CMSC 631 – Program Analysis and Understanding
Fall 2003
New Directions

Program Analysis and Understanding
• This semester, we’ve covered a lot of material about programs and programming languages
• Main areas of static program analysis:
  ■ Data flow analysis
  ■ Abstract interpretation
  ■ Type systems
  ■ Theorem Proving
  ■ Model checking
• Today: An assortment of things we didn’t cover

The Space of Static Analyses
• Are these four really different?
  ■ What is the connection between them?

Type Systems and Theorem Proving
• Type systems are really dumb theorem provers
  ■ Only theorems ... running time hard to predict
  ■ Hard to understand what works and what doesn’t
  - Encoded in the decision procedure

Proof-Carrying Code (Necula and Lee)
• How do you know a program is safe to run?
  ■ It’s pretty difficult to decide given just the code!
    - Coming up with the proof of correctness is hard
  ■ Idea: code comes with a safety proof
    - You just check the proof, which is often “easy”

  • Applications:
    ■ Mobile/distributed code
    ■ Compilers (do you trust gcc?)

Types and PCC
• So what are these proofs?
  ■ In theory, any thing you like
  ■ In applications so far, type (and memory) safety!
• But...type systems are easy theorem provers
  ■ Except we need type safety proofs for executable code
  ■ Translation from high-level source obscures details
  ■ Enter typed assembly language (Morrisett et al)
  ■ Bring types all the way through the compiler
Data Flow and Model Checking

- Schmidt, “Data Flow Analysis is Model Checking of Abstract Interpretations.” POPL98.
  - State space: Program execution tree
    - Each conditional branch is a fork in the tree
- Consider very-busy expressions:
  \[ VBE(p) = \text{Used}(p) \cup (\text{notMod}(p) \cap (\bigcap_{p < \text{succ}} VBE(p'))) \]
- Reformatted as model checking the exec. space:
  \[ \text{isVBE}(e) = \nu Z. \text{isUsed}(e) \lor (\text{notMod}(e) \land \Box Z) \]
  (here \( \nu \) is the greatest fixpoint operator)

Model Checking and Theorem Proving

- Model checkers are fully automated theorem provers
  - Again, they prove “dumb” theorems
  - But somewhat smarter than type systems
    - E.g., they handle concurrency, complicated properties
  - But don’t do a good job with complex structures
    - E.g., functions, data structures

Model Checking and Type Systems

- Naik and Palsberg, “A type system equivalent to a model checker”
  - (not yet published; see Palsberg’s web page)
  - Shows how to construct a type system that accepts exactly the set of programs that a model checker passes

Optimizing Compilers

- Traditional target of data flow analysis
  - (We’ve seen lots of other applications of data flow)
  - Is program optimization relevant?
    - Moore’s Law:
      - Advances in hardware double computing power every 18 months
    - Proebsting’s Law:
      - Advances in compiler optimizations double computing power every 18 years

Optimization Research Today

- Still around: “Doing optimization X increases speed by Y%”
  - What about Proebsting’s Law?
  - Y% may be smaller than cache effects
  - Optimizations all interact in complicated ways
- New (?) area: low-power devices
  - Typically, faster code = lower power consumption
  - But optimization itself may take power
- New (?) area: new chip designs
  - E.g., speculative execution on IA-64

Language-Based Security

- Writing secure software is hard
  - Adversary is malicious: looking for bugs
  - Hard to test for security flaws
    - Often errors on non-covered paths
- Not many mechanisms in languages for security
  - Type and memory safety help (e.g., don’t use C)
  - One exception: Stack inspection in Java
    - But what does it mean? What security can it achieve?
Secure Information Flow

• A popular notion of security: non-interference
  - Idea: Program is a function $H \times L \rightarrow H' \times L'$
    - $H = \text{high security, } L = \text{low security}$
  - High-security inputs should not leak to low-security outputs
    - Leaving $L$ fixed and changing only $H$ should not change $L'$
  - Is this a safety property? A liveness property?
  - What evidence shows this property is violated?

Enforcing Non-Interference

• Types distinguish high- and low-security data
  - Guarantee $H$ never flows to $L$
  - Dual of tainted/untainted type qualifiers
• But wait! What about the following:
  if $(H)$ then <do a lot of work> else sleep(100)
  - No direct flow from $H$ to $L$
  - This is a covert channel
  - Need to make PC high-security in this case

Enforcing Non-Interference (cont’d)

• But wait! What about multi-threaded code?
  if $(H)$ then <do a lot of work> else sleep(100)
  - Other process may observe schedule to find $H$
  - Need to make sides of conditional take equal time
• But wait! What if we’re supposed to leak info?
  if (passwd matches) then log-in else fail
  - Need some way to declassify information
    - In fact, this is the key to making this all work
    - The jury is still out on whether any of this is practical

Object-Oriented Languages

• We’ve mostly talked about imperative programming
  - With higher-order functions, a la ML
  - But OOP is very popular these days
  - How do we analyze object-oriented programs?
  - First step: basic theory
    - Object calculi
    - (In practice, people tend to use versions of Java)

An Object Calculus

• Terms (Abadi and Cardelli)
  - $e ::= x$ variable
  - $e = s(x) e_1, ..., e_n$ object
  - $e.\ell$ method app
  - $e := s(x)e$ method update
• Methods take self (this) parameter
  - $s(x)e$ is a method whose self parameter is named $x$
    and whose body is $e$
• No fields, only methods
  - Just like in pure lambda calculus, nothing but functions

Reduction Rules

• Let $o = [\ell_1 = s(x_1)e_1, ..., \ell_n = s(x_n)e_n]$
• Two possible reductions:
  - Invocation:
    - $\alpha.\ell e \rightarrow e[\alpha[x]]$
  - Update:
    - $\alpha.\ell := s(x)e \rightarrow [\ell_1 = s(x_1)e_1, ..., \ell_n = s(x_n)e, ..., \ell_n = s(x_n)e]$
Power of the Calculus?

- Can encode arithmetic, functions, recursion etc.

- Notice: No classes, only objects
  - Can model classes in the calculus
  - As well as inheritance, etc

- Can extend to typed object calculi
  - Subtyping, polymorphism, etc.

Dynamic Analysis

- In this course, we’ve focused on static analysis
  - But what about doing analysis of program runs?
  - Examples we’ve seen: Daikon, CCured

- Advantages:
  - Full, precise information about execution
  - Typically involves much lighter-weight reasoning
    - E.g., no worries about aliasing, context-sensitivity, or path sensitivity
    - But also some real engineering challenges

Eraser: Race Condition Detector

- Instrument program to track set of held locks
  - Safety property: For each shared memory location l, there is at least one lock always held when l is accessed
  - Basic algorithm:
    - Associate a lock set $S(l)$ with each location $l$
    - Whenever $l$ is accessed, $S(l) := S(l) \cap \{ \text{held locks} \}$
    - Error if $S(l)$ empty for some $l$
  - Refinements:
    - Only track lock set once location is shared, and more

What Do These Have in Common?

- They’re all checking safety properties
- There’s a well-defined notion of “going wrong”
- So we can do many styles of analysis:
  - Purely static: Programs that may go wrong are bad
  - Purely dynamic: Programs aren’t wrong until we find an error
  - Static then dynamic: Programs that may go wrong are instrumented with a monitor to prevent errors
  - Dynamic then static: Not usually done (but interesting!)

Aliasing

- Aliasing always seems to be a problem
  - Punted in classical data flow
  - Addressed by type systems (subtyping refs)
  - Many styles of alias analysis

- Why haven’t we solved this problem yet?
  - We have in some cases
  - There’s no one-size-fits-all solution

Linearity and Strong Updates

- An idea that came up over and over again
  - But sometimes not called this

- Idea: Every update affects exactly one location
  - If abstraction models location as a singleton, can do strong update, otherwise must do weak update

- Other application: functional programming
  - In-place update of linear data structure elements
Separation Logics: Axiomatic Semantics of the Heap

- Recall weakest precondition for assignments
  \( \{P[E;x]\} x := E \{P\} \)
  - What happens if \( x \) is aliased?
- Introduce predicates that directly describe heap
  \[ P ::= E \rightarrow E_1, E_2 \quad E \text{ points to the pair } (E_1, E_2) \]
  \[ | \quad \text{emp} \quad \text{The heap is empty} \]
  \[ | \quad P \ast Q \quad \text{Spatial conjunction} \]

- In \( P \ast Q \), the portions of the heap that \( P \) and \( Q \) describe must be disjoint

Example

- \( x \mapsto 3, y \)

Example

- \( y \mapsto 4, x \)

Example

- \( (x \mapsto 3, y) \ast (y \mapsto 4, x) \)

Linearity Again

- So why is rule for assignment valid?
  \( \{P[E;x]\} x := E \{P\} \)
- Because each location in the store is linear
  - In previous example, use of \( \ast \) implies \( x, y \) not aliased

Other Programming Paradigms

- We didn’t talk much about OOP
  - But I assume you’ve seen plenty of that
- We covered functional programming, a little
  - We focused on ML, which is call by value
    - Mostly functional language, but includes imperative constructs
  - The other camp: purely functional programming
    - No assignment statements!
    - (Not even one’s you’re not “supposed” to use much)
Purely Functional Programming

- Main exemplar: Haskell

- Why is not having updatable refs good?
  - Gives you mathematical-style reasoning
  - Example: What does (f x == f x) evaluate to?
    - ML/C/Java/etc.: can’t tell
    - Haskell: always will return true

Lazy Evaluation

- No side effects, so...
  - Can evaluate arguments to functions whenever we like

- Example:
  - integers n = n:(integers (n+1))
    - An infinite loop in a call by value language
  - take 5 (integers 0)
    - [0,1,2,3,4]
    - Works perfectly in Haskell; integers 5 not computed because take does not require it

Monads

- But real systems need to do I/O
  - Which is definitely a side effect

- How to incorporate into a “pure” language?
  - Thread the state through the computation
    type IO a = World → (a, World)
    - A value of type IO a is a function that takes the world as input and produces a new world as output, along with an a

- Examples:
  - getChar : IO char
  - putChar : char → IO ()
  - (For more info, see Peyton Jones’s “Tackling the Awkward Squad,” where these examples are from)

Sequencing and More

- A monad has two operations:
  - (>>=) : IO a → (a → IO b) → IO b
    - “Bind”: perform the first action, then the second
    - echo = getChar >>= putChar
    - return : a → IO a
    - Perform a non-side effecting computation
      - getChar >>= (c1 → getChar >>= (c2 → return (c1, c2)))

- Notice that the world is never duplicated (!)
- Notice that the IO monad is “sticky”

Aspect-Oriented Programming

- Goal: separation of concerns
  - Separate aspects of code should reside separately

- Canonical example: caching
  - Suppose we want to add memoization to a module
  - In a regular language, need to spread cache code all around:
    - foo(...) { if (cache hit) then ... else ...; save in cache; return }
    - bar(...) { if (cache hit) then ... else ...; save in cache; return }
  - We’d like the caching code to be physically together

Point Cuts and Weaving

- We use point cuts to intercept ordinary method calls and add in new aspects
  - Example (ignore syntax):
    - BeforeCall(*) { if cache hit then ... else do call }
    - AfterCall(*) { save in cache; return }

- A weaving phase distributes the aspects
  - Produces code we saw before, for standard compiler

- Main questions: Good idea? Semantics?