CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

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 (Semantics, Encodings, Language Features, Types, Metatheory)
- What I've been doing for the last 5 years (applying this stuff to low-level code)

A "fair but biased" view of the field
(whatever that means)

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 small-step operational semantics (or large-step).

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 (or other theorem provers), TinkerType, ...

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

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. It is a game it "wins" when 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. O'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 OCaml.

Omitted: Church numerals, equivalence proofs, etc.

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)
- Constructs have intro and elim forms
- Eager vs. lazy (evaluation order, *thunking*)

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., `.1`)

1st-class allows parameterization; every language has limits

More language features

There are many features we did not examine carefully...

Arrays:

- 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!

Back to syntax

We said syntax was "uninteresting" but that's mostly because it's a solved problem.

- Grammars admitting efficient automated parsers an amazing success
- Gives rigorous technical reasons to despise deviations (e.g., `typedef` in C)
- Syntax extensions (e.g., macros) 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 completely closed area (good recent papers)

Control operators

We did little interesting control-flow:

- Intraprocedural break, goto, etc. not hard but complicates formalism
- Exceptions are interesting but can be explained as "bubble-up an option"
- *First-class continuations* are extremely powerful and mind-bending

```
(+ 3 (call/cc (lambda (k) (+ 4 (k 2))))) ; produces 5
```

But it's first-class and can escape!

Stare at this for fun!

```
(let* ((x 0)
       (y #t)
       (z (+ 3 (call/cc (lambda (k)
                          (begin (set! x k) 4))))))
   (if y
       (begin (set! y #f) (x 5))
       z))
```

Seriously useful for backtracking computation, coroutines (think two communicating stacks; very painful to code up), etc.

Sometimes called the "goto of functional languages" (extremely elegant, easy to misuse)

(The result is 8)

Continuation-Passing Style

A continuation represents the "rest of computation" much like a "return address" in assembly code.

In fact, you can translate a $\boldsymbol{\lambda}$-term into an equivalent $\boldsymbol{\lambda}$-term that always invokes explicit continuations.

A term of type $\tau_1 \longrightarrow \tau_2$ translates to one of type $\tau_1 \longrightarrow (\tau_2 \longrightarrow \tau_{ans}) \longrightarrow \tau_{ans}$

The target of the translation is an even smaller language, e.g., all applications are to variables.

Some functional-language compilers actually do this to enable optimizations and/or make call/cc cheaper.

It's also very important in theory (a smaller language)

Threads

Except for one homework problem, we ignored parallel or concurrent execution. Key questions:

- What can happen when?
- When can a thread be interrupted or killed?
- How do threads communicate and synchronize?
- How (and how many) threads are created?

New programming errors: races and deadlocks
New optimization opportunities: scheduling, lock removal

Language support for catching errors a research development in the last 5 years (still hot)

Logic Programming

Some languages do search for you using *unification*

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

- 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!

Constructs Summary

We could have spent a week or more on each of macros, continuations, CPS, threads, coroutines, and unification.

Even in languages with just functions, objects, assignment, and exceptions, these other constructs help you design and can sometimes be painfully coded up.

Conceptually, we have a term language with a (rich) set of well-defined constructs and *then* we consider a type system for eliminating (everything but) a well-defined set of programs...

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.
  - 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

Just an approximation

There are other approaches to describing decidable properties of programs:

- Dataflow analysis (plus: more convenient for flow-sensitive, minus: less convenient for higher-order)
- Abstract interpretation (plus: defined very generally, minus: defined very generally)

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.

Metatheory

We studied many properties of our models, especially typed λ-calculi: safety, termination, parametricity, erasure

We mentioned inference, which for ML is clever and brittle.

The coolest thing we didn't do is the Curry-Howard isomorphism... In System F, there's no term of type $\alpha \to \beta$, but there is one of $\alpha \to (\alpha \to \beta) \to \beta$.

Look at the typing rules for products (and) and sums (or)! Another reason (besides termination) we had to add fix.

"Programs are proofs" "Types are propositions"

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

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:

- π-calculus for communicating processes
  - There are only channels (to send and recieve) and processes
  - More primitive than λ because application becomes one send and one receive
- σ-calculus for objects (late-binding)
- Also decades on denotational models of λ-calculi (terms are math functions over environments)

Safety is my business

All else equal, safe language are better than unsafe ones. (Java finally caught on after 15 years of ML, 20 of Scheme, 45 of Lisp)

But assembly languages and C are easier to implement, easier to reason about space and time, and (still) the de facto standard for systems level programming and software distribution.

A safe assembly language could allow untrusted, unsigned code!

A safe C-like language could let you write (most of) your low-level system with the same guarantee as a HLL.

But a C-like language cannot (just) use HLL techniques like implicit indirection and garbage collection.

Shameless plugs

Did you know there's a research language that uses a fancy type system and flow analysis to ensure safety of C-like programs while still allowing some manual memory management, flat data, etc.? (I'm still working on it; real languages are hard!)

Did you know there are many language-based techniques for finding bugs in C or C-like programs?

Automation is key; we are drowning in C code.

Almost everything we try finds bugs; can we find them all? The most important ones? Before the hackers do?

Take 590DG – this is a hot area about to move past "I found a lot of bugs too"

Last Slide

- Languages and models of them follow guiding principles
- Now you can't say I didn't show you continuations or Curry-Howard
- We can apply this beautiful stuff to ugly languages

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.