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:
  \[ VBE(p) = Used(p) \cup \left( \text{notMod}(p) \cap \left( \bigcap_{p \in \text{succ}(p)} VBE(p') \right) \right) \]
- Reformatted as model checking the exec. space:
  \[ \text{isVBE}(e) = \nu Z.\text{isUsed}(e) \lor (\text{notMod}(e) \land \square Z) \]
  (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”
  - 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
    - \( \langle \cdot \rangle \) : Statement → (State → State)
    - \( \langle \text{skip} \rangle \) : \( \lambda s.s \)
    - \( \langle x=e \rangle \) : \( \lambda s.[v\ \&\ x] \) where \( v = \langle e \rangle s \)
    - \( \langle \text{if } B \text{ then } C \text{ else } C' \rangle \) : \( \lambda s.(\langle B \rangle s, \langle C \rangle s, \langle C' \rangle s) \) where
      \[ \begin{align*}
      &\text{if(true, } v, v') = v' \\
      &\text{if(false, } v, v') = v'
      \end{align*} \]

Loops in Denotational Semantics

- Loops are tricky:
  - Want \( \langle \text{while } B \text{ do } C \rangle \) to be defined in terms of \( B \) and \( C \)
    - \( \langle \text{while } B \text{ do } C \rangle = \lambda s.s \quad \text{if } \langle B \rangle s = \text{false} \)
    - \( \langle \text{while } B \text{ do } C \rangle = \langle C \rangle ; \langle \text{while } B \text{ do } C \rangle \quad \text{if } \langle B \rangle s = \text{true} \)
  - 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 \( \bot \)
- An \( \omega \)-chain is infinite increasing sequence
  - \( x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots \)
  - A partial order is complete (a “cpo”) if every \( \omega \)-chain has a least upper bound
    - Written \( \sqcup \{ x_i \mid i \in w \} \) (Following Abadi, CS263)
**Continuous Functions**

- Let $P_1$ and $P_2$ be two complete partial orders
- A function $f : P_1 \to P_2$ is **continuous** if
  - It is monotonic
    - $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$
  - For all $\omega$-chains
    - $\biguplus_{i \in \omega} f(x_i) = f\left(\biguplus_{i \in \omega} x_i\right)$

**Fixed-Point Theorem**

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

**Proof: First step**

- Claim: $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq ...$
  - i.e., $f^i(\bot)$ forms an $\omega$-chain
- Proof:
  - $\bot \sqsubseteq f(\bot)$ definition of $\bot$
  - $f(\bot) \sqsubseteq f(f(\bot))$ monotonicity
  - $f(f(\bot)) \sqsubseteq f(f(f(\bot)))$ monotonicity
  - ...

**Proof: $f$ is a Fixpoint**

- $f(f(\text{fix}(f))) = f(\biguplus_{i \in \omega} f^i(\bot))$ by definition
- $= \biguplus_{i \in \omega} f(f^i(\bot))$ by continuity
- $= \biguplus_{i \in \omega} f^{i+1}(\bot)$
- $= (\biguplus_{i \in \omega} f^{i+1}(\bot)) \sqcup \bot$
- $= \biguplus_{i \in \omega} f^i(\bot)$
- $= \text{fix}(f)$ by definition

**Proof: $f$ is the Least Fixed Point**

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

**A Useful CPO**

- Let $F$ be the set of functions $\text{State} \to (\text{State} \sqcup \bot)$
- Define $f \sqsubseteq g$ if $f(x) = g(x)$ or $f(x) = \bot$
- Then $F$ is a CPO with bottom
Denotational Semantics of While

- Goal: $\{\text{while } B \text{ do } C\}$ defined in terms of $B$ and $C$
- Let $G = \lambda f.\lambda s.\text{if } (B)(s) \text{ then } f((C)(s)) \text{ else } s$
  - $G$ “unrolls” one iteration of the loop, using $f$ for the recursive call
  - Notice $G : F \rightarrow F$ and $G$ continuous
- Define $\{\text{while } B \text{ do } C\} = \text{fix}(G)$
- Then $\text{fix}(G) = G(\text{fix}(G)) = \lambda s.\text{if } (B)(s) \text{ then } \text{fix}(G)((C)(s)) \text{ else } s$
  - $\text{fix}(G)$ 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 $H \times L \rightarrow H' \times L'$
    - $H =$ 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
- But wait! What about multi-threaded code?
  - if $(H)$ then $<\text{do a lot of work}>$ 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 $(\text{passwd matches})$ 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 amount of locking improves liveness

Mistakes Have Consequences

• A data race in the Therac-25, a radiation therapy machine, gave patients massive overdoses of radiation, 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
  
  \[
  \text{atomic ( s ) /* execute s atomically */}
  \]

  ▪ 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