Greg Morrisett

CS252r: Advanced Topics in Programming Languages
Goal

Understand how to build models of dynamic systems and *formally* reason about them.
Dynamic Systems

(Possibly infinite) set of states and a transition relation between states.
Examples

- DFAs and NDFAs
- Turing machines
- Programming languages
- Protocols
- Operating systems
- Robots
- Financial market
- Bio-molecular pathways
- ...

...
Construct *machine-checkable* proofs of properties about the behavior of systems.
Examples

- Properties on the dynamics
  - “no type error”
  - “no missing cases”
  - “no path that does X”
  - “every path terminates”
  - “no information leaked”

- Translations between systems
  - Compilers, interpreters, reductions, models
  - Preserves some relation on the dynamics
    - e.g., bisimulation, refinement
Formal methods (i.e., math) today:
- Modeling & reasoning happens with pen & paper.
- Good for “deep” properties of simple systems.
- But bad for large systems with complex interactions.
  - c.f., *The Definition of Standard ML*
  - Physics reaction: reduce to a simpler model
  - CS reaction: automate it!
Ideally

- Write the code for your system in a scalable programming language.
  - Features for abstraction, re-use, re-factoring, representation changes, synthesis, etc.
  - i.e., Software Engineering --- always in flux.
- Write the models and proofs in a scalable environment.
  - Features for abstraction, re-use, re-factoring, etc.
CompCert compiler (X.Leroy)

- Compiler that maps C to PPC, X86, ARM
  - A little more than what’s done in 153
  - e.g., like GCC –O1

- Proof that the compiler is extensionally correct.
  - “If source program S has input/output behavior X, then the target program T has behavior X.”
  - Note: must build a model (think interpreter) for C code and machine code so we can relate their behaviors.
Another Example

- SEL4 Micro-Kernel (NICTA)
  - Based on L4 Micro-kernel, written in C
  - Built model of behavior in (subset of) Haskell
  - Proved correspondence of behavior between the C code and the Haskell code.
  - Proved properties about the Haskell code.
    - E.g., no kernel panic
Not really scalable yet.
- For CompCert, the models and proofs were an order of magnitude bigger than the code for the compiler.
  - Though models are very re-usable.
- For SEL4, took something like 20 man years.
- So this is still an active area of research...
  - From “formal methods” to “proof engineering”
Lay the ground work for proof engineering.

How to model the behavior of dynamic systems in a scalable fashion?

- Need structured, compositional ways to assemble descriptions of the “state” of the system, as well as the transitions.
- Need proof techniques for reasoning about states and transitions.
- Gotta walk before you can run...
Arithmetic expressions
\[ e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \]

Boolean expressions
\[ b ::= true \mid false \mid e_1 = e_2 \mid e_1 \leq e_2 \]

Commands
\[ c ::= \text{skip} \mid x := e \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

Memory
\[ m : \text{Id} \rightarrow \mathbb{Z} \]

States
\[ S ::= <m,c> \]

Transitions
\[ S_1 \rightarrow S_2 \]
Kinds of Properties

- Program-level
  - e.g., this command computes n factorial and places the result in x.
  - e.g., this command is equivalent to that command
- Language
  - e.g., for all c1 c2 c3, (c1 ; c2) ; c3 is equivalent to c1 ; (c2 ; c3).
  - e.g., for all c, optimize(c) is equivalent to c.
  - e.g., for all c, analyze(c) = true implies c’s behavior will respect policy P.
Three major kinds of models:

- **Operational models**
  - Think interpreter written in math
  - Easiest to write & scale to full languages
  - Hard to reason about

- **Denotational models**
  - Think compiler to set theory (or variants thereof)
  - Tricky problems (recursion, concurrency, ...)
  - Good for reasoning equationally, compositionally
  - Usually not good for intensional details (e.g., timing, space, etc.)

- **Axiomatic models**
  - Think compiler to logic
  - Similar scaling problems with denotational methods
  - Good for logical reasoning, not just equational.
  - Provides built-in support for abstraction.
Example: Regular Expressions

- Operational models: DFA
  - Easy to write an interpreter.
  - Hard to prove one graph accepts the same language as another one.

- Denotational models: regexps
  - Not so easy to build an interpreter
  - Simple algebraic rules for reasoning about equivalence, containment, etc.
Move on to more “sophisticated” linguistic constructs:
- Procedures & h.o. functions
- Data structures: products and sums
- First-class mutable data
- Control: exceptions, continuations, threads, streams

Organized by types
- Simple types
- Polymorphism
- Dependency
Big Proof Hammers

- Induction
- Co-Induction
- Types
- Logical Relations
- Program logics
Use the Coq proof assistant.

- Unified way of writing code, models, theorems, and proofs.
- Good news: Coq will tell you when you have a proof.
- Bad news: Coq won’t be much help in writing the proofs --- anymore than your compiler is a help in writing your source code.

Analogy: difference between algorithm & code

“The difference between theory and practice is greater in practice than in theory.”
Logistics

- [www.eecs.harvard.edu/~greg/cs252rfa09/](http://www.eecs.harvard.edu/~greg/cs252rfa09/)
  - Perhaps to be moved to cs252rfa11
- Class: MWF 3-4pm (usually)
  - Only moderately structured
    - I’ll go where you want/need me to go
  - Bring laptop with Coq loaded
  - Be prepared for “code/proof” reviews
  - Participation counts a lot
- Homework:
  - Daily/weekly assignments in Coq
  - Either it compiles or it doesn’t...
  - Must keep up!
Sources

- *Software Foundations (SF)* by B.C. Pierce et al.
- *The Formal Semantics of Programming Languages: An Introduction* (Winskel) by G. Winskel
- *Types and Programming Languages (TAPL)* by B.C. Pierce.
- *Certified Programming with Dependent Types (CPDT)* by A. Chlipala.
Who Should[not] Take This?

- Need at least CS51 & 121
  - Comfortable with functional programming
  - Basics of logic, discrete math
  - Some computability, automata theory, etc.
- Big overlap with CS152 (PL)
  - Similar kinds of topics
  - But degree of formalization is more intense
  - Might consider Chlipala’s MIT class instead