CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2006

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

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
    $p ::= b \mid p \wedge p \mid p \vee p \mid p \rightarrow p \mid \textbf{true} \mid \textbf{false}$
- 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 STλC with base types $b_1, b_2, \ldots$, *no constants*, pairs, and sums

Even without constants, plenty of terms type-check:

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

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

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

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

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

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

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 λ-calculus there is a logic and for every logic a typed λ-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
- Shows $\boldsymbol{\lambda}$-calculus is no more (or less) "made up" than logic.
  - Type systems are not *ad hoc* piles of rules.

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":

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

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

STλC 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":

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

Fix

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

Remember the typing rule for fix:

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

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).

Toward Evaluation Contexts

(untyped) λ-calculus with extensions has lots of "boring inductive rules":

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

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')} \qquad \frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')}$$

$$\frac{e \to e'}{\textsf{match } e \textsf{ with A}x.\ e_1 \mid \textsf{B}y.\ e_2 \to \textsf{match } e' \textsf{ with A}x.\ e_1 \mid \textsf{B}y.\ e_2}$$

and some "interesting do-work rules":

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

$$\frac{}{\textsf{match A}(v) \textsf{ with A}x.\ e_1 \mid \textsf{B}y.\ e_2 \to e_1[v/x]}$$

$$\frac{}{\textsf{match B}(v) \textsf{ with A}y.\ e_1 \mid \textsf{B}x.\ e_2 \to e_2[v/x]}$$

Evaluation Contexts

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

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

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

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

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

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 $e_a \xrightarrow{p} e'_a$.

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).

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 each interpreter version equivalent to 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" (called *continuations*) first-class in the language.

- "get current continuation; bind it to a variable"
- "replace current continuation with saved one"

$$
\begin{aligned}
e &::= \ldots \mid \textbf{letcc } x.\ e \mid \textbf{throw } e\ e \mid \textbf{cont } E \\
v &::= \ldots \mid \textbf{cont } E \\
E &::= \ldots \mid \textbf{throw } E\ e \mid \textbf{throw } v\ E
\end{aligned}
$$

$$\overline{E[\textbf{letcc } x.\ e] \rightarrow E[e[\textbf{cont } E/x]]} \qquad \overline{E[\textbf{throw } (\textbf{cont } E')\ v] \rightarrow E'[v]}$$