Programming Languages Lecture 13

Wrapup

Allyn Dimock

U. Mass. Lowell

Today’s Material

- What relevant skills did you come in with
- What skills do you leave with
- How does material fit together
- What didn’t we cover
- Where to go from here
- What didn’t we cover that might be on the qualifier.

Incoming skill set

- Imperative programming in C
- Discrete math, or 91.500, or 91.502
  - Basic proof skills: propositional calculus, induction.
- 91.301 (or equivalent)
  - Scheme: special forms vs. application, expressions
  - Functional programming, accumulators
  - Experience writing / modifying some interpreter: environments, values
  - Some operational semantics: substitution model, environment model.

Current skill set

- Ability to learn a new language in a week
- Imperative, Functional, Logic, and OO programming
- Extensive experience with Standard ML
- Basics: variables, scope, parameter passing, types systems, data abstraction
- Fundamental math: formal proof, BNF, $\lambda$-calc, F.O.L.
- Denotational and operational semantics
- Interpreters
- Language design ideas
Details: programming paradigms

Imperative programming — ImpCore
- sequencing, loops, functions

Functional programming — μScheme, SML, μML
- first-class functions composing, currying, folding

Logic programming — Prolog
- clausal form, unification, resolution, search

Object-oriented programming — μSmalltalk
- message passing, classes and objects, inheritance

Details: Standard ML

typed functional programming
- datatypes
- patterns
- exceptions
- (module system)

Details: Basics

Variables
- imperative: assignment, l-values vs. r-values, environments and stores
- functional: binding
- logic: unification, partially-specified data

Scope: static vs. dynamic

Parameter passing: call-by-value vs. call-by-reference vs. substitution

Type systems: simply-typed, parametric polymorphism, subtype polymorphism, type checking, Hindley-Milner type inference, subtyping

Data abstraction: abstract data types, objects, (modules)

Details: Fundamental math

Axioms and rules, formal proof

BNF representation of context-free grammars, abstract syntax

λ-calculus: substitution, β-reduction, combinators, reduction strategies: outermost (normal order) vs. innermost, βν-reduction, encoding data as functions

First-order logic: its denotational semantics, interpretations, satisfaction and validity, theories, conversion to clausal form

Partial orders, complete partial orders, and domains
Details: Semantics

- Operational semantics
  - evaluation (big-step), small-step, values.
    - environments and stores.
- Denotational semantics
  - compositionality, example semantics of numerals,
    regular expressions, imperative language features,
    Scheme subset. Using continuation passing style to
    specify order of computation.
- Interpreters
  - definitional interpreters, metacircular interpreters,
    interpreters from operational semantics, interpreters
    from denotational semantics: "concrete semantics".

Details: Correctness

- Correctness:
  - inductive proofs of correctness of recursive
    functional programs.
  - bisimulation and homomorphism proofs of
    correctness of abstract data types.

Details: Language design ideas

- An underlying model of computation
- Influence of
  - Expressive power
  - Methodology
  - Implementation
- Interpreters
- What can you name, what can you parameterize
- Correctness counts

Language design ideas

- Underlying model: Von Neumann machines, various
  logics such as lambda-calculus, F.O.L., others...
- Expressive power
  - Formal: Turing complete? (all this semester)
  - Less formal: What is easy to express?
  - What compromises to allow multiple paradigms? (All
    languages this semester admit imperative
    programming.)
- Methodology
  - Imperative / Functional / Logic / OO / (Concurrent)
  - Structured control flow / Data abstraction / Code
    reuse
Language design ideas II

- Implementation
  - Does language need to run efficiently
  - Scripting languages vs. production programming
  - Compromises: performance vs. correctness
  - Reference implementations

- Interpreters
  - Quick testing of ideas.
  - Metacircular interpreters allow absorbing the features that you are not studying
  - Reference implementations: interpreters
    - from operational semantics
    - from (concrete) denotational semantics

Language design ideas III

- Correctness counts:
  - Verbal descriptions are open to misinterpretation
  - Formalisms are necessary.
  - Write your code once and it should run on all compilers / interpreters.

Language design example: ImpCore

- Model: Von Neumann machine.
- Methodology: Imperative programming.
- Efficiency: Easy to achieve.
- Abstraction and parameterization:
  - Abstract locations as variables.
  - Replace gotos in model with conditionals and loops.
  - Abstract code sequences as functions
  - Parameterize code sequences as functions taking arguments. Choose call-by-value argument passing.
- Choose function body or top level as only scopes.
  - Choose not to support nested functions.

How does it all fit together?
How does it all fit together

- Formalisms
  - Denotational semantics is basic: show correctness of operational semantics, (and Hoare logics).

- Type Systems
  - Fail-stop behavior
  - Static type systems: "A well-typed program can not go wrong"
    - Progress
    - Subject reduction
  - Semantics of statically-typed programming languages defined only for well-typed programs
  - Static type systems exclude some programs that would always run correctly

How does it all fit together II

- Understanding languages in multiple ways
  - Semantics to say what is computed
  - Interpreters
  - Semantics as code
    - Opportunity to try new features
  - Look at tiny fragment of language: $\lambda$-calculus – functional; Horn-clause formulae – logic; the “while-loop” language – imperative; Abadi-Cardelli object calculus, or Pierce et. al. miniJava – OO. Pi-calculus, mobile ambients – concurrent...
  - Proofs of correctness of
    - Programs
  - Language definitions (consistent)

What didn’t we cover?

- I/O: semantics of observable behavior (+ final value)
- Nondeterminism: multiple possible values
- Concurrency: a course in itself
- High-performance implementations: a course in itself (91.534)
- Much of the math: a course in itself (91.538)
- More complex uses of semantics: (also 91.538)
- Languages based on different models: set theory, logics with equality
- Program design techniques (OO Patterns, Components, Aspect-Oriented).
- Abstracting away exact execution: Use the math of denotational semantics to prove properties of programs.
Where to go from here?

In Industry
- Apply your ability to learn new languages
  - Is the language the right match for the problem?
    (Q: when is C the right match?)
- Pick up a new scripting language
  (Ruby – Smalltalk, Scsh – Scheme, ...)
- Write interpreters: "Inside any big program there is a small language trying to get out"

U. Mass. Lowell
- The language qualifying exam for the PhD program
- Specific courses:
  91.534 (Compilers), 91.538 (Semantics), 91.540...
- Independent study

Where to go from here II

Resources for continued interest
- Consider going to NEPLS (New England Programming Languages Seminar)
  next meeting June 24, Williams College
- Check out colloquia
- Seminar series at Northeastern, Boston University.

Overlap
- Languages for graphics or guis
- Languages for managing high security data
- Languages for database querying / manipulation
- Language for modeling cell biology
- Language for specifying combinatorial search algorithms

Notes

Administrative:
Do not give answers to other students. Some problems, mostly elementary ones, will be reused next semester. Advanced problems will generally change. Result: a student seeing your answers will not do well in the course. (Not to mention academic policy).
What we missed on the qualifying exam

Hoare logic, module systems.

http://www.cs.uml.edu/~wang/quals/SampleExam.htm