CSE 505:
Concepts of Programming Languages

Dan Grossman

Fall 2008

Lecture 20— Effect Systems; Continuation Passing;

Course Summary and Everything Else

79.5 Minutes of PL left

Today's lecture:

1. Effect systems, finally :-)
2. Continuation Passing — and CPS transformation
3. Overview of what we did — and didn't — do

Will likely go too fast for it all to sink in

- But at least "you know it's out there"

Today's material will be 0%–3% of the final

But overview still hopefully very useful for understanding the course

More about final exam

- Thursday December 11, 8:30-10:20AM
  - I didn't pick the early time
- Intended to test the material since the midterm (lectures 10–19 and homeworks 3–5), but obviously material accumulates
- Will post old exams and cover sheet
- You can bring your own reference sheet

Type-and-Effect Systems

Our plain-old type systems have judgments like $\Gamma \vdash e : \tau$ to mean:

- e won't get stuck
- If e produces a value, that value has type τ

Adding *effects* reuses the "plumbing" of our typing rules to compute something about "how e executes".

- There are many things we might want to conservatively approximate
  - Example: What exceptions might get thrown
- All effect systems are very similar, especially how they treat functions
  - Example: All values have no effect since their "computation" does nothing

First a type system

(In this example, exceptions raise constant strings s)

$$\tau \quad ::= \quad \textsf{bool} \mid \tau \rightarrow \tau \mid \tau * \tau$$

$$e \quad ::= \quad x \mid \textsf{true} \mid \textsf{false} \mid \lambda x.\ e \mid e\ e \mid (e, e) \mid e.1 \mid e.2$$
$$\mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \mid \textsf{raise}\ s \mid \textsf{try}\ e\ \textsf{handle}\ s\ e$$

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

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash \textsf{true} : \textsf{bool}} \qquad \frac{}{\Gamma \vdash \textsf{false} : \textsf{bool}}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

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

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 : \tau}$$

$$\frac{}{\Gamma \vdash \textsf{raise}\ s : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textsf{try}\ e_1\ \textsf{handle}\ s\ e_2 : \tau}$$

Add effects

$$\epsilon \ ::= \ \ldots\text{sets of strings}\ldots$$

$$\tau \ ::= \ \textbf{bool} \mid \tau \xrightarrow{\epsilon} \tau \mid \tau * \tau$$

$$e \ ::= \ x \mid \textbf{true} \mid \textbf{false} \mid \lambda x.\ e \mid e\ e \mid (e, e) \mid e.1 \mid e.2$$
$$\mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{raise}\ s \mid \textbf{try}\ e\ \textbf{handle}\ s\ e$$

$$\boxed{\Gamma \vdash e : \tau; \epsilon} \qquad \frac{}{\Gamma \vdash x : \Gamma(x); \emptyset} \qquad \frac{}{\Gamma \vdash \textbf{true} : \textbf{bool}; \emptyset} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool}; \emptyset}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2; \epsilon}{\Gamma \vdash \lambda x.\ e : \tau_1 \xrightarrow{\epsilon} \tau_2; \emptyset} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\epsilon_3} \tau_1; \epsilon_1 \qquad \Gamma \vdash e_2 : \tau_2; \epsilon_2}{\Gamma \vdash e_1\ e_2 : \tau_1; \epsilon_1 \cup \epsilon_2 \cup \epsilon_3}$$

$$\frac{\Gamma \vdash e_1 : \tau_1; \epsilon_1 \qquad \Gamma \vdash e_2 : \tau_2; \epsilon_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2; \epsilon_1 \cup \epsilon_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2; \epsilon}{\Gamma \vdash e.1 : \tau_1; \epsilon} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2; \epsilon}{\Gamma \vdash e.2 : \tau_2; \epsilon}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool}; \epsilon_1 \qquad \Gamma \vdash e_2 : \tau; \epsilon_2 \qquad \Gamma \vdash e_3 : \tau; \epsilon_3}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau; \epsilon_1 \cup \epsilon_2 \cup \epsilon_3}$$

$$\frac{}{\Gamma \vdash \textbf{raise}\ s : \tau; \{s\}} \qquad \frac{\Gamma \vdash e_1 : \tau; \epsilon_1 \qquad \Gamma \vdash e_2 : \tau; \epsilon_2}{\Gamma \vdash \textbf{try}\ e_1\ \textbf{handle}\ s\ e_2 : \tau; (\epsilon_1 - \{s\}) \cup \epsilon_2}$$

Key facts

Soundness: If $\cdot \vdash e : \tau; \epsilon$ and e raises uncaught exception s, then $s \in \epsilon$.

- Corollary to Preservation and Progress (once you define the operational semantics for exceptions)

All effect systems work this way:

- Values effectless
- Functions have *latent effects*
- Conservative due to if and try/handle
- Subeffecting (not shown) is sound and important
  - Functions covariant in effects

Only a couple rules special to this effect system

- Not always sets and $\cup$

Other effect systems

- Definitely terminates (0) or possibly diverges (1)
  - Give **fix** e effect 1
  - Give values effect 0
  - Treat $\cup$ as max
  - No change to rules for functions, pairs, conditionals, etc.
- What type casts might occur (Nita POPL08)
- Are the right variables used in transactions (Moore POPL08)
- Does code obey a locking protocol
- ...

Really a general way to lift static analysis to higher-order functions

- And you want things like *effect polymorphism* to give a useful type to functions like map

Continuation-Passing Style

A program is in CPS if *no function ever returns*

- Instead every function takes an extra argument (a function called "the continuation") that it *calls with the result*
- So an interpreter does *not* need a call-stack
- Every call is a tail call

Surprising part: There exists CPS transformations that take *any* λ-calculus program and produce an equivalent one in CPS.

- When translation-target runs, it builds closures that call other closures and this "list" is "where the call-stack went"
- A term of type $\tau_1 \rightarrow \tau_2$ translates to one of type $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_{ans}) \rightarrow \tau_{ans}$, i.e., a "foo returner" becomes a λ that takes a-λ-that-takes-a-"foo"-and-finishes-the-program and finishes-the-program.

Target language

We'll consider λ-calculus with addition and call variables "values" (for sake of translation, no effect on semantics)

Target of our translation ("programs in CPS"):

$$
\begin{array}{rcl}
e & ::= & v \mid v\ v \mid v\ v\ v \mid v\ (v + v) \\
v & ::= & x \mid c \mid \lambda x.\ e
\end{array}
$$

So we need no call-stack: at each step we either call a function (with 1 or 2 arguments) or add two constants.

- Theorem: Evaluation stays in this smaller language.

Now we just need a translation $C(e)$ from any λ-calculus term to something in this smaller language that is equivalent.

- Actually $C(e)\ (\lambda x.\ x)$ to "get started" since $C(e)$ will be a function that takes a continuation k and passes its result to k.

Here is one

Define translation $C(e)$ by mutual induction with $V(v)$ that helps translate values-and-variables.

$C(e)$ produces a function taking a continuation.

$$
\begin{aligned}
C(v) &= \lambda k.\ k\ V(v) \\
C(e_1 + e_2) &= \lambda k.\ C(e_1)\ (\lambda x_1.\ C(e_2)\ (\lambda x_2.\ k\ (x_1 + x_2))) \\
C(e_1\ e_2) &= \lambda k.\ C(e_1)\ (\lambda x_1.\ C(e_2)\ (\lambda x_2.\ x_1\ x_2\ k))
\end{aligned}
$$

$$
\begin{aligned}
V(c) &= c \\
V(x) &= x \\
V(\lambda x.\ e) &= \lambda x.\ \lambda k.\ C(e)\ k
\end{aligned}
$$

Note: This translation is pretty inefficient; fancier ones exist.

More on Continuations

This translation is important in theory and at the core of SML/NJ.

- Also advocated in many compiler "middle ends"
  - "Compiling with continuations" (Appel 80s, Kennedy 07)
  - Notice how every intermediate expression gets bound to a variable
- Makes implementing letcc and throw from lecture 10 easy and $O(1)$.
- A great way to think about and program web computations — encode continuation in URL to avoid server-side state and support the back-button.
  - I can point you to papers.

Overview

Review and highlights of what we did and did not do:

1. Semantics
2. Encodings
3. Language Features
4. Concurrency
5. Types
6. Metatheory

Review of Basic Concepts

Semantics matters!

We must reason about what software does and does not *do*, if implementations are *correct*, and if changes *preserve meaning*.

So we need a precise *meaning* for programs.

Do it once: Give a *semantics* for all programs in a language. (Infinite number, so use induction for syntax and semantics)

Real languages are big, so build a smaller model. Key simplifications:

- Abstract syntax
- Omitted language features

Danger: not considering related features at once

Operational Semantics

An *interpreter* can use *rewriting* to transform one program state to another one (or an immediate answer).

When our interpreter is written in the metalanguage of a judgment with inference rules, we have an "operational semantics".

This metalanguage is convenient (instantiating rule schemas), especially for proofs (induction on derivation height).

Omitted: Automated checking of judgments and proofs.

- Proofs by hand are wrong, especially for full languages.
- See Coq, Twelf, ...

Denotational Semantics

A *compiler* can give semantics as *translation* plus *semantics-of-target*.

If the target-language and meta-language are math, this is *denotational semantics*.

Can lead to elegant proofs, exploiting centuries of mathematics.

But building models is really hard!

Omitted: Denotation of while-loops (need recursion-theory), denotation of lambda-calculus (maps of environments, etc.)

Meaning-preserving translation is compiler-correctness.

Equivalence

With semantics plus "what is observable" we can determine equivalence.

In *security*, often more is observable than PLs assume.

- Because PLs want optimizations to be "correct"
- Because security is worried about "side channels"

In the real world, many languages have "implementation defined" features:

- C/C++ word-size, endianness, etc.
- Scheme evaluation order
- Java thread scheduling
- SML int size
- ...

Semantics Used?

- Standard ML has a small (few dozen pages) formal semantics.
- Caml has an implementation.
- Standards bodies write boat anchors.
- Some real-word successes, e.g., Wadler and XML queries, Manson and Java Memory Model, ...

Encodings

Our small models aren't so small if we can *encode* other features as derived forms.

Example: pairs in lambda-calculus, triples as pairs, ...

"Syntactic sugar" is a key concept in language-definition and language-implementation.

But special-cases are important too.

- Example: if-then-else in Caml.
- This is often a *design* question.

Language Features

We studied many features: assignment, loops, scope, higher-order functions, tuples, records, datatypes, references, threads, objects, constructors, multimethods, ...

We demonstrated some good *design principles*:

- Bindings should permit systematic renaming (α-conversion)
- Constructs should be first-class: permit abstraction and abbreviation using full power of language
- Constructs have intro and elim forms
- Eager vs. lazy (evaluation order, *thunking*)

Recall datatypes and classes support different flavors of extensibility.

- Omitted: work on better supporting both flavors (mixins, traits, open datatypes, EML, ...)

More on first-class

We didn't emphasize enough the convenience of first-class status: any construct can be passed to a function, stored in a data structure, etc.

Example: We can apply functions to computed arguments ($f(e)$ as opposed to $f(x)$). But in YFL, can you:

- Compute the function $e'(e)$
- Pass arguments of any type (e.g., other functions)
- Compute argument lists (cf. Java, Scheme, ML)
- Pass operators (e.g., +)
- Pass projections (e.g., `.l`)

1st-class allows parameterization; every language has limits

Omitted feature: Arrays

An array is a pretty simple feature we just never bothered with:

- introduction form: make-array function of a length and an initial value (or function for computing it)
- elimination forms: subscript and update, may get stuck (or cost the economy billions if it's C)

Why do languages have arrays and records?

- Arrays allow 1st-class lengths and index-expressions
- Records have fields with different types
- Hence some "very dynamic" languages like Ruby just have arrays

Nice to have the vocabulary we need!

Omitted feature: Exceptions

Semantics are pretty easy:

- One way: Use a stack of evaluation contexts; throw pops one off
- Another way: Compile away to sums (normal result or exception result) and put a match around every expression.

Typing is also easy: An exception throw can have any type (types describe the value produced by normal termination)

Omitted feature: Macros

We deemed syntax "uninteresting" only because the parsing problem is solved.

- Grammars admitting fast automated parsers an amazing success
- Gives rigorous technical reasons to despise deviations (e.g., `typedef` in C)

But syntax extensions (e.g., macros) are now understood as more than textual substitution

- Always was (strings, comments, etc.)
- Macro *hygiene* (related to capture) crucial, rare, and sometimes not what you want.
- Not a closed area

Omitted feature: Foreign-function calls

Language designers/implementors often guilty of "control the world syndrome".

Heterogeneity increasingly important and relying on byte-based I/O throws away everything we have been doing across langugage boundaries.

Omitted feature: Unification

Some languages do search for you using *unification*

```
append([], X, X)
append(cons(H,T), X, cons(H,Y)) :- append(T, X, Y)

append(cons(1,cons(2,null)), cons(3,null), Z)
append(W, cons(4,null), cons(5,cons(4,null)))
```

- More than one rule can apply (leads to search)
- Must instantiate rules with same terms for same variables.

Sound familiar? *Very* close connection with our meta-language of inference rules. Our "theory" can be a programming paradigm!

(See also the Alchemy project at UW for unification with probabilities.)

More omitted features: Haskell coolness

Some functional languages (most notably Haskell) have call-by-need semantics for function application.

Haskell is also purely functional, moving any effects (exceptions, I/O, references) to a layer above using something called *monads*. So at the core level, you *know* `(f x)*2` and `(f x)+(f x)` are equivalent.

Programming in a *monadic style* is useful for lots of things (but takes an hour to teach).

Haskell also has *type classes* which allow you to constrain type variables via "interfaces".

- Similar uses to bounded polymorphism and interfaces, but not based on subtyping.

Omitted features summary

I'm sure there are more:

1. Arrays
2. Macros
3. Exceptions
4. Foreign-function calls
5. Unification
6. Lazy evaluation (another name for call-by-need)
7. Monads
8. Type classes

Concurrency

Feels like "more than just more languages features" because it changes so many of your assumptions.

Omitted: *Process calculi* (e.g., π-caclulus) — "the lambda calculus of concurrent and distributed programming"

The hot thing: software transactions (`atomic : (unit->'a)->'a`)

- Lots of papers from the WASP group in the last couple years
  - Formal operational semantics, equivalence between them under appropriate effect systems
  - Prototype implementations and optimizations for ML, Scheme, Java
  - My favorite analogy

Types

- A type system can prevent bad operations (so safe implementations need not include run-time checks)
- I program fast in ML by relying on type-checking
- Deep connection to logic
- "Getting stuck" is undecidable so decidable type systems rule out good programs (to be *sound* rather than *complete*)
  - May need new language constructs (e.g., fix in STLC)
  - May require code duplication (hence polymorphism)
  - A balancing act to avoid the Pascal-array debacle

Safety = Preservation + Progress (an invariant is preserved and if the invariant holds you're not stuck) is a general phenomenon.

Just an approximation

There are other approaches to describing/checking decidable properties of programs:

- Dataflow analysis (plus: more convenient for flow-sensitive, minus: less convenient for higher-order); see 501
- Abstract interpretation (plus: defined very generally, minus: defined very generally)
- Model-checking (a course in itself 3 years ago)

Zealots of each approach (including types) emphasize they're more general than the others.

Polymorphism

If every term has one simple type, you have to duplicate too much code (can't write a list-library).

Subtyping allows subsumption. A subtyping rule that makes a safe language unsafe is wrong.

Type variables allow an incomparable amount of power. They also let us encode strong-abstractions, the end-goal of modularity and security.

Ad-hoc polymorphism (static-overloading) saves some keystrokes.

Inference

Real languages allow you to omit more type information than our formal typed languages.

Inference is elegant for some languages, impossible for others.

- Not a closed area (e.g., Generalized Abstract Data Types)

But the error messages are often bad because a small error may cause a type problem "far away".

- That's why Ben Lerner and I did "Seminal"

Metatheory

We studied many properties of our models, especially typed $\boldsymbol{\lambda}$-calculi: safety, termination, parametricity, erasure

Remember to be clear about what the question is!

Example: Erasure... Given the typed language, the untyped language, and the *erase* meta-function, do erasure and evaluation commute?

Example: Subtyping decidable... Given a collection of inference rules for $\boldsymbol{\Delta} \vdash \boldsymbol{\tau_1} \leq \boldsymbol{\tau_2}$, does there exist an *algorithm* to decide (for all) $\boldsymbol{\Delta}$, $\boldsymbol{\tau_1}$ and $\boldsymbol{\tau_2}$ whether a derivation of $\boldsymbol{\Delta} \vdash \boldsymbol{\tau_1} \leq \boldsymbol{\tau_2}$ exists?

Last Slide

- Languages and models of them follow guiding principles
- Now you can't say I didn't show you continuation-passing style
- We can apply this stuff to make software better!!

Defining program behavior is a key obligation of computer science. Proving programs do not do "bad things" (e.g., violate safety) is a "simpler" undecidable problem.

- A necessary condition for modularity
- Hard work (subtle interactions demand careful reasoning)
- Fun (get to write compilers and prove theorems)

You might have a PL issue in the next few years... I'm in CSE556.