# CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2005

Lecture 10— Curry-Howard Isomorphism, Evaluation Contexts, Stacks, Abstract Machines

## Outline

Two totally different topics:

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

  A series of small steps from our operational semantics to a fairly efficient "low-level" implementation!

  Note: `lec10.ml` contains much of today's lecture

Later: Evaluation contexts / stacks will 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
    $p ::= b \mid p \land p \mid p \lor p \mid p \to p \mid \textbf{true} \mid \textbf{false}$
- 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$\boldsymbol{\lambda}$C with base types $b_1, b_2, \ldots$, *no constants*, pairs, and sums

Even without constants, plenty of terms type-check:

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

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

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

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

$\lambda f{:}b_1 \rightarrow b_3.\ \lambda g{:}b_2 \rightarrow b_3.\ \lambda z{:}b_1 + b_2.\ (\mathsf{case}\ z\ x.f\ z \mid x.g\ z)$

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

And plenty of types have no terms with that type:

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

Punchline: I knew all that because of logic, not PL!

# Propositional Logic

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

$$\frac{p_1}{p_1 + p_2} \qquad \frac{p_2}{p_1 + p_2} \qquad \frac{p_1 \qquad p_2}{p_1 * p_2} \qquad \frac{p_1 * p_2}{p_1} \qquad \frac{p_1 * p_2}{p_2}$$

$$\frac{p_1 \rightarrow p_2 \qquad p_1}{p_2}$$

We have one language construct and typing rule for each one!

The Curry-Howard Isomorphism: For every typed $\lambda$-calculus there is a logic and for every logic a typed $\lambda$-calculus such that:

- If there is a closed expression with a type, then the corresponding proposition is provable in the logic.
- If there is no such expression, then the corresponding proposition is not provable in the logic.

# Why care?

Because:

- This is just fascinating.
- 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
- Is pretty good "evidence" that $\boldsymbol{\lambda}$-calculus is no more (or less) "made up" than logic.

So, every typed $\boldsymbol{\lambda}$-calculus is a proof system for a logic...

Is ST$\boldsymbol{\lambda}$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":

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

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

ST$\lambda$C has *no* proof rule 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":

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

# 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! For example: $\textbf{fix}\ \lambda x{:}b_3.\ x$ has type $b_3$.

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

# Toward Evaluation Contexts

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

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v\ e_2 \rightarrow v\ e_2'} \qquad \frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \qquad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}$$

$$\frac{e_1 \rightarrow e_1'}{(e_1, e_2) \rightarrow (e_1', e_2)} \qquad \frac{e_2 \rightarrow e_2'}{(v_1, e_2) \rightarrow (v_1, e_2')} \qquad \frac{e \rightarrow e'}{\mathsf{inl}(e) \rightarrow \mathsf{inl}(e')}$$

$$\frac{e \rightarrow e'}{\mathsf{inr}(e) \rightarrow \mathsf{inr}(e')} \qquad \frac{e \rightarrow e'}{\mathsf{case}\ e\ x.e_1 \mid x.e_2 \rightarrow \mathsf{case}\ e'\ x.e_1 \mid x.e_2}$$

and some "interesting do-work rules":

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

$$\frac{}{\mathsf{case}\ \mathsf{inl}(v)\ x.e_1 \mid x.e_2 \rightarrow e_1[v/x]} \qquad \frac{}{\mathsf{case}\ \mathsf{inr}(v)\ x.e_1 \mid x.e_2 \rightarrow e_2[v/x]}$$

# Evaluation Contexts

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

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \textsf{case}\ E\ x.e_1 \mid x.e_2$$

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

Semantics is now just "interesting work" rules (written $e \xrightarrow{\text{p}} e'$) and:

$$\frac{e \xrightarrow{\text{p}} e'}{E[e] \rightarrow E[e']}$$

So far, this is just concise notation that pushes the work to *decomposition*: Given $e$, find an $E$ and $e_a$ such that $e = E[e_a]$ and $e_a$ can take a primitive step.

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).

We can "incrementally maintain the decomposition" if we represent it conveniently. Instead of nested contexts, we can keep a list:

$$S ::= \cdot \mid Lapp(e)::S \mid Rapp(v)::S \mid Lpair(e)::S \mid \ldots$$

See the code: This representation is *isomorphic* (there's a bijection) to evaluation contexts.

## Stack-based machine

This new form of evaluation-context is a stack.

Since we don't re-decompose at each step, our "program state" is a stack and an expression.

At each step, the stack may grow (to recur on a nested expression) or shrink (to do a primitive step)

Now that we have an explicit stack, we are not using the meta-language's call-stack (the interpreter is just a while-loop).

But substitution is still using the meta-language's call-stack.

## Stack-based with environments

Our last step uses environments, much like you will in homework 3.

Now *everything* in our interpreter is tail-recursive (beyond the explicit representation of environments and stacks, we need only $O(1)$ space).

You could implement this last interpreter in assembly without using a call instruction.

## Conclusions

Proving equivalence of each version of our interpreter with the next is tractable.

In our last version, every primitive step is $O(1)$ time and space *except* variable lookup (but that's easily fixed in a compiler).

Perhaps more interestingly, evaluation contexts "give us a handle" on the "surrounding computation", which will let us do funky things like make "stacks" first-class in the language (homework 4?).