

CS152: Programming Languages

Lecture 12 — The Curry-Howard Isomorphism

Dan Grossman
Spring 2011

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 $p ::= b \mid p \wedge p \mid p \vee p \mid p \rightarrow p$
- ▶ 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:

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

$$\begin{aligned} e & ::= x \mid \lambda x. e \mid e e \\ & \quad \mid (e, e) \mid e.1 \mid e.2 \\ & \quad \mid \mathbf{A}(e) \mid \mathbf{B}(e) \mid \mathbf{match} \ e \ \mathbf{with} \ \mathbf{A}x. e \mid \mathbf{B}x. e \\ \tau & ::= b \mid \tau \rightarrow \tau \mid \tau * \tau \mid \tau + \tau \end{aligned}$$

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

Example programs

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

has type

$b_{17} \rightarrow b_{17}$

Example programs

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

has type

$$b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2$$

Example programs

$$\lambda x:b_1 \rightarrow b_2 \rightarrow b_3. \lambda y:b_2. \lambda z:b_1. x z y$$

has type

$$(b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3$$

Example programs

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

has type

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

Example programs

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

has type

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

Example programs

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

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 that type):

$$b_{17} \rightarrow b_{17}$$

$$b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2$$

$$(b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3$$

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

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

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

There are also lots of “empty” types (no closed term of that type):

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

And “I” have 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:

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

$\Gamma \vdash p$

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

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

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

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

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

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

$$\frac{p \in \Gamma}{\Gamma \vdash p}$$

$$\frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \rightarrow p_2}$$

$$\frac{\Gamma \vdash p_1 \rightarrow p_2 \quad \Gamma \vdash p_1}{\Gamma \vdash p_2}$$

Guess what!!!!

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

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

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

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{A}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{B}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x:\tau_1 \vdash e_1 : \tau \quad \Gamma, y:\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{match} \ e \ \mathbf{with} \ \mathbf{Ax.} \ e_1 \ | \ \mathbf{By.} \ e_2 : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ e_2 : \tau_1}$$

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

$b_{17} \rightarrow b_{17}$

Example programs

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

is a proof that

$$b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2$$

Example programs

$$\lambda x:b_1 \rightarrow b_2 \rightarrow b_3. \lambda y:b_2. \lambda z:b_1. x \ z \ y$$

is a proof that

$$(b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3$$

Example programs

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

is a proof that

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

Example programs

$\lambda f:b_1 \rightarrow b_3. \lambda g:b_2 \rightarrow b_3. \lambda z:b_1 + b_2.$
(match z with $Ax. f\ x$ | $Bx. g\ x$)

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

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

is a proof that

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

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”:

$$\overline{\Gamma \vdash p_1 + (p_1 \rightarrow p_2)}$$

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

STLC has *no* proof for this; no closed expression has this type

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:

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

Example classical proof

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

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.

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.

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”

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**:

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau}{\Gamma \vdash \mathbf{fix} \ e : \tau}$$

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 corresponding typed λ calculus (and no consistent logic has something as “powerful” as **fix**).

- ▶ Example: When we add universal types (“generics”) in a few lectures, that corresponds to adding universal quantification

If you remember one thing: the typing rule for function application is *modus ponens*