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 $\lambda$-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
    \[ p ::= b \mid p \land p \mid p \lor p \mid p \to p \mid \text{true} \mid \text{false} \]
- 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\(\lambda\)C with base types \(b_1, b_2, \ldots\), no constants, pairs, and sums

Even without constants, plenty of terms type-check:

\[
\begin{align*}
\lambda x : b_{17}. & \; x \\
\lambda x : b_1. & \; \lambda f : b_1 \rightarrow b_2. \; f \; x \\
\lambda x : b_1 \rightarrow b_2 \rightarrow b_3. & \; \lambda y : b_2. \; \lambda z : b_1. \; x \; z \; y \\
\lambda x : b_1. & \; (\text{inl}(x), \text{inl}(x)) \\
\lambda f : b_1 \rightarrow b_3. & \; \lambda g : b_2 \rightarrow b_3. \; \lambda z : b_1 + b_2. \; (\text{case} \; z \; x. \; f \; z \mid x. \; g \; z) \\
\lambda x : b_1 \ast b_2. & \; \lambda y : b_3. \; ((y, x.1), x.2) \\
\end{align*}
\]

And plenty of types have no terms with that type:

\[
\begin{align*}
b_1 & \quad b_1 \rightarrow b_2 \quad b_1 + (b_1 \rightarrow b_2) \\
b_1 & \rightarrow (b_2 \rightarrow b_1) \rightarrow b_2
\end{align*}
\]

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

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

\[
\begin{array}{cccccc}
p_1 & p_2 & p_1 & p_2 & p_1 \ast p_2 & p_1 \ast p_2 \\
p_1 + p_2 & p_1 + p_2 & p_1 * p_2 & p_1 & p_1 & p_2 \\
\end{array}
\]

\[
p_1 \to p_2 \quad p_1 \\
\hline
p_2 \quad p_2
\]

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 $\lambda$-calculus is no more (or less) “made up” than logic.

So, every typed $\lambda$-calculus is a proof system for a 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”:

\[ p_1 + (p_1 \rightarrow p_2) \]

(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”:

\[
((p_1 + (p_1 \rightarrow p_2)) \ast (p_1 \rightarrow p_3) \ast ((p_1 \rightarrow p_2) \rightarrow p_3)) \rightarrow p_3
\]
A “non-terminating proof” is no proof at all.

Remember the typing rule for fix:

$$
\Gamma \vdash e : \tau \rightarrow \tau \\
\begin{array}{c}
\Gamma \vdash \text{fix } e : \tau
\end{array}
$$

That let’s us prove anything! For example: $\text{fix } \lambda x : b_3. \ x$ 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":

$\frac{e_1 \to e'_1}{e_1 \; e_2 \to e'_1 \; e_2}$  $\frac{e_2 \to e'_2}{v \; e_2 \to v \; e'_2}$  $\frac{e \to e'}{e.1 \to e'.1}$  $\frac{e \to e'}{e.2 \to e'.2}$

$\frac{e_1 \to e'_1}{(e_1, e_2) \to (e'_1, e_2)}$  $\frac{e_2 \to e'_2}{(v_1, e_2) \to (v_1, e'_2)}$  $\frac{e \to e'}{\text{inl}(e) \to \text{inl}(e')}$

$\frac{e \to e'}{\text{inr}(e) \to \text{inr}(e')}$  $\frac{e \to e'}{\text{case } e \; x.e_1 \mid x.e_2 \to \text{case } e' \; x.e_1 \mid x.e_2}$

and some "interesting do-work rules":

$\frac{(\lambda x. \; e) \; v \to e[v/x]}{(v_1, v_2).1 \to v_1}$  $\frac{(v_1, v_2).2 \to v_2}{\text{case } \text{inl}(v) \; x.e_1 \mid x.e_2 \to e_1[v/x]}$  $\frac{\text{case } \text{inr}(v) \; x.e_1 \mid x.e_2 \to e_2[v/x]}$
Evaluation Contexts

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

\[ E ::= [\cdot] \mid E \ e \mid \nu \ E \mid (E, e) \mid (\nu, E) \mid \text{case } E \ x.e_1 \mid x.e_2 \]

Define “filling the hole” \( E[e] \) in the obvious way (see ML code).
Semantics is now just “interesting work” rules (written \( e \xrightarrow{P} e' \)) and:

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

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:

\[ S ::= \cdot \mid \text{Lapp}(e)::S \mid \text{Rapp}(v)::S \mid \text{Lpair}(e)::S \mid \ldots \]

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?).