A slight variant

Let’s take the explicitly typed simply-typed lambda-calculus with:

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

\[
e ::= x \mid \lambda x. e \mid ee \mid (e, e) \mid e.1 \mid e.2 \mid A(e) \mid B(e) \mid \text{match } e \text{ with } Ax. e \mid Bx. e
\]

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

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

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

has type

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

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

has type

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

Example programs

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

has type

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

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

has type

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

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

has type

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

Empty and Nonempty Types

Have seen several “nonempty” types (closed terms of type exist):

\[
\begin{align*}
& b_{17} \rightarrow b_{17} \\
& b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2 \\
& (b_1 \rightarrow b_2) \rightarrow b_3 \rightarrow b_1 \rightarrow b_3 \\
& b_1 \rightarrow ((b_1 + b_7) * (b_1 + b_4)) \\
& (b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3 \\
& (b_1 * b_2) \rightarrow b_3 \rightarrow ((b_3 * b_1) * b_2)
\end{align*}
\]

There are also many “empty” types (no closed term of type exists):

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

And there is a “secret” way of knowing whether a type will be empty; let me show you propositional logic...

Propositional Logic

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

\[
\begin{align*}
p & ::= b \mid p \rightarrow p \mid p * p \mid p + p \\
\Gamma & ::= \cdot \mid \Gamma, p
\end{align*}
\]

\[\Gamma \vdash p\]

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

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

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

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

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

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

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

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

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

Guess what!!!!

That’s exactly our type system, erasing terms and changing each \( \tau \) to a \( p \)

\[
\begin{array}{c}
\Gamma \vdash e : \tau
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\frac{}{\Gamma \vdash e : \tau_1 \ast \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2 \\
\frac{}{\Gamma \vdash e.1 : \tau_1}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e.2 : \tau_2 \\
\frac{}{\Gamma \vdash e.2 : \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e : \tau_1 \\
\frac{}{\Gamma \vdash A(e) : \tau_1 + \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash B(e) : \tau_1 + \tau_2 \\
\frac{}{\Gamma \vdash B(e) : \tau_1 + \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \\
\frac{}{\Gamma \vdash \text{match } e \text{ with } Ax. e_1 | By. e_2 : \tau}
\end{array}
\]

\[
\begin{array}{c}
\Gamma(x) = \tau \\
\frac{}{\Gamma, x : \tau_1 \vdash e : \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \\
\frac{}{\Gamma \vdash e_2 : \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \\
\frac{}{\Gamma \vdash e_2 : \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \\
\frac{}{\Gamma \vdash e_2 : \tau_2}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \\
\frac{}{\Gamma \vdash e_2 : \tau_2}
\end{array}
\]
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 logic 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...

Example programs

- \( \lambda x : b_1. \; \lambda f : b_1 \to b_2. \; f \; x \)
  - is a proof that
  - \( b_1 \to (b_1 \to b_2) \to b_2 \)

- \( \lambda x : b_1 \to b_2 \to b_3. \; \lambda y : b_2. \; \lambda z : b_1. \; x \; z \; y \)
  - is a proof that
  - \( (b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3 \)
Example programs

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

is a proof that

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

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

is a proof that

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

Example programs

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

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

STLC does not support this law; for example, no closed expression has type \( b_7 + (b_7 \rightarrow b_5) \)

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

Can still “branch on possibilities” by making the excluded middle an explicit assumption:

\[ ((p_1 + (p_1 \rightarrow p_2)) \ast (p_1 \rightarrow p_3) \ast ((p_1 \rightarrow p_2) \rightarrow p_3)) \rightarrow p_3 \]

Example classical proof

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

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 a weekday or not a weekday, I can get to campus by 10AM.

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

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

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

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 a weekday or not a weekday, I can get to campus by 10AM.

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

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

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

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: 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 (raises an exception, infinite loop, etc.)

\[ \text{let rec } f \ x = f \ x \]
\[ \text{let } z = f \ 0 \]

Last word on Curry-Howard

It’s not just STLC and constructive propositional logic

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

Example: When we add universal types (“generics”) in a later lecture, that corresponds to adding universal quantification

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