Programming Language Theory

Summary and Conclusion
Final Exam and Course Evaluations

- **Final Exam**
  - Available from Mon. Dec. 7 @ 12:00AM through Tue. Dec. 8 @ 12:00PM, with a 2.5hr time limit.
  - Will use a (dummy/single question) myCourses Quiz to distribute the link to the final exam PDF and to time stamp the start/end of an individual’s exam.
  - Encouraged to use the official exam time: Mon. Dec. 7, 1:30PM–4:00PM; Prof. Fluet will definitely be available via Zoom during this time, and also Mon. Dec. 7, 10:00AM–12:30PM.

- Emphasis on material since Midterm Exam 2
  - Weeks 10-15; Homeworks 5–7
  - (but, obviously, material accumulates)

- open book, open notes
  - if necessary, appendix with essentials provided
Course in a nutshell

Highlights of what we did (and did not) do:

1. Abstract Syntax
2. Operational Semantics
3. Encodings
4. Types, Types, and more Types
5. Metatheory
6. Concurrency and Shared Memory and Message Passing
Fundamental Principle: **Semantics matters!**

Reason about what software does and does not *do*. Need a precise *meaning* for programs.

Do it once: Give a *semantics* for all programs in a language.

- (infinite number of progs, so use induction for syntax and semantics)

Real languages are big, so build a smaller model.

Key simplifications:

- Abstract syntax
- Omit 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 give semantics: as *rewrites* that transform one program state to another (or to an immediate answer).

*Operational semantics*: when interpreter is written in the metalanguage of a judgement with inference rules.

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

Omitted: Automated checking of judgements, derivations, and proofs

▶ Proofs by hand are wrong, especially for full languages

▶ See NuPRL, MetaPRL, Twelf, Isabelle/HOL, Coq, Agda, LEAN, . . .
Denotational Semantics

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

*Denotational semantics*: when the target-language and meta-language are mathematics.

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”, 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”

Many “real world” languages have “implementation defined” features:

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

If we can encode other features as derived forms, then our small models aren’t so small.

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

“Syntactic sugar”

▶ a key concept in language-definition and language-implementation

But special-cases are important too

▶ Example: if-then-else in SML
▶ This is often a design question
Language Features

We studied *many* features:

- assignment, loops, scope, higher-order functions, tuples, records, subtyping, datatypes, references, threads, . . .

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 λ-calculus focus on
Language Features

We have aimed toward the principles. Examples:

- Typing rules with sound logical interpretations under the Curry-Howard Isomorphism
- Soundness and completeness with respect to a policy (like “don’t get stuck”).

We have avoided an exhaustive march through language features. Examples we could do in 5-60 minutes each:

- arrays, macros, exceptions, foreign-function calls, monads, type classes
Types

- A type system can prevent bad operations
  - (so safe implementations need not include run-time checks)
- 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)

Safety = Progress + Preservation

- Progress: if the invariant holds, then you’re not stuck
- Preservation: the invariant is preserved
Type Systems: (just) an approximation

Other approaches to describing/checking decidable properties of programs:

▶ Dataflow analysis
  ▶ plus: more convenient for flow-sensitive properties
  ▶ minus: less convenient for higher-order

▶ Abstract interpretation
  ▶ plus: defined very generally
  ▶ minus: defined very generally

▶ Model-checking

Beware: zealots of each approach (including type systems) emphasize they’re more general than the others.
Polymorphism

- If every term has one simple type, then you have to duplicate too much code.
  - (e.g., 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 $\lambda$-calculi:

▶ safety, termination, parametricity, erasure

Remember to be clear about what the question is!

▶ Another one of those lessons that transcends PL

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

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

Feels like “more than just more languages features” because it changes so many of your assumptions.
▶ (e.g., whether or not two expressions are equivalent)

Omitted: *Process calculi* (e.g., $\pi$-calculus)
▶ “the lambda calculus of concurrent and distributed programming”

The hot thing: software transactions
(val atomic : (unit->'a)->'a)
The elegant thing: synchronous events
(val sync : 'a event -> 'a)
Why study programming languages this way?  (from Lecture 1)

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

The value of a model is in its:

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

Why we mostly do it for programming languages:

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

But this “theory” makes you a better computer scientist

- focus on the model-building (applicable everywhere), not just the PL features
The End

- 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!

15 weeks is only enough time to scratch the surface, but . . .

- You are more prepared than you think to tackle interesting papers
- My door is always open to talk PL!