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.
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:
  \[ VBE(p) = \text{Used}(p) \cup \text{notMod}(p) \cup (\bigcap_{p \in \text{succ}} VBE(p')) \]
- 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

Loops in Denotational Semantics

- Loops are tricky:
  - Want \( \langle \text{while} B \text{ do } C \rangle \) to be defined in terms of \( B \) and \( C \)
  - \( \langle \text{while} B \text{ do } C \rangle = \lambda s.s \) if \( \langle B \rangle s = \text{false} \)
  - \( \langle \text{while} B \text{ do } C \rangle = \langle C; \text{while } B \text{ do } C \rangle \) if \( \langle B \rangle s = \text{true} \)
  - But that’s not compositional reasoning!
    - \( \text{while} \) is defined in terms of itself
  - Solution: Need to compute a fixpoint
    - Define domains on which minimal fixpoints exist

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

Complete Partial Orders

- A partial order \((P, \preceq)\) is a set \( P \) and a reflexive, transitive, antisymmetric binary relation \( \preceq \)
- A partial order has a bottom if it has a least element \( \bot \)
- An \( \omega \)-chain is infinite increasing sequence
  - \( x_0 \preceq x_1 \preceq x_2 \preceq \ldots \)
- A partial order is complete (a “cpo”) if every \( \omega \)-chain has a least upper bound
  - Written \( \bigcup \{x_i | i \in w\} \) (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
    - $\bigcup_{i \in \omega} f(x_i) = f\left(\bigcup_{i \in \omega} x_i\right)$

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)

**Proof: First step**

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

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

- $f(fix(f)) = f\left(\bigcup_{i \in \omega} f^i(\bot)\right)$ by definition
- $= \bigcup_{i \in \omega} f(f^i(\bot))$ by continuity
- $= \bigcup_{i \in \omega} f^{i+1}(\bot)$
- $= (\bigcup_{i \in \omega} f^{i+1}(\bot)) \cup \bot$
- $= \bigcup_{i \in \omega} f^i(\bot)$
- $= fix(f)$ by definition

**A Useful CPO**

- Let $F$ be the set of functions $\text{State} \to (\text{State} \cup \bot)$
- 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 \( G = \lambda z \lambda s. \text{if } (B)(s) \text{ then } f((C)(s)) \text{ else } s \)
- \( G \) "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 \( \text{tainted/untainted} \) type qualifiers
- But wait! What about the following:
  - \( \text{if } (H) \text{ then } L := 1 \text{ else } L := 0 \)
    - No direct flow from \( H \) to \( L \)
      - This is a covert channel
      - Need to make PC high-security in this case
- But wait! What if we’re supposed to leak info?
  - \( \text{if } (\text{passwd matches}) \text{ then } \text{log-in else fail} \)
    - Need some way to \text{classify} 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)
  \[ e ::= x \quad \text{variable} \]
  \[ | \begin{array}{l} \llbracket \ell_1 = s(x_1)e_1, \ldots, \ell_n = s(x_n)e_n \rrbracket \quad \text{object} \\ \ell.e \quad \text{method app} \\ \ell.e := s(x)e \quad \text{method update} \end{array} \]

• 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 \( o = \llbracket \ell_1 = s(x_1)e_1, \ldots, \ell_n = s(x_n)e_n \rrbracket \)

  Two possible reductions:
  ▪ Invocation:
    \[ \ell.e \rightarrow e[o \xi] \]
  ▪ Update:
    \[ \ell.e := s(x)e \rightarrow \llbracket \ell_1 = s(x_1)e_1, \ldots, \ell = s(x)e, \ldots, \ell_n = s(x_n)e_n \rrbracket \]

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.

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 \text{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”