CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2005

Lecture 18

A Biased Pocket-Guide to the PL Universe

79.5 Minutes of PL left

- Review and highlights of what we did and did not do:
  1. Semantics
  2. Encodings
  3. Language Features
  4. Types
  5. Metatheory
- What the WASP group has been up to that relates to 505

Actually, the field is at least half “language implementation” but that’s 501 not 505. (If you want to know how compilers deal with something, come ask me.)

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 a 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.
- Proofs about ML programs are too hard.
- See Twelf, Coq, ...
- (informal reading group next quarter; talk to Marius)

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 work, and treating code as math is "the right thing to do".

But building models is really hard!

Omitted: Denotation of while-loops (need recursion-theory), denotation of lambda-calculus (maps of environments? can avoid recursion in typed setting).

Meaning-preserving translation is compiler-correctness...

Rhodium in one slide

Again, compiler-correctness is meaning-preserving translation.

For optimization, source and target are same language.

If an optimization:

- is written in a restricted meta-language
- uses a trusted engine for computing what's legal
- is connected to the semantics of the language

Then an automated theorem prover can show "once and for all" that an optimization is *correct*.

Sorin will defend next month and go to UCSD.

Erika is making the optimizations easier to write by automatically generating the "boring cases".

Other Semantics

- Axiomatic Semantics: A program is a query-engine. Keywords: weakest-preconditions, Hoare triples, ... A program *means* what you can prove about it.
- Game Semantics: A program is its interaction with the context. To "win", it "produces an answer". (Less mature idea; seems useful for dealing with the all-important context.)

Useful?

- Standard ML has an impressive, small (few dozen pages) formal semantics.
- Caml has an implementation.
- Standards bodies write boat anchors.
- Recent success: Wadler and XML queries.

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.
- I do not know how to answer this *design* question.

Omitted: Church numerals, equivalence proofs, etc.

Fancy encoding: Continuation-Passing Style

There is a smaller λ-calculus:

$$e ::= x \mid \lambda x.\, e \mid e\; y \mid e\; (\lambda x.\, e') \mid e\; c$$

That is, the right-side of an application is always a value or variable (i.e., computed in O(1) time and space).

Evaluation stays in this (sub)language!

So inductively we don't need a stack (every decurried call is a tail-call)!

And there's a translation from the full λ-calculus to this sublanguage!!

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" becomes a λ that (takes a λ that takes a "foo" and finishes-the-program) and finishes-the-program.

At the end of the day, e and $(translate(e))(\lambda x.\, x)$ are equivalent.

That word!

The definition of $\boldsymbol{translate}(e)$ is short but mind-bending.

We have seen continuations (homework 4) and continuation-passing style (last slide).

If our source language has letcc and throw, we can extend $\boldsymbol{translate}$ even though the target language doesn't!

And letcc and throw are O(1) operations!

Roughly, the translation is turning every context into a $\boldsymbol{\lambda}$, so:

- We don't need a stack (there's a $\boldsymbol{\lambda}$ that encodes it)
- Letcc just binds the right $\boldsymbol{\lambda}$ to the right variable
- Throw just calls a different $\boldsymbol{\lambda}$ than we otherwise would

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

Language Features

We studied many features: assignment, loops, scope, higher-order functions, tuples, records, variants, first-class references, exceptions, 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*)

We saw datatypes and classes better support different flavors of extensibility. Keunwoo's thesis is on supporting both in the presence of *parameterized modules*.

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

Nice to have the vocabulary we need!

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

Semi-omitted feature: Threads

On the homework we investigated:

- Threads and locks formally
- join and condition variables informally

Concurrent programs have two common sources of errors sequential programs don't:

- Races (unsynchronized access to shared data)
- Deadlock (threads stuck waiting for locks)

For the last 10 years or so, much exciting work on PL tools (e.g., type systems) to reduce these errors.

Now many of us are excited about *atomicity* (others like Peter-Michael are excited about *chords*).

Atomicity

Atomic is an easier-to-use and harder-to-implement concurrency primitive:

```
atomic : (unit->'a)->'a
```

Execute its argument as though no other thread has interleaved execution, but maintain fair scheduling.

Mike Ringenburg: Caml with atomicity, particularly efficient because Caml has little mutation and runs only one thread at a time

Ben Hindman: Java with atomicity, via source-to-source translation (with lots of fancy OO tricks), performance will hopefully be tolerable

(Lots of work left to do on the latter.)

Omitted: 15 minutes on why atomic is better than locks.

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!

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.

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

Omitted features summary

I'm sure there are more:

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

Types

(You should know I'm called a "types person")

- A type system can prevent bad operations (so safe implementations need not include run-time checks)
- I program fast in ML because I rely on type-checking
- "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 Chambers, Winter 06 and Nathan's work on a "flow engine"
- Abstract interpretation (plus: defined very generally, minus: defined very generally)
- Model-checking see special course, Qadeer (MSR), Spring 06

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

Types as "abstract interpretation" example: (3) = int (4) = int
(+) = fun x,y. if x=int and y=int then int else fail
Typechecks if abstract-interpretation does not produce "fail"

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.

Note: In C, casts (subsumptions) are often "correct" only under certain architectural assumptions. Marius is going to analyze your C code and tell you what those assumption are.

Inference

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

Inference is elegant for some languages, impossible for others.

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

Ben Lerner: Can we use *search* to find a "nearby" program that does typecheck and show you the difference?

Metatheory

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

Every type system we come up with corresponds to a logic and vice-versa! (*Constructive* logic (no excluded middle) essential to computation.)

Remember to be clear about what the question is!

Example: Erasure... Given the typed language, the untyped language, and the erase meta-function, does 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?

Other models

We studied two models in depth: IMP (intraprocedural manipulation of global variables) and lambda-calculus (lexically-scoped higher-order functions).

There are good newer core models for other paradigms:

- $\boldsymbol{\pi}$-calculus for communicating processes
  - There are only channels (to send and recieve) and processes
  - More primitive than $\boldsymbol{\lambda}$ because application becomes one send and one receive
- $\boldsymbol{\sigma}$-calculus for objects (late-binding)
- Also decades on denotational models of $\boldsymbol{\lambda}$-calculi (terms are math functions over environments)

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 5 years... I'm in CSE556.