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

\[
e ::= x | \lambda x: \tau. \ e | e \ e \\
     (e, e) | e.1 | e.2 \\
L(e) | R(e) | \text{case } e \text{ of } L(x) => e | R(x) => e
\]

\[
v ::= \lambda x: \tau. \ e | (v, v) | L(v) | R(v)
\]

\[
\tau ::= b_i | \tau \rightarrow \tau | \tau \ast \tau | \tau + \tau
\]

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

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

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

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

\[\Gamma \vdash p\]

\[
\begin{array}{c}
\Gamma \vdash p_1 \\
\Gamma \vdash p_2
\end{array} \quad \frac{\Gamma \vdash p_1 \ast p_2}{\Gamma \vdash p_1} \quad \frac{\Gamma \vdash p_1 \ast p_2}{\Gamma \vdash p_2}
\]

\[
\begin{array}{c}
\Gamma \vdash p_1 \\
\Gamma \vdash p_2
\end{array} \quad \frac{\Gamma \vdash p_1 + p_2}{\Gamma \vdash p_1 + p_2} \quad \frac{\Gamma \vdash p_1 + p_2}{\Gamma \vdash p_3} \quad \frac{\Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3}{\Gamma \vdash p_3}
\]

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

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

\[
\Gamma \vdash e : \tau
\]

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\hline
\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2
\]

\[
\Gamma \vdash e : \tau_1 \times \tau_2 \\
\hline
\Gamma \vdash e.1 : \tau_1
\]

\[
\Gamma \vdash e.2 : \tau_2
\]

\[
\Gamma \vdash e : \tau_1 \\
\hline
\Gamma \vdash L(e) : \tau_1 + \tau_2
\]

\[
\Gamma \vdash e : \tau_2 \\
\hline
\Gamma \vdash R(e) : \tau_1 + \tau_2
\]

\[
\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau \\
\hline
\Gamma \vdash \text{case } e \text{ of } L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2 : \tau
\]

\[
\Gamma(x) = \tau \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \\
\hline
\Gamma \vdash x : \tau \quad \Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2
\]

\[
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\hline
\Gamma \vdash e_1 \ e_2 : \tau_1
\]
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

$$\lambda x : b_1 . \lambda f : b_1 \to b_2 . f \ x$$

is a proof that

$$b_1 \to (b_1 \to b_2) \to b_2$$
Example proofs

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

is a proof that

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

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

is a proof that

\[ b_1 \to ((b_1 + b_7) * (b_1 + b_4)) \]
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 λ-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”:

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

(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):

\[
((p_1 + (p_1 \to p_2)) * (p_1 \to p_3) * ((p_1 \to p_2) \to p_3)) \to p_3
\]
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, 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.”

Matthew Fluet
Programming Language Theory
Lecture 13
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” done.

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 \( \text{fix} \):

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

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

So the “logic” is **inconsistent** (and therefore worthless).

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

\[
\begin{align*}
\text{fun } f \ x &= f \ x \quad (* f : 'a \to 'b *) \\
\text{val } z &= f \ 0 \quad (* z : 'a *)
\end{align*}
\]
Continuations?

Recall the typing rules for \texttt{letcc} and \texttt{throw}:

\[
\begin{align*}
\Gamma, x : \text{cont } \tau_a & \vdash e_b : \tau_a & \Gamma \vdash e_k : \text{cont } \tau_a & \Gamma \vdash e_a : \tau_a \\
\Gamma & \vdash \text{letcc } x. e_b : \tau_a & \Gamma & \vdash \text{throw } e \ e : \tau
\end{align*}
\]

- \texttt{letcc}: from a \text{cont } \tau_a assumption produce a \tau_a, produce \tau_a
- \texttt{throw}: from a \text{cont } \tau_a and a \tau_a, produce (any) \tau

But, STLC w/ \texttt{letcc} and \texttt{throw} (and w/o \texttt{fix}) is terminating.
- Not (necessarily) inconsistent.
Continuations?

Recall the typing rules for \texttt{letcc} and \texttt{throw}:

\[
\begin{align*}
\Gamma, x : \text{cont} \tau_a & \vdash e_b : \tau_a \\
\Gamma & \vdash \text{letcc } x. \ e_b : \tau_a \\
\Gamma & \vdash e_k : \text{cont} \tau_a \quad \Gamma & \vdash e_a : \tau_a
\end{align*}
\]

\[
\begin{align*}
\Gamma, \neg p & \vdash p \\
\Gamma & \vdash p
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash \neg p \quad \Gamma & \vdash p \\
\Gamma & \vdash q
\end{align*}
\]

- \texttt{letcc} is a form of proof-by-contradiction
- \texttt{throw} is law of non-contradiction

CPS transformation corresponds to double-negation translation that maps classical proofs to intuitionistic proofs.
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 `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*. 