CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2009

Lecture 19— Course Summary and Wrap-Up

Last lecture

Would like to focus on:

- The “big ideas” we covered
- How the parts of the course fit together
- Revisiting the goals and motivation from lecture 1

To be honest, studying for the final is much more about getting up to speed on details and particular techniques

- This lecture is not aimed as much at the next 4 days
- Yet the big picture can help organize your thoughts

More about final exam

- Monday December 14, 10:30AM-12:20PM
- Intended to test the material since the midterm (lectures 10–19 and homeworks 3–5), but obviously material accumulates
- Old exams and reference-pages posted
- You can bring your own reference sheet
- I think it's on the long and difficult side, but that's okay

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

Computer Science: Building abstract models is "what we do" — the models in PL have a particular flavor

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 the 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", so specification is weaker
- Because security is worried about "side channels"

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

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

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*)

Most things boil down to scope, levels of indirection, and eagerness

- Exactly what models like $\boldsymbol{\lambda}$-calculus focus on

Many more features

We have aimed toward the principles:

- Example: Typing rules should have sound logical interpretations under the Curry-Howard Isomorphism
- Example: Soundness and completeness with respect to a policy like “don’t get stuck”

Rather than an exhaustive march through language features:

Examples we could do in 5-60 minutes each: Arrays, macros, exceptions, foreign-function calls, monads, type classes

Other paradigms: Unification, i.e., logic programming, is like how our inference rules work

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, all now readable by you
  - Formal operational semantics, equivalence between them under appropriate effect systems
  - Prototype implementations and optimizations for ML, Scheme, Java
  - My favorite analogy: "The Transactional Memory / Garbage Collection Analogy" [OOPSLA 2007]

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 and/or 503
- 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).
- But just as important and probably more interesting is using polymorphism to enforce abstractions.
- Subtype polymorphism is based on subsumption. A subtyping rule that makes a safe language unsafe is wrong.
- Parametric polymorphism is based on type variables. It has incomparable benefits to subtyping.

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!

- Another one of those research lessons that transcends PL

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 $\vdash \boldsymbol{\tau_1} \leq \boldsymbol{\tau_2}$, does there exist an *algorithm* to decide for all $\boldsymbol{\tau_1}$ and $\boldsymbol{\tau_2}$ whether a derivation of $\vdash \boldsymbol{\tau_1} \leq \boldsymbol{\tau_2}$ exists?

From Lecture 1: Is this Really about PL?

Building a rigorous and precise model is a hallmark of quality research.

The value of a model is in its:

- fidelity
- convenience for establishing (proving) properties
- revealing alternatives and design decisions
- ability to concisely communicate ideas

Why we mostly do it for programming languages:

- Elegant things we all use
- Remarkably complicated (need rigor)

But I deeply believe this "theory" makes you a better CSE researcher

- Focus on the model-building, not just the PL features

Last Slide

- Defining program behavior is a key obligation of computer science
- Languages and models of them follow guiding principles
- We can apply this stuff to make software better
- And it's fun (glad I'm not a dog)

10 weeks is only enough time to scratch the surface

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

You have grading to do

I am going to distribute course evaluation forms so you may rate the quality of this course. Your participation is voluntary, and you may omit specific items if you wish. To ensure confidentiality, do not write your name on the forms. There is a possibility your handwriting on the yellow written comment sheet will be recognizable; however, I will not see the results of this evaluation until after the quarter is over and you have received your grades. Please be sure to use a No. 2 PENCIL ONLY on the scannable form.

I have chosen *(name)* to distribute and collect the forms. When you are finished, he/she will collect the forms, put them into an envelope and mail them to the Office of Educational Assessment. If there are no questions, I will leave the room and not return until all the questionnaires have been finished and collected. Thank you for your participation.