CS-XXX: Graduate Programming Languages

Lecture 12 — The Curry-Howard Isomorphism

Dan Grossman
2012

Curry-Howard Isomorphism

What we did:

- Define a programming language
- Define a type system to rule out programs we don't want

What logicians do:

- Define a logic (a way to state propositions)
  - Example: Propositional logic $p ::= b \mid p \land p \mid p \lor p \mid p \to p$
- 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:

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

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\ e \mid e\ e \\ & \mid & (e, e) \mid e.1 \mid e.2 \\ & \mid & \mathsf{A}(e) \mid \mathsf{B}(e) \mid \textbf{match } e \textbf{ with } \mathsf{A}x.\ e \mid \mathsf{B}x.\ e \\ \tau & ::= & b \mid \tau \to \tau \mid \tau * \tau \mid \tau + \tau \end{array}$$

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

Example programs

$$\lambda x{:}b_{17}.\ x$$

has type

$$b_{17} \to b_{17}$$

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

$$\lambda x{:}b_1 \to b_2 \to b_3.\ \lambda y{:}b_2.\ \lambda z{:}b_1.\ x\ z\ y$$

has type

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

Example programs

$$\lambda x{:}b_1.\ (\mathsf{A}(x), \mathsf{A}(x))$$

has type

$$b_1 \to ((b_1 + b_7) * (b_1 + b_4))$$

Example programs

$$\lambda f{:}b_1 \to b_3.\ \lambda g{:}b_2 \to b_3.\ \lambda z{:}b_1 + b_2.$$
$$(\mathbf{match}\ z\ \mathbf{with}\ \mathsf{A}x.\ f\ x \mid \mathsf{B}x.\ g\ x)$$

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

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

Empty and Nonempty Types

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

$$b_{17} \to b_{17}$$
$$b_1 \to (b_1 \to b_2) \to b_2$$
$$(b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3$$
$$b_1 \to ((b_1 + b_7) * (b_1 + b_4))$$
$$(b_1 \to b_3) \to (b_2 \to b_3) \to (b_1 + b_2) \to b_3$$
$$(b_1 * b_2) \to b_3 \to ((b_3 * b_1) * b_2)$$

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

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

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

Propositional Logic

With $\to$ for implies, $+$ for inclusive-or and $*$ for and:

$$p ::= b \mid p \to p \mid p * p \mid p + p$$
$$\Gamma ::= \cdot \mid \Gamma, p$$

$\boxed{\Gamma \vdash p}$

$$\frac{\Gamma \vdash p_1 \quad \Gamma \vdash p_2}{\Gamma \vdash p_1 * p_2} \qquad \frac{\Gamma \vdash p_1 * p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 * p_2}{\Gamma \vdash p_2}$$

$$\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 + p_2} \qquad \frac{\Gamma \vdash p_2}{\Gamma \vdash p_1 + p_2}$$

$$\frac{\Gamma \vdash p_1 + p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3}{\Gamma \vdash p_3}$$

$$\frac{p \in \Gamma}{\Gamma \vdash p} \qquad \frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \to p_2} \qquad \frac{\Gamma \vdash p_1 \to p_2 \quad \Gamma \vdash p_1}{\Gamma \vdash p_2}$$

Guess what!!!!

That's *exactly* our type system, erasing terms and changing each τ to a p

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \quad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 : \tau}$$

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

Curry-Howard Isomorphism

- Given a well-typed closed term, 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
  - λ is no more or less made up than implication
- Revisit our examples under the logical interpretation...

Example programs

$$\lambda x{:}b_{17}.\ x$$

is a proof that

$$b_{17} \to b_{17}$$

Example programs

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

is a proof that

$$b_1 \to (b_1 \to b_2) \to b_2$$

Example programs

$$\lambda x{:}b_1 \to b_2 \to b_3.\ \lambda y{:}b_2.\ \lambda z{:}b_1.\ x\ z\ y$$

is a proof that

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

Example programs

$$\lambda x{:}b_1.\ (\mathsf{A}(x), \mathsf{A}(x))$$

is a proof that

$$b_1 \to ((b_1 + b_7) * (b_1 + b_4))$$

Example programs

$$\lambda f{:}b_1 \to b_3.\ \lambda g{:}b_2 \to b_3.\ \lambda z{:}b_1 + b_2.$$
$$(\textbf{match}\ z\ \textbf{with}\ \mathsf{A}x.\ f\ x \mid \mathsf{B}x.\ g\ x)$$

is a proof that

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

Example programs

$$\lambda x{:}b_1 * b_2.\ \lambda y{:}b_3.\ ((y, x.1), x.2)$$

is a proof that

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

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 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":

$$\frac{}{\Gamma \vdash p_1 + (p_1 \to p_2)}$$

(Think "$p + \neg p$" – also equivalent to double-negation $\neg\neg p \to p$)

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

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:

$$((p_1 + (p_1 \to p_2)) * (p_1 \to p_3) * ((p_1 \to p_2) \to p_3)) \to p_3$$

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"

Fix

A "non-terminating proof" is no proof at all

Remember the typing rule for **fix**:

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix}\ e : \tau}$$

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.)

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


Last word on Curry-Howard

It's not just STLC and constructive propositional logic

Every logic has a correspondng typed λ calculus (and no consistent logic has something as "powerful" as **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 *modus ponens*