

# CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2008

Lecture 10— Curry-Howard Isomorphism; Evaluation Contexts

# Outline

---

Two totally different topics:

- Curry-Howard Isomorphism
  - Types are propositions
  - Programs are proofs
- Equivalent ways to express evaluation of  $\lambda$ -calculus
  - Evaluation contexts
  - Explicit stacks
  - Closures instead of substitution

A series of equivalent implementations from our operational semantics to a fairly efficient “low-level” implementation!

Note: See ML code for much of the second topic

Evaluation contexts / stacks also let us talk about *continuations*

# 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 ST $\lambda$ C with base types  $b_1, b_2, \dots$ ,  
*no constants*, pairs, and sums

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

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

## 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

---

So we have seen several “nonempty” types (closed terms of that type):

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

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

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

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

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

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

But 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 every  $\tau$  to a  $p$

$\Gamma \vdash e : \tau$

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

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

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

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

# Curry-Howard Isomorphism

---

- Given a closed term that type-checks, we can 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.
- 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
- Let's revisit our examples under the logical interpretation...

## Example proofs

---

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

is a proof that

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

## Example proofs

---

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

is a proof that

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



## Example proofs

---

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

is a proof that

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

## Example proofs

---

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

is a proof that

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

## Example proofs

---

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

is a proof that

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

## Example proofs

---

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

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).
- For decades these were separate 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 ST $\lambda$ C 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”:

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

(Think “ $p$  or not  $p$ ” – also equivalent to double-negation.)

ST $\lambda$ C has *no* proof for this; there is no expression with 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”.

You can still “branch on possibilities”:

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

## Example classical proof

---

Theorem: I can always 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 if it's a weekday, 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 could not prove the theorem above, but you could 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! For 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  $ST\lambda C$  and intuitionistic propositional logic.

*Every* logic has a corresponding typed  $\lambda$  calculus (and no consistent logic has something like 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*.

## Toward Evaluation Contexts

(untyped)  $\lambda$ -calculus with extensions has lots of “boring inductive rules”:

$$\begin{array}{c}
 \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2} \quad \frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \quad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2} \\
 \\
 \frac{e_1 \rightarrow e'_1}{(e_1, e_2) \rightarrow (e'_1, e_2)} \quad \frac{e_2 \rightarrow e'_2}{(v_1, e_2) \rightarrow (v_1, e'_2)} \quad \frac{e \rightarrow e'}{\mathbf{A}(e) \rightarrow \mathbf{A}(e')} \quad \frac{e \rightarrow e'}{\mathbf{B}(e) \rightarrow \mathbf{B}(e')} \\
 \\
 \frac{}{e \rightarrow e'}
 \end{array}$$

**match  $e$  with  $\mathbf{A}x. e_1 \mid \mathbf{B}y. e_2 \rightarrow$  match  $e'$  with  $\mathbf{A}x. e_1 \mid \mathbf{B}y. e_2$**

and some “interesting do-work rules”:

$$\frac{}{(\lambda x. e) v \rightarrow e[v/x]} \quad \frac{}{(v_1, v_2).1 \rightarrow v_1} \quad \frac{}{(v_1, v_2).2 \rightarrow v_2}$$

**match  $\mathbf{A}(v)$  with  $\mathbf{A}x. e_1 \mid \mathbf{B}y. e_2 \rightarrow e_1[v/x]$**

**match  $\mathbf{B}(v)$  with  $\mathbf{A}y. e_1 \mid \mathbf{B}x. e_2 \rightarrow e_2[v/x]$**

## Evaluation Contexts

We can define *evaluation contexts*, which are expressions with one hole where “interesting work” may occur:

$$\begin{aligned} E ::= & [\cdot] \mid E e \mid v E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\ & \mid \mathbf{A}(E) \mid \mathbf{B}(E) \mid (\text{match } E \text{ with } \mathbf{A}x. e_1 \mid \mathbf{B}y. e_2) \end{aligned}$$

Define “filling the hole”  $E[e]$  in the obvious way (see ML code).

Semantics is now just “interesting work” rules (written  $e \xrightarrow{\mathbf{P}} e'$ ) and:

$$\frac{e \xrightarrow{\mathbf{P}} e'}{E[e] \rightarrow E[e']}$$

So far, just concise notation pushing the work to *decomposition*: Given  $e$ , find an  $E, e_a, e'_a$  such that  $e = E[e_a]$  and  $e_a \xrightarrow{\mathbf{P}} e'_a$ .

Theorem (Unique Decomposition): If  $\cdot \vdash e : \tau$ , then  $e$  is a value or there is exactly one decomposition of  $e$ .

## Second Implementation

---

So far two interpreters:

- Old-fashioned small-step: derive a step, and iterate
- Evaluation-context small-step: decompose, fill the whole with the result of the primitive-step, and iterate

Decomposing “all over” each time is awfully redundant (as is the old-fashioned build a full-derivation of each step).

Will be more efficient in next lecture...