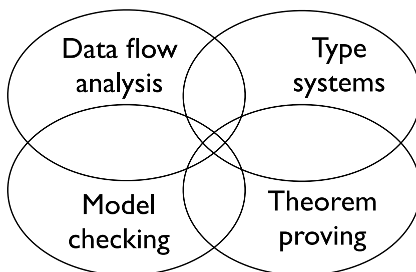
CMSC 631 – Program Analysis and Understanding
Spring 2006

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 are (basically) "e has type t," but...
    - Type checking and inference are decidable and efficient (often)
    - Type systems also know what they can't prove
      - Can usually also prove "e has no type"
- Theorem proving systems are really smart
  - Can prove very ambitious theorems
  - Often undecidable, 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:

  $VBE(p) = Used(p) \cup (notMod(p) \cap (\bigcap_{p' \in succ(p)} VBE(p')))$
- Reformatted as model checking the exec. space:

  $isVBE(e) = \nu Z.isUsed(e) \vee (notMod(e) \wedge \Box Z)$

  (here ν 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"
  - Shows how to construct a type system that accepts exactly the set of programs that a model checker passes

Denotational Semantics

- What mathematical structures do programs represent?
  - How do we reason *compositionally* about programs?

State : Variables→Values

⟪·⟫ : Statement→(State→State)

⟪skip⟫ : λs.s

⟪x=e⟫ : λs.s[v\x]   where v = ⟪e⟫s

⟪if B then C else C′⟫ : λs.if(⟪B⟫s, ⟪C⟫s, ⟪C′⟫s) where

    if(true, v, v′) = v

    if(false, v, v′) = v′

Loops in Denotational Semantics

- Loops are tricky:
  - Want ⟪while B do C⟫ to be defined in terms of B and C
  - ⟪while B do C⟫ = λs.s   if ⟪B⟫s = false
  - ⟪while B do C⟫ = ⟪C; while B do C⟫   if ⟪B⟫s = true
  - But that's not compositional reasoning!
    - while is defined in terms of itself
- Solution: Need to compute a fixpoint
  - Define *domains* on which minimal fixpoints exist

Complete Partial Orders

- A partial order $(P, \sqsubseteq)$ is a set P and a reflexive, transitive, antisymmetric binary relation $\sqsubseteq$
- A partial order *has a bottom* if it has a least element $\perp$
- An ω-*chain* is infinite increasing sequence
  - $x0 \sqsubseteq x1 \sqsubseteq x2 \sqsubseteq ...$
- A partial order is *complete* (a "cpo") if every ω-chain has a least upper bound
  - Written $\sqcup\{xi \mid i \in w\}$

(Following Abadi, CS263)

Continuous Functions

- Let P1 and P2 be two complete partial orders
- A function $f : P1 \rightarrow P2$ is *continuous* if
  - It is monotonic
    - $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$
  - For all ω-chains
    - $\sqcup_{i \in w} f(x_i) = f(\sqcup_{i \in w} x_i)$

Fixed-Point Theorem

- Let P be a cpo with bottom
- Let $f : P \rightarrow P$ be a continuous function
- Let $f^{i}(x) = f(f(\ldots f(x))$ (i times)
- Define $fix(f) = \sqcup_{i \in \omega} f^{i}(\perp)$
- Then fix(f) is the *least fixed point* of f
  - $f(fix(f)) = fix(f)$
  - if $f(x) = x$ then $fix(f) \sqsubseteq x$

Proof: First step

- Claim: $\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \ldots$
  - I.e., $f^{i}(\perp)$ forms an ω-chain
- Proof:
  - $\perp \sqsubseteq f(\perp)$ definition of ⊥
  - $f(\perp) \sqsubseteq f(f(\perp))$ monotonicity
  - $f(f(\perp)) \sqsubseteq f(f(f(\perp)))$ monotonicity
  - ...

Proof: f is a Fixpoint

$f(fix(f)) = f(\sqcup_{i \in \omega} f^{i}(\perp))$ by definition

$= \sqcup_{i \in \omega}(f(f^{i}(\perp)))$ by continuity

$= \sqcup_{i \in \omega} f^{i+1}(\perp)$

$= (\sqcup_{i \in \omega} f^{i+1}(\perp)) \sqcup \perp$

$= \sqcup_{i \in \omega} f^{i}(\perp)$

$= fix(f)$ by definition

Proof: f is the Least Fixed Point

- Let y be another fixed point, $f(y) = y$
- Then
  - $\perp \sqsubseteq y$ by definition of bottom
  - $f(\perp) \sqsubseteq f(y) = y$ by monotonicity
  - $f(f(\perp)) \sqsubseteq f(y) = y$ by monotonicity
  - $\ldots f^{i}(\perp) \sqsubseteq y$ for all $i \in \omega$
  - $\sqcup(f^{i}(\perp)) \sqsubseteq y$
  - $fix(f) \sqsubseteq y$

A Useful CPO

- Let F be the set of functions $State \rightarrow (State \cup \perp)$
- Define $f \sqsubseteq g$ if $f(x) = g(x)$ or $f(x) = \perp$
- Then F is a CPO with bottom

Denotational Semantics of While

- Goal: ⟪while B do C⟫ defined in terms of B and C
- Let G = λf.λs.if ⟪B⟫(s) then f(⟪C⟫(s)) else s
  - G "unrolls" one iteration of the loop, using f for the recursive call
  - Notice G : F → F and G continuous
- Define ⟪while B do C⟫ = fix(G)
- Then fix(G) = G(fix(G)) =

  λs.if ⟪B⟫(s) then fix(G)(⟪C⟫(s)) else s
- fix(G) is the least function with this property

Denotational Semantics

- A very compelling theory
  - Composition reasoning very powerful
  - Requires a lot of math
  - Makes some proofs easier
- Today, operational semantics mostly used
  - A lot simpler to understand
  - Reduces to a lot of symbol pushing
  - But hard to reuse results

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×L → H′×L′
    - H = high security, L = 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 L := 1 else L := 0
  - 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

Multi-Threaded Programming

- Writing multi-threaded programs is hard
  - The scheduler can interleave threads unpredictably
    - Means that it's hard to understand all the possible behaviors of your program
    - And if you do find a bug, it's hard to reproduce
  - Multi-threaded programs trade off safety and liveness
    - A safe program won't have (harmful) race conditions
      - Typically use mutual exclusion locks to enforce
      - But locking forces threads to block
    - A live program will make progress
      - Reducing the amoung of locking improves liveness

Mistakes Have Consequences

- A data race in the Therac-25, a radiation therapy machine, gave patients massive overdoses of radition, killing at least five people

- A data race was partially responsible for the northeastern US blackout of August 14, 2004, one of the worst in North American history

Parallelism is Becoming the Norm

- Desktop machines with >1 CPU are not very expensive
- Chip-Level Multiprocessors are increasingly common
  - Intel, AMD, IBM, Sun are building/will build them

- An effort to keep up with Moore's law

How Do We Program These Things?

- One idea: *atomic sections*

```
atomic { s } /* execute s atomically */
```

  - Up to the programming language to decide how to implement this
- Ideas for implementing this
  - Hardware transactional memory
  - Software transactional memory
    - Typically both of these will be optimistic concurrency
  - Mutual exclusion
    - Inferred by some static analysis