CSE 505: Concepts of Programming Languages

Dan Grossman
Fall 2009
Lecture 10—Curry-Howard Isomorphism; Evaluation Contexts
Outline

A couple left-over topics from last lecture:

- Sums in “real” programming languages
- A fuller explanation of why it’s called fix

Two totally different topics:

- Curry-Howard Isomorphism
  - Types are propositions
  - Programs are proofs
- Evaluation contexts, explicit stacks, and first-class continuations
Recall sums

\[ e ::= \ldots \mid A(e) \mid B(e) \mid \text{match } e \text{ with } Ax. \ e \mid Bx. \ e \]

\[ v ::= \ldots \mid A(v) \mid B(v) \]

\[ \tau ::= \ldots \mid \tau_1 + \tau_2 \]

\[
\begin{align*}
\text{match } A(v) \text{ with } Ax. \ e_1 & \mid By. \ e_2 \rightarrow e_1[v/x] \\
\text{match } B(v) \text{ with } Ax. \ e_1 & \mid By. \ e_2 \rightarrow e_2[v/y] \\
\end{align*}
\]

\[ e \rightarrow e' \]

\[
\begin{align*}
A(e) & \rightarrow A(e') \\
B(e) & \rightarrow B(e') \\
e & \rightarrow e' \\
\end{align*}
\]

\[
\begin{align*}
\text{match } e \text{ with } Ax. \ e_1 & \mid By. \ e_2 \rightarrow \text{match } e' \text{ with } Ax. \ e_1 \mid By. \ e_2 \\
\Gamma \vdash e : \tau_1 & \quad \Gamma \vdash e : \tau_2 \\
\Gamma \vdash A(e) : \tau_1 + \tau_2 & \quad \Gamma \vdash B(e) : \tau_1 + \tau_2 \\
\Gamma \vdash e : \tau_1 + \tau_2 & \quad \Gamma, x:\tau_1 \vdash e_1 : \tau \\
& \quad \Gamma, y:\tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{match } e \text{ with } Ax. \ e_1 & \mid By. \ e_2 : \tau \\
\end{align*}
\]
What are sums for?

- Pairs, structs, records, aggregates are fundamental data-builders
- Sums are just as fundamental: “this or that not both”
- You have seen how Caml does sums (datatypes)
- Worth showing how C and Java do the same thing
  - A primitive in one language is an idiom in another
Sums in C

type t = A of t1 | B of t2 | C of t3

match e with A x -> ...

One way in C:

struct t {
    enum {A, B, C} tag;
    union {t1 a; t2 b; t3 c;} data;
};

... switch(e->tag){ case A: t1 x=e->data.a; ...

• No static checking that tag is obeyed

• As fat as the fattest variant (avoidable with casts)
  – Mutation costs us again!

• Shameless plug: Cyclone has ML-style datatypes
Sums in Java

type t = A of t1 | B of t2 | C of t3
match e with A x -> ...

One way in Java (t4 is the match-expression’s type):

abstract class t {abstract t4 m();}
class A extends t { t1 x; t4 m(){...}}
class B extends t { t2 x; t4 m(){...}}
class C extends t { t3 x; t4 m(){...}}
... e.m() ...

• A new method for each match expression
• Supports extensibility via new variants (subclasses) instead of extensibility via new operations (match expressions)
Recall Fix

Note: Like let rec but using a \( \lambda \) to bind the name

\[
e ::= \ldots \mid \text{fix } e
\]

\[
e \rightarrow e'
\]

\[
\text{fix } e \rightarrow \text{fix } e'
\]

\[
\text{fix } \lambda x. e \rightarrow e[\text{fix } \lambda x. e/x]
\]

\[
\Gamma \vdash e : \tau \rightarrow \tau
\]

\[
\Gamma \vdash \text{fix } e : \tau
\]

Factorial example:

\[
\text{fix } \lambda f. \lambda n. \text{if } n < 1 \text{ then } 1 \text{ else } n \ast (f(n - 1))
\]

- Operationally, substitution unrolls the recursion one level

- For type system, \( \lambda f. \lambda n. \text{if } n < 1 \text{ then } 1 \text{ else } n \ast (f(n - 1)) \)

  has type \( \text{int } \rightarrow \text{int} \) \( \rightarrow \) \( \text{int } \rightarrow \text{int} \).
Why called fix?

My slide in the last lecture could have explained fix-points much better...

In math, the fix-point of a function $g$ is an $x$ such that $g(x) = x$.

- This makes sense only if $g$ has type $\tau \to \tau$ for some $\tau$.
- A particular $g$ could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type $\texttt{int} \to \texttt{int}$:
  - $\lambda x. x + 1$ has no fix-points
  - $\lambda x. x * 0$ has one fix-point
  - $\lambda x. \text{absolute\_value}(x)$ has an infinite number of fix-points
  - $\lambda x. \text{if } x < 10 \&\& x > 0 \text{ then } x \text{ else } 0$ has 10 fix-points
Higher types

At higher types like \((\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})\), the notion of fix-point is exactly the same (but harder to think about)

- For what inputs \(f\) of type \(\text{int} \rightarrow \text{int}\) is \(g(f) = f\).

Examples:

- \(\lambda f. \lambda x. (f \ x) + 1\) has no fix-points
- \(\lambda f. \lambda x. (f \ x) \ast 0\) (or just \(\lambda f. \lambda x. 0\)) has 1 fix-point
  - The function that always returns 0
  - In math, there is exactly one such function (cf. equivalence)
- \(\lambda f. \lambda x. \text{absolute\_value}(f \ x)\) has an infinite number of fix-points:
  Any function that never returns a negative result
Back to factorial

Now, what are the fix-points of
\[ \lambda f. \lambda x. \text{if } x < 1 \text{ then } 1 \text{ else } x * (f(x - 1)) \]?

It turns out there is exactly one (in math): the factorial function!

And \text{fix } \lambda f. \lambda x. \text{if } x < 1 \text{ then } 1 \text{ else } x * (f(x - 1)) \text{ behaves just like the factorial function, i.e., it behaves just like the fix-point of } \lambda f. \lambda x. \text{if } x < 1 \text{ then } 1 \text{ else } x * (f(x - 1)).

(This isn’t really important, but I like explaining terminology and showing that programming is deeply connected to mathematics.)
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 $p ::= b \mid p \land p \mid p \lor p \mid p \rightarrow p$
- 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 \(b_1, b_2, \ldots\), \textit{no constants}, pairs, and sums

\[
e ::= x | \lambda x. e | e e
| (e, e) | e.1 | e.2
| A(e) | B(e) | \text{match } e \text{ with } A x. e | B x. e
\]

\[
\tau ::= b | \tau \rightarrow \tau | \tau \ast \tau | \tau + \tau
\]

Even without constants, plenty of terms type-check with \(\Gamma = \cdot \ldots\)
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

\[ \lambda x : b_1. \ (A(x), A(x)) \]

has type

\[ b_1 \rightarrow ((b_1 + b_7) \ast (b_1 + b_4)) \]
Example programs

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

has type

\[ (b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3 \]
Example programs

\[ \lambda x : b_1 \ast b_2. \lambda y : b_3. ((y, x.1), x.2) \]

has type

\[(b_1 \ast b_2) \rightarrow b_3 \rightarrow ((b_3 \ast b_1) \ast b_2)\]
Empty and Nonempty Types

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

\[ b_{17} \rightarrow b_{17} \]
\[ b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2 \]
\[ (b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3 \]
\[ b_1 \rightarrow ((b_1 + b_7) \ast (b_1 + b_4)) \]
\[ (b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3 \]
\[ (b_1 \ast b_2) \rightarrow b_3 \rightarrow ((b_3 \ast b_1) \ast b_2) \]

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 → for implies, + for inclusive-or and * for and:

\[ p ::= b | p \rightarrow p | p * p | p + p \]

\[ \Gamma ::= \cdot | \Gamma, p \]

\[ \Gamma \vdash p \]

\[ \begin{array}{c}
\Gamma \vdash p_1 \quad \Gamma \vdash p_2 \\
\hline
\Gamma \vdash p_1 * p_2
\end{array} \]

\[ \begin{array}{c}
\Gamma \vdash p_1 \quad \Gamma \vdash p_2 \\
\hline
\Gamma \vdash p_1 + p_2
\end{array} \quad \begin{array}{c}
\Gamma \vdash p_1 \quad \Gamma \vdash p_2 \\
\hline
\Gamma \vdash p_1 \rightarrow p_2
\end{array} \quad \begin{array}{c}
\Gamma \vdash p_3 \\
\hline
\Gamma \vdash p_1 \rightarrow p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3
\end{array} \]

\[ \begin{array}{c}
p \in \Gamma \\
\hline
\Gamma \vdash p
\end{array} \quad \begin{array}{c}
\Gamma, p_1 \vdash p_2 \\
\hline
\Gamma \vdash p_2
\end{array} \quad \begin{array}{c}
\Gamma \vdash p_1 \quad \Gamma \vdash p_2 \\
\hline
\Gamma \vdash p_1
\end{array} \]

Dan Grossman
CSE505 Fall 2009, Lecture 10
Guess what!!!!

That’s exactly our type system, erasing terms and changing every $\tau$ to a $p$

$$\Gamma \vdash e : \tau$$

$$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2$$

$$\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2$$

$$\Gamma \vdash e.1 : \tau_1$$

$$\Gamma \vdash e.2 : \tau_2$$

$$\Gamma \vdash e : \tau_1$$

$$\Gamma \vdash A(e) : \tau_1 + \tau_2$$

$$\Gamma \vdash e : \tau_2$$

$$\Gamma \vdash B(e) : \tau_1 + \tau_2$$

$$\Gamma \vdash e : \tau_1 + \tau_2$$

$$\Gamma, x:\tau_1 \vdash e_1 : \tau$$

$$\Gamma, y:\tau_2 \vdash e_2 : \tau$$

$$\Gamma \vdash \text{match } e \text{ with } Ax. \ e_1 \mid By. \ e_2 : \tau$$

$$\Gamma(x) = \tau$$

$$\Gamma, x:\tau_1 \vdash e : \tau_2$$

$$\Gamma \vdash e_1 : \tau_2 \to \tau_1$$

$$\Gamma \vdash e_2 : \tau_2$$

$$\Gamma \vdash e_1 \ e_2 : \tau_1$$
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

\[ \lambda x: b_1. (A(x), A(x)) \]

is a proof that

\[ b_1 \rightarrow ((b_1 + b_7) * (b_1 + b_4)) \]
Example proofs

$$\begin{align*}
  \lambda f : b_1 \to b_3. &\quad \lambda g : b_2 \to b_3. &\quad \lambda z : b_1 + b_2. \\
  (\text{match } z \text{ with } A x. f x \mid B x. g x)
\end{align*}$$

is a proof that

$$(b_1 \to b_3) \to (b_2 \to b_3) \to (b_1 + b_2) \to b_3$$
Example proofs

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

is a proof that

\( (b_1 \star b_2) \rightarrow b_3 \rightarrow ((b_3 \star b_1) \star b_2) \)
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”:

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

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

ST\( \lambda \)C has no proof for this; there is no closed 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”:

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

Remember the typing rule for fix:

\[
\frac{\Gamma \vdash e : \tau \rightarrow \tau}{\Gamma \vdash \text{fix } e : \tau}
\]

That lets us prove anything! For example: \(\text{fix } \lambda x : b_3. \ x\) 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.)

\[
\begin{align*}
\text{let rec } f \ x &= f \ x \\
\text{let } z &= f \ 0
\end{align*}
\]
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 \texttt{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”:

- $e_1 \rightarrow e'_1$
- $e_2 \rightarrow e'_2$
- $e \rightarrow e'$
- $e \rightarrow e'$

\[
\frac{e_1 \rightarrow e'_1}{e_1 \cdot e_2 \rightarrow e'_1 \cdot e_2} \quad \frac{e_2 \rightarrow e'_2}{v \cdot e_2 \rightarrow v \cdot e'_2} \quad \frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \quad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}
\]

- $e_1 \rightarrow e'_1$
- $e_2 \rightarrow e'_2$
- $e \rightarrow e'$
- $e \rightarrow e'$

\[
\frac{(e_1, e_2) \rightarrow (e'_1, e'_2)}{(v_1, e_2) \rightarrow (v_1, e'_2)} \quad \frac{A(e) \rightarrow A(e')}{A(e) \rightarrow A(e')} \quad \frac{B(e) \rightarrow B(e')}{B(e) \rightarrow B(e')}
\]

- $e \rightarrow e'$

match $e$ with $Ax. \ e_1 \mid By. \ e_2 \rightarrow$ match $e'$ with $Ax. \ e_1 \mid By. \ e_2$

and some “interesting do-work rules”:

- $(\lambda x. \ e) \ v \rightarrow e[v/x]$
- $(v_1, v_2).1 \rightarrow v_1$
- $(v_1, v_2).2 \rightarrow v_2$

match $A(v)$ with $Ax. \ e_1 \mid By. \ e_2 \rightarrow e_1[v/x]$

match $B(v)$ with $Ay. \ e_1 \mid Bx. \ e_2 \rightarrow e_2[v/x]$
Evaluation Contexts

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

\[
E ::= [\cdot] \mid E \ e \mid v \ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\
\mid A(E) \mid B(E) \mid (\text{match } E \text{ with } Ax. \ e_1 \mid By. \ e_2)
\]

Define “filling the hole” \( E[e] \) in the obvious way (stapling).

Semantics now uses two judgments \( e \rightarrow e' \) and \( e \xrightarrow{P} e' \), but the former has only 1 rule and the latter has just the “interesting work”:

\[
\begin{align*}
&\quad e \xrightarrow{P} e' \\
\Rightarrow &\quad E[e] \rightarrow E[e'] \\
&\quad (\lambda x. \ e) v \xrightarrow{P} e[v/x] \\
&\quad (v_1, v_2).1 \xrightarrow{P} v_1 \\
&\quad (v_1, v_2).2 \xrightarrow{P} v_2 \\
&\quad \text{match } A(v) \text{ with } Ax. \ e_1 \mid By. \ e_2 \xrightarrow{P} e_1[v/x] \\
&\quad \text{match } B(v) \text{ with } Ay. \ e_1 \mid Bx. \ e_2 \xrightarrow{P} e_2[v/x]
\end{align*}
\]
So what?

So far, all we have done is rearrange our semantics to be more concise

- Each boring rule become a form of $E$

Evaluation relies on *decomposition* (unstapling the right subtree):
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 $e$.

- Hence evaluation is deterministic
- In fact it’s still CBV left-to-right

But the real power from defining $E$ is that it lets us *reify* continuations (evaluation stacks) ...
Continuations

First-class continuations in one slide:

\[
\begin{align*}
e & ::= \ldots \mid \text{letcc } x. \ e \mid \text{throw } e \ e \mid \text{cont } E \\
v & ::= \ldots \mid \text{cont } E \\
E & ::= \ldots \mid \text{throw } E \ e \mid \text{throw } v \ E
\end{align*}
\]

\[
E[\text{letcc } x. \ e] \rightarrow E[(\lambda x. \ e)(\text{cont } E)]
\]

\[
E[\text{throw } (\text{cont } E') \ v] \rightarrow E'[v]
\]

Very powerful and general: For example, non-preemptive multithreading in the language. Exceptions. “Time travel.”
Connection to interpreters

A “real” (efficient, natural) interpreter for lambda-calculus (or ML) would not be like our small-step semantics

• Would re-decompose the whole program for each step!

Instead, maintain the decomposition incrementally

• With a stack to remember “what to work on next”!

Also, don’t use substitution; use environments (see your homework)

• At this point, need just one while-loop, pairs, and malloc

And if your stacks are heap-allocated and immutable, you can implement continuation operations (letcc and throw) in $O(1)$ time.

• A nice (and provably correct) sequence of more primitive interpreters

• Can post Caml code for the curious