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

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{A}(e) \mid \mathsf{B}(e) \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e \mid \mathsf{B}x.\ e \\ v & ::= & \ldots \mid \mathsf{A}(v) \mid \mathsf{B}(v) \\ \tau & ::= & \ldots \mid \tau_1 + \tau_2 \end{array}$$

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

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

$$\frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')}$$

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

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 : \tau}$$

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 λ to bind the name

$$e ::= \ldots \mid \textbf{fix } e$$

$$\frac{e \to e'}{\textbf{fix } e \to \textbf{fix } e'} \qquad \frac{}{\textbf{fix } \lambda x.\ e \to e[\textbf{fix } \lambda x.\ e/x]}$$

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix } e : \tau}$$

Factorial example:
$\textbf{fix } \lambda f.\ \lambda n.\ \textbf{if } n < 1 \textbf{ then } 1 \textbf{ else } n * (f(n-1))$

- Operationally, substitution unrolls the recursion one level
- For type system, $\lambda f.\ \lambda n.\ \textbf{if } n < 1 \textbf{ then } 1 \textbf{ else } n * (f(n-1))$ has type $(\textbf{int} \to \textbf{int}) \to (\textbf{int} \to \textbf{int})$.

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 $\tau \rightarrow \tau$ for some τ.
- A particular g could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type **int** $\rightarrow$ **int**:
  - $\lambda x.\ x + 1$ has no fix-points
  - $\lambda x.\ x * 0$ has one fix-point
  - $\lambda x.$ absolute_value(x) has an infinite number of fix-points
  - $\lambda x.$ **if** $x < 10\ \&\&\ x > 0$ **then** x **else** 0 has 10 fix-points

Higher types

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

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

Examples:

- $\lambda f.\ \lambda x.\ (f\ x) + 1$ has no fix-points
- $\lambda f.\ \lambda x.\ (f\ x) * 0$ (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)
- $\lambda f.\ \lambda x.$ 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
$\boldsymbol{\lambda f.\ \lambda x.}$ **if** $\boldsymbol{x < 1}$ **then** $\boldsymbol{1}$ **else** $\boldsymbol{x * (f(x-1))}$?

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

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

(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 $p ::= b \mid p \land p \mid p \lor p \mid p \to p$
- 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$\boldsymbol{\lambda}$C with base types $\boldsymbol{b_1, b_2, \ldots}$, *no constants*, pairs, and sums

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.\ e \mid e\ e \\
 & \mid & (e, e) \mid e.1 \mid e.2 \\
 & \mid & \mathsf{A}(e) \mid \mathsf{B}(e) \mid \textbf{match } e \textbf{ with } \mathsf{A}x.\ e \mid \mathsf{B}x.\ e \\
\tau & ::= & b \mid \tau \rightarrow \tau \mid \tau * \tau \mid \tau + \tau
\end{array}
$$

Even without constants, plenty of terms type-check with $\boldsymbol{\Gamma = \cdot}$ ...

Example programs

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

has type

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

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

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

has type

$$\boldsymbol{(b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3}$$

Example programs

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

has type

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

Example programs

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

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):

$b_{17} \to b_{17}$

$b_1 \to (b_1 \to b_2) \to b_2$

$(b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3$

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

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

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

But there are also lots of "empty" types (no closed term of that type):

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

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:

$$
\begin{array}{lcl}
p & ::= & b \mid p \rightarrow p \mid p * p \mid p + p \\
\Gamma & ::= & \cdot \mid \Gamma, p
\end{array}
$$

$\boxed{\Gamma \vdash p}$

$$
\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 * p_2} \qquad \frac{\Gamma \vdash p_1 * p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 * p_2}{\Gamma \vdash p_2}
$$

$$
\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 + p_2} \qquad \frac{\Gamma \vdash p_2}{\Gamma \vdash p_1 + p_2} \qquad \frac{\Gamma \vdash p_1 + p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3}{\Gamma \vdash p_3}
$$

$$
\frac{p \in \Gamma}{\Gamma \vdash p} \qquad \frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \rightarrow p_2} \qquad \frac{\Gamma \vdash p_1 \rightarrow p_2 \qquad \Gamma \vdash p_1}{\Gamma \vdash p_2}
$$

Guess what!!!!

That's *exactly* our type system, erasing terms and changing every τ to a p

$\boxed{\Gamma \vdash e : \tau}$

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

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \quad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 : \tau}$$

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

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
  - $\boldsymbol{\lambda}$ is no more or less made up than implication
- Let's revisit our examples under the logical interpretation...

Example proofs

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

is a proof that

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

Example proofs

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

is a proof that

$$\boldsymbol{b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2}$$

Example proofs

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

is a proof that

$$\boldsymbol{(b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3}$$

Example proofs

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

is a proof that

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

Example proofs

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

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 $\boldsymbol{\lambda}$-calculus is a proof system for some 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":

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

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

STλC 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":

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

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:

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix}\ e : \tau}$$

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 STλC and intuitionistic propositional logic.

Every logic has a correspondng typed λ 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) $\boldsymbol{\lambda}$-calculus with extensions has lots of "boring inductive rules":

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

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

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

and some "interesting do-work rules":

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

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

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

Evaluation Contexts

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

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

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":

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

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

$$\overline{\mathsf{match}\ \mathsf{A}(v)\ \mathsf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \xrightarrow{p} e_1[v/x]}$$

$$\overline{\mathsf{match}\ \mathsf{B}(v)\ \mathsf{with}\ \mathsf{A}y.\ e_1 \mid \mathsf{B}x.\ e_2 \xrightarrow{p} e_2[v/x]}$$

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:

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

$$\frac{}{E[\textbf{letcc}\ x.\ e] \to E[(\lambda x.\ e)(\textbf{cont}\ E)]}$$

$$\frac{}{E[\textbf{throw}\ (\textbf{cont}\ E')\ v] \to E'[v]}$$

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 $\boldsymbol{O(1)}$ time.

- A nice (and provably correct) sequence of more primitive interpreters
- Can post Caml code for the curious