed expression with this type

Logics without this rule are called constructive (a.k.a., intuitionistic). They’re useful because proofs:

▶ “know how the world is”
▶ “are executable”
▶ “produce examples”

You can still “branch on possibilities” (make the excluded middle an explicit assumption):

$$((p_1 + (p_1 \rightarrow p_2)) \ast (p_1 \rightarrow p_3) \ast ((p_1 \rightarrow p_2) \rightarrow p_3)) \rightarrow p_3$$
Classical vs. Constructive

Theorem: I can always wake up at 9AM and get to campus by 10AM.

(Classical) Proof:
If it is a weekday, I can take a bus that leaves at 9:30AM.
If it is not a weekday, traffic is light and I can drive.
Since it is or is not a weekday, I can get to campus by 10AM.

Problem:
If you wake up and don't know whether or not it is a weekday, this proof does not let you construct a plan to get to campus by 10AM.

In constructive logic, that never happens: From a proof, can always extract a program that "does" what you proved "could be".

You could not prove the theorem above, but you could prove: "If I know whether or not it is a weekday, then I can get to campus by 10AM".
Classical vs. Constructive

Theorem: I can always wake up at 9AM and get to campus by 10AM.

(Classical) Proof:
If it is a weekday, I can take a bus that leaves at 9:30AM.
If it is not a weekday, traffic is light and I can drive.
Since it is or is not a weekday, I can get to campus by 10AM.

Problem:
If you wake up and don’t know whether or not it is a weekday, this proof does not let you construct a plan to get to campus by 10AM.

In constructive logic, that never happens: From a proof, you can always extract a program that “does” what you proved “could be.” You could not prove the theorem above, but you could prove: “If I know whether or not it is a weekday, then I can get to campus by 10AM.”
Classical vs. Constructive

Theorem: I can always wake up at 9AM and get to campus by 10AM.

(Classical) Proof:
If it is a weekday, I can take a bus that leaves at 9:30AM.
If it is not a weekday, traffic is light and I can drive.
Since it is or is not a weekday, I can get to campus by 10AM.

Problem:
If you wake up and don’t know whether or not it is a weekday,
this proof does not let you construct a plan to get to campus by 10AM.
Classical vs. Constructive

Theorem: I can always wake up at 9AM and get to campus by 10AM.

(Classical) Proof:
If it is a weekday, I can take a bus that leaves at 9:30AM.
If it is not a weekday, traffic is light and I can drive.
Since it is or is not a weekday, I can get to campus by 10AM.

Problem:
If you wake up and don’t know whether or not it is a weekday,
this proof does not let you construct a plan to get to campus by 10AM.

In constructive logic, that never happens: From a proof,
can always extract a program that “does” what you proved “could be”.

You could not prove the theorem above, but you could prove:
“If I know whether or not it is a weekday, then I can get to campus by 10AM”.
Recursion?

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

Recall the typing rule for \texttt{fix}:

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

That let’s us prove anything!
For example: \texttt{fix \ \lambda x:\!b_3. \ x} has type \texttt{b_3}.

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

Related: In ML, a “value” of type `'a` never terminates normally (it must raise an exception, diverge, etc.)

\begin{verbatim}
fun f x = f x  (* f : 'a \rightarrow 'b *)
val z = f 0   (* z : 'a *)
\end{verbatim}
Last word on Curry-Howard

It's not just STLC and intuitionistic propositional logic.

*Every* logic has a corresponding typed $\lambda$ calculus

- No consistent logic has something as “powerful” as $\text{fix}$
- Example: When we add universal types (“generics”) in a future lecture, that corresponds to adding universal quantification.

If you remember one thing:

- The typing rule for function application is *modus ponens*. 