CMSC 631 – Program Analysis and Understanding
Fall 2017
PL: A Whirlwind Tour
Semantics and Foundations
Program Semantics

• To analyze programs, we must know what they mean
  ▪ Semantics comes from the Greek semaino, “to mean”

• Most language semantics informal. But we can do better by making them formal. Two main styles:
  ▪ Operational semantics (major focus)
    - Like an interpreter
  ▪ Denotational semantics
    - Like a compiler
Operational Semantics

• This is the most popular style of semantics

• Evaluation is described as transitions in some abstract machine; a definitional interpreter
  ▪ The meaning of a program is its fully reduced form (a.k.a. a value)
  ▪ Two main styles: Small step and big step

• Can be used to model a large variety of programming language features
  ▪ Concurrency, non-determinism, run-time cost, higher order functions (see https://dl.acm.org/citation.cfm?id=805852) …
Denotational Semantics

• The meaning of a program is defined as a mathematical object, e.g., a function or number

• Typically define an interpretation function \([ \ ]\)
  - Meaning of program fragment (arg) in a given state
  - E.g., \([ x+4 \])_\sigma = 7
    - \sigma is the state — a map from variables to values
    - Here \(\sigma(x) = 3\)

• Gets interesting when we try to find denotations of loops or recursive functions
Denotational Semantics Example

• $b ::= \text{true} \mid \text{false} \mid b \lor b \mid b \land b \mid e = e$

• $e ::= 0 \mid 1 \mid \ldots \mid x \mid e + e \mid e \ast e$

• $s ::= e \mid x := e \mid \text{if } b \text{ then } s \text{ else } s \mid \text{while } b \text{ do } s$

Semantics (booleans):

- $\llbracket \text{true} \rrbracket_\sigma = \text{true}$

- $\llbracket b_1 \lor b_2 \rrbracket_\sigma = \begin{cases} \text{true} & \text{if } \llbracket b_1 \rrbracket_\sigma = \text{true} \text{ or } \llbracket b_2 \rrbracket_\sigma = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$

- $\llbracket e_1 = e_2 \rrbracket_\sigma = \begin{cases} \text{true} & \text{if } \llbracket e_1 \rrbracket_\sigma = \llbracket e_2 \rrbracket_\sigma \\ \text{false} & \text{otherwise} \end{cases}$
Denotational Semantics cont’d

- \( [\ x \ ]\sigma \) = \( \sigma(x) \)

- \( [\ x := e \ ]\sigma \) = \( \sigma[x \mapsto [e]\sigma] \)
  (remap \( x \) to \( [e]\sigma \) in \( \sigma \))

- \( [\ if\ b\ then\ s1\ else\ s2\ ] \) = \( \begin{cases} [s1]\sigma & \text{if } [b]\sigma = \text{true} \\ [s2]\sigma & \text{if } [b]\sigma = \text{false} \end{cases} \)
Complication: Recursion

- The denotation of a loop is decomposed into the denotation of the loop itself

\[
\llbracket \text{while } b \text{ do } s \text{ end} \rrbracket_\sigma = \begin{cases} 
\llbracket s; \text{while } b \text{ do } s \text{ end} \rrbracket_\sigma & \text{if } \llbracket b \rrbracket_\sigma = \text{true} \\
\sigma & \text{if } \llbracket b \rrbracket_\sigma = \text{false}
\end{cases}
\]

- Recursive functions introduce a similar problem

- Solution: Denotation not in terms of sets of values, but as complete partial orders (CPOs).
  - Poset with some additional properties. Dana Scott (CMU) applied these to PL semantics (Scott domains)
  - Ensures we can always solve the recursive equation
Other language features

• Tuples, variants (unions), higher order and recursive functions all possible
  ▪ Higher order functions are functions that can take other functions as arguments

• But other language features are much harder
  ▪ E.g., concurrency

• See Cornell CS611 for a full development
Applications

• More powerful than operational semantics in some applications, notably *equational reasoning*

  ▪ The Foundational Cryptography Framework (probabilistic programs)
    - http://adam.petcher.net/papers/FCF.pdf
  
  ▪ A Semantic Account of Metric Preservation (privacy)
    - https://www.cis.upenn.edu/~aarthur/metcpo.pdf

  ▪ Basic Reasoning (equivalence)
Lambda Calculus

• Invented by Alonzo Church in 1930

• Three syntactic forms

\[ e ::= x \quad \text{variable} \]
\[ \lambda x . e \quad \text{function} \]
\[ e \ e \quad \text{function application} \]

• One (operational) reduction rule

\[ (\lambda x . e_1 ) \ e_2 \rightarrow e_1 [e_2 \ x] \quad \text{(replace } x \text{ with } e_2 \text{ in } e_1) \]
Example

• Conditionals
  - true = \( \lambda x.\lambda y. x \)  
    false = \( \lambda x.\lambda y. y \)
  - if \( a \) then \( b \) else \( c \) = \( a \ b \ c \)
  - if true then \( b \) else \( c \) = \( (\lambda x.\lambda y. x) \ b \ c \rightarrow (\lambda y. b) \ c \rightarrow b \)
  - if false then \( b \) else \( c \) = \( (\lambda x.\lambda y. y) \ b \ c \rightarrow (\lambda y. y) \ c \rightarrow c \)

• Can also represent numbers, pairs, data structures, recursive functions, etc.
  - Turing complete!
Recursion with Y combinator

\[ Y = \lambda f. (\lambda x. f (x \ x)) (\lambda x. f (x \ x)) \]

Then \( Y \ F = \)

\[
(\lambda f. (\lambda x. f (x \ x)) (\lambda x. f (x \ x))) \ F \rightarrow \\
(\lambda x. F (x \ x)) (\lambda x. F (x \ x)) \rightarrow \\
F ((\lambda x. F (x \ x)) (\lambda x. F (x \ x)))
\]

\[ = F (Y \ F) \]

- \( Y \ F \) is a fixed point (aka “fixpoint”) of \( F \)
  - Thus \( Y \ F = F (Y \ F) = F (F (Y \ F)) = \ldots \)
Other Core Calculi

• SKI combinator calculus
  - Reduced version of the lambda calculus
  - Ix = x; Kxy = x; Sxyz = xz(yz)

• Π calculus for communicating processes
  - spi-calculus extends to cryptographic protocols
  - CSP (communicating sequential processes) similar

• SECD machine
  - Abstract machine for lambda calculus evaluation
  - See also CESK machine, Krivine machine, …
Automated Reasoning
Static Program Analysis

• Method for proving properties about a program’s executions
  ▪ Works by analyzing the program without running it

• Static analysis can prove the absence of bugs
  ▪ Testing can only establish their presence

• Many techniques
  ▪ Abstract interpretation
  ▪ Dataflow analysis
  ▪ Symbolic execution
  ▪ Type systems, …
Soundness and Completeness

- Suppose a static analysis $S$ attempts to prove property $R$ of program $P$
  - E.g., $R = “\text{program has no run-time failures}”$
  - $S(P) = \text{true}$ implies $P$ has no run-time failures

- An analysis is **sound** iff
  - for all $P$, if $S(P) = \text{true}$ then $P$ exhibits $R$

- An analysis is **complete** iff
  - for all $P$, if $P$ exhibits $R$ then $S(P) = \text{true}$
Abstract Interpretation

- Rice’s Theorem: Any non-trivial property of programs is undecidable (not both sound and complete)
  - Uh-oh! We can’t do anything. So much for this course...

- Need to make some kind of approximation
  - Abstract the behavior of the program
  - ...and then analyze the abstraction in a sound way
    - Proof about abstract program —> proof of real one

- Seminal papers: Cousot and Cousot, 1977, 1979
### Example

**e ::= n | e + e**

- **Abstract semantics:**

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>0</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Notice the need for ? value
- Arises because of the abstraction
Abstract Domains, and Semantics

• Many abstractions possible
  ▪ **Signs** (previous slide)
  ▪ **Intervals**: $\alpha(n) = [l,u]$ where $l \leq n \leq u$
    - $l$ can be $-\infty$ and $u$ can be $+\infty$
  ▪ **Convex polyhedra**: $\alpha(\sigma) = \text{affine formula over variables in domain of } \sigma$, e.g., $x \leq 2y + 5$
    - where $\sigma$ is a state mapping variables to numbers
    - *relational* domain

• Abstract semantics for standard PL constructs
  ▪ Assignments, sequences, loops, conditionals, etc.
Abstracting Abstract Machines

• Instead of abstracting a normal programming language, we can abstract its abstract machine
  ▪ E.g., a CESK machine, or SECD machine
• This can be done systematically
• Great tutorial at https://dvanhorn.github.io/redex-aam-tutorial/
Applications: Abstract Interpretation

• ASTREE (ENS and others)
  ▪ Used to detect all possible runtime failures (divide by zero, null pointer deref, array out of bounds) on embedded code
  ▪ Used regularly on Airbus avionics software

• Terminator (Microsoft)
  ▪ Analyzes code to prove that it terminates (!)
  ▪ Applied to device drivers for Windows kernel
    - Tricky part is reasoning about the heap
Dataflow Analysis

- Classic style of program analysis
- Used in optimizing compilers
  - Constant propagation
  - Common sub-expression elimination
  - Loop unrolling and code motion
- Efficiently implementable
  - At least, *intraprocedurally* (within a single proc.)
  - Use bit-vectors, fixpoint computation
\begin{verbatim}
x := 3
if (!x) then
  y := z + w
else
  L: { y := 0 }
x := 2 * x
if (!x) then goto L
\end{verbatim}
Lattices and Termination

• Dataflow facts form a lattice

\[
\begin{align*}
\text{x = ?} \\
\text{x = 3} & \quad \text{x = 6} & \quad \text{...} \\
\text{x = *}
\end{align*}
\]

• Each statement has a transfer function
  
  Out(S) = Gen(S) U (In(S) - Kill(S))

• Terminates because
  
  Finite height lattice
  
  Monotone transformation functions
Applications: Dataflow analysis

• Optimizing compilers
  - i.e., any good compiler (gcc, LLVM)

• ESP: Path-sensitive program checker (Microsoft)
  - Example: can check for correct file I/O properties, like files are opened for reading before being read

• FindBugs (UMD): applies dataflow analysis to discover null pointer and other bugs in JVM programs
Relating Dataflow and AbsInterp

• Abstract interpretation was originally developed as a formal justification for data flow analysis

• As such, mechanics are similar:
  ▪ Abstract domain, organized as a lattice
  ▪ Transfer functions = abstract semantics
  ▪ Fixed point computation
    - “join” at terminus of conditional, while
    - iterate until abstract state unchanged
Symbolic Execution

• Testing works
  ▪ But, each test only explores one possible execution
    - assert(f(3) == 5)
  ▪ We hope test cases generalize, but no guarantees

• Symbolic execution generalizes testing
  ▪ Allows unknown symbolic variables in evaluation
    - \( y = \alpha; \) assert(f(y) == 2*y-1);
  ▪ If execution path depends on unknown, conceptually fork symbolic executor
    - int f(int x) { if (x > 0) then return 2*x - 1; else return 10; }
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.     x = -2;
6. }
7. if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10. }
11. assert(x+y+z!=3)
Relating SymExe and AbsInterp

• Symbolic execution is a kind of abstract interpretation, but
  ▪ Abstract domain may not be a lattice (includes concrete elements)
    - so no guarantee of termination
  - No joins at control merge points
    - again, challenges termination
• But lack of termination trades off with completeness
  ▪ No correct program is implicated falsely
Applications: Symbolic Execution

• SAGE (Microsoft)
  ▪ Used as a fuzz tester to find buffer overruns etc. in file parsers. Now industrial product

• KLEE (Imperial), Angr (UCSB), Triton (Inria), ...
  ▪ Research systems used to enforce security specifications, find vulnerabilities, explore configuration spaces, and more
Automating Axiomatic Semantics

- \{P\} S \{Q\}
  - If statement \(S\) is executed in a state satisfying precondition \(P\), then \(S\) will terminate, and \(Q\) will hold of the resulting state
  - Partial correctness: ignore termination

- Such Hoare triples proved via set of rules
  - Rules proved sound WRT denotational or operational semantics
Proofs of Hoare Triples

- Example rules
  - Assignment: \( \{Q[E \mapsto x]\} x := E \{Q\} \)
  - Conditional: \( \begin{align*}
  &\{P \land B\} S1 \{Q\} \quad \{P \land \neg B\} S2 \{Q\} \\
  &\{P\} \text{ if } B \text{ then } S1 \text{ else } S2 \{Q\}
  \end{align*} \)

- Example proof (simplified)

\( \begin{align*}
\{y>3\} x := y \{x>3\} &\quad \{\neg(y>3)\} x := 4 \{x>3\} \\
\{\} \text{ if } y>3 \text{ then } x := y \text{ else } x := 4 \{x>3\}
\end{align*} \)
Automation: Program as Logic

• Can use axiomatic semantics to convert a program’s semantics into a logical formula
  - **Weakest preconditions** can be automatically generated for everything but loops
  - Also, **strongest postconditions**, or **verification conditions**, by going forward, not backward
  - Can infer loop invariants in some cases

• Can make the process modular with hand-written method pre- and post conditions
Extensions

• Separation logic
  ▪ For reasoning about the heap in a modular way
  ▪ Contrasts with rules due to John McCarthy
• “modifies” clauses for method calls, side effects
• Dijkstra monads
  ▪ Extends Hoare-style reasoning to functional programs (i.e., those with functions that can take functions as arguments)
• Rely-guarantee reasoning for multiple threads
Applications: Verification

• Dafny (Microsoft)
  - Can perform deep reasoning about programs
    - Array out-of-bounds, null pointer errors, failure to satisfy internal invariants
  - Employs the Z3 SMT solver
  - Ironclad project: https://www.microsoft.com/en-us/research/project/ironclad/

• Long line of other tools, e.g., Spec# (Microsoft), F* (Microsoft), ESC/Java (many)
Type Systems

• A type system is
  ▪ a *tractable syntactic method for proving the absence of certain program behaviors* by *classifying* phrases according to the *kinds of values* they compute. --Pierce

• They are good for
  ▪ Detecting errors (don’t add an integer and a string)
  ▪ Abstraction (hiding representation details)
  ▪ Documentation (tersely summarize an API)

• Designs trade off efficiency, readability, power
**Simply-typed λ-calculus**

\[ e ::= x | n | \lambda x:\tau . e | e \ e \]

\[ \tau ::= \text{int} | \tau \rightarrow \tau \]

\[ A ::= \cdot | A, x:\tau \]

---

**Type Inference**

\[ A \vdash e : \tau \]

in type environment \( A \), expression \( e \) has type \( \tau \)

---

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \vdash n : \text{int} )</td>
<td>( A \vdash x : A(x) )</td>
</tr>
<tr>
<td>( x \in \text{dom}(A) )</td>
<td>( A \vdash e1 : \tau \rightarrow \tau' )</td>
</tr>
<tr>
<td>( A , \tau : x \vdash e : \tau' )</td>
<td>( A \vdash e2 : \tau )</td>
</tr>
<tr>
<td>( A \vdash \lambda x : \tau . e : \tau \rightarrow \tau' )</td>
<td>( A \vdash e1 \ e2 : \tau' )</td>
</tr>
</tbody>
</table>
Type Safety

• If $\vdash e : \tau$ then either
  
  ▪ there exists a value $v$ of type $\tau$ such that $e \rightarrow^* v$, or
  
  ▪ $e$ diverges (doesn’t terminate)

• Corollary: $e$ will never get “stuck”
  
  ▪ never evaluates to a normal form that is not a value

• Proof by induction on the typing derivation
Type Inference

• Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?
Type Language

• Problem: Consider the rule for functions

\[ A, x:t \vdash e : t' \]

\[ \frac{}{A \vdash \lambda x:t.e : t \to t'} \]

• Without type annotations, where do we get \( t \)?
  - We’ll use type variables to stand for as-yet-unknown types
    - \( t ::= \alpha \mid \text{int} \mid t \to t \)
  - We’ll generate equality constraints \( t = t \) among the types and type variables
    - And then we’ll solve the constraints to compute a typing
Type Inference Rules

\[ A \vdash n : \text{int} \]

\[ x \in \text{dom}(A) \]

\[ A \vdash x : A(x) \]

\[ A, x : \alpha \vdash e : t' \quad \alpha \text{ fresh} \]

\[ A \vdash \lambda x. e : \alpha \to t' \]

\[ A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2 \]

\[ t_1 = t_2 \to \beta \]

\[ \beta \text{ fresh} \]

\[ A \vdash e_1 \ e_2 : \beta \]

“Generated” constraint
Example

\[
\begin{align*}
A, x: \alpha & \vdash x: \alpha \\
\hline
A & \vdash (\lambda x.x) : \alpha \rightarrow \alpha & A & \vdash 3 : \text{int} & \alpha \rightarrow \alpha = \text{int} \rightarrow \beta \\
& \hline
A & \vdash (\lambda x. x) \ 3 : \beta
\end{align*}
\]

• We collect all constraints appearing in the derivation into some set \( C \) to be solved
• Here, \( C \) contains just \( \alpha \rightarrow \alpha = \text{int} \rightarrow \beta \)
  ▪ Solution: \( \alpha = \text{int} = \beta \)
• Thus this program is typable, and we can derive a typing by replacing \( \alpha \) and \( \beta \) by \text{int} in the proof
Scaling up

• Type inference works well in limited settings
  ▪ Hindley-Milner type inference in ML seems to be a sweet spot

• The more fancy the type language, the more difficult it gets to do well
  ▪ Higher-order functions and subtyping, dependent types, linear types, …

• Connection:
  ▪ Whole-program type inference = static analysis
Types, Types, Types, Oh my!

- Sums $\tau_1 + \tau_2$
- Products $\tau_1 \times \tau_2$
- References $\tau \text{ ref}$
- Recursive types $\mu \alpha. \tau$
- Universals $\forall \alpha. \tau$
- Existentials $\exists \alpha. \tau$
- Dependent functions $\Pi x: \tau_1. \tau_2$
- Dependent products $\Sigma x: \tau_1. \tau_2$

$\alpha \text{ list} = \forall \alpha. \mu \beta. \text{unit} + (\alpha \times \beta)$
Dependent Types

- Useful for expressing properties of programs
  - \([1;2;3] : \text{int list}\)
  - \([1;2;3] : \text{int 3 list}\)
  - `append: 'a \text{n list} \rightarrow 'a \text{m list} \rightarrow 'a (\text{m+n} \text{ list})`

- The above types are encoded using the primitive concepts above (plus a little more)

- Gives stronger assurances of correct usage
  - Prove impossibility of run-time match failures
Refinement Types

• Normal types accompanied by logical formula to refine the set of legal values

• Example: \{ n:int | n \geq 0 \}
  ▪ Type for non-negative integers
  ▪ This is a kind of dependent type

• Present in several languages
  ▪ Liquid Haskell, F*
Subtyping

• Liskov substitution principle:
  ■ If for each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $o_1$, the behavior of $P$ is unchanged when $o_2$ is substituted for $o_1$ then $S$ is a subtype of $T$.
    - Written $S <: T$

• Informal statement
  ■ If anyone expecting a $T$ can be given an $S$ instead, then $S$ is a subtype of $T$. 
Goodness Properties by Typing

• Formulate an operational semantics for which violation of a useful property results in a stuck state. Eg,
  ▪ The program **divides by zero**, dereferences a **null pointer**, accesses an **array out of bounds**
  ▪ A thread attempts to **dereference a pointer without holding a lock** first
  ▪ The program **uses tainted data** (potentially from an adversary) where untainted data expected (e.g., as a format string)
• Then formulate a type system that enforces the property, and prove type safety
Linear Types for Safe Memory

• Garbage collection is used by most languages to help ensure type safety
  ▪ But it can add memory overhead, excessive pause times, and general overhead

• Manual memory management is faster, but a frequent source of bugs
  ▪ Use-after-free bugs, (some) memory leaks

• Idea: Enforce correct use of manual memory management through the type system
Cyclone

- Safe variant of C
  - Bounds safety (no access outside of arrays, etc.) and temporal safety (no use-after-free)
  - Manual and automatic memory management
  - Modern features (pattern matching, polymorphism…)
- Unique pointers cannot be aliased
  - `int *`U x = malloc(sizeof(int));
  - y = x; // consumes x
  - // *x = 0; would fail; x was consumed
  - free(y); // ok
Proof of Soundness

• Operational semantics wherein memory is tagged with whether it’s valid or not
  ▪ Freeing memory makes it invalid
  ▪ We use memory once—ignore recycling

• Whenever a pointer is dereferenced, check that the target in memory is valid; stuck if not

• Type safety: non-stuckness implies no freed memory is ever used
Rust

• Actively developed by Mozilla. Uses ideas from Cyclone and other languages

• *Ownership* in Rust =~ linearity
  - Only one variable can own a free-able resource
  - Assignment transfers ownership
  - Temporary aliasing allowed within a limited program scope; called borrowing
    - [https://rustbyexample.com/scope/borrow.html](https://rustbyexample.com/scope/borrow.html)
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);
    // `c` is destroyed and the memory freed
}

fn main() {
    // _Stack_ allocated integer
    let x = 5u32;

    // *Copy* `x` into `y` - no resources are moved
    let y = x;

    // Both values can be independently used
    println!("x is {}, and y is {}", x, y);

    // `a` is a pointer to a _heap_ allocated integer
    let a = Box::new(5i32);

    println!("a contains: {}", a);

    // *Move* `a` into `b`
    let b = a;
    // The pointer address of `a` is copied (not the data) into `b`.
    // Both are now pointers to the same heap allocated data, but
    // `b` now owns it.

    // Error! `a` can no longer access the data, because it no longer owns the
    // heap memory
    //println!("a contains: {}", a);
    // TODO ^ Try uncommenting this line

    // This function takes ownership of the heap allocated memory from `b`
    destroy_box(b);

    // Since the heap memory has been freed at this point, this action would
    // result in dereferencing freed memory, but it's forbidden by the compiler
    // Error! Same reason as the previous Error
    //println!("b contains: {}", b);
    // TODO ^ Try uncommenting this line
}
ATS and F*

• ATS: Language for systems code that employs both dependent types and linear types
  ▪ Used to develop device drivers and kernels that are provably safe

• F*: Also useful for systems code. Employs dependent and refinement types
  ▪ Avoids thorny issues with memory management
  ▪ Everest project for provably correct TLS, HTTPS
Checked C

- Started at Microsoft Research ~2 years ago
  - https://github.com/Microsoft/checkedc
- Focus is on annotations to enforce bounds safety
- Backward compatible with existing C
  - Thus, soundness property is partial
- Current status
  - Special types for safe pointers, safe arrays, safe null-terminated pointers
  - Annotations for easy interoperation with legacy code
Secure Information Flow

• Secure information flow (secrecy)
  ▪ password: secret int, guess: public int
  ▪ type system ensures secret values can’t be inferred by observing public values

• Dual: Avoiding undue influence (integrity)
  ▪ user_pass: tainted string, db_query: untainted string
  ▪ Make sure that tainted data does not get used where untainted data is required
Kinds of Information Flows

• How can information flow from \text{H} to \text{L}?

• \textit{Direct flows}

\begin{verbatim}
h := l;
x := l; y := x; h := y;
\end{verbatim}

• \textit{Implicit flows}

\begin{verbatim}
h := h \mod 2;
l := 0;
if h == 1 then l := 1 else skip
\end{verbatim}

– The low order bit of \text{h} was copied through the pc!
Preventing Explicit Flows

• Goal: Build a program analysis that will prevent flows from high security inputs to low security outputs
  – But first, let’s generalize from just two security levels (high, low) to many

• Security labels:
  – Lattice \((S, \leq)\)
    – \(S\) is the set of labels
    – \(s_1 \leq s_2\) if \(s_1\) allowed to flow to \(s_2\)
      » e.g., \(\text{let } f (x:s_2) = \ldots \text{ in } f (y:s_1)\)
    – confidentiality: \(s_1\) is “less secret” than \(s_2\)
    – integrity: \(s_1\) is “more trusted” than \(s_2\)
Preventing Explicit Flows by Typing

• Build a type system that rejects programs with bad explicit flows
  – $e ::= x | e \, \text{op} \, e | n$
  – $c ::= \text{skip} | x := e | \text{if } e \\text{then } c_1 \\text{else } c_2 | \text{while } e \\text{do } c$
  – $t ::= \text{int } S$  \textit{types tagged with security level}
  – $A : \text{vars } \rightarrow t$
### Preventing Explicit Flows (cont’d)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \vdash x : t )</td>
<td></td>
</tr>
<tr>
<td>( A \vdash x : A(x) )</td>
<td>( A \vdash n : \text{int } S )</td>
</tr>
<tr>
<td>( A \vdash c )</td>
<td></td>
</tr>
<tr>
<td>( A \vdash \text{skip} )</td>
<td>( A \vdash e : \text{int } S )  ( A(x) = \text{int } S' )  ( S \leq S' )</td>
</tr>
<tr>
<td>( A \vdash e : \text{int } S )  ( A \vdash c1 )  ( A \vdash c2 )</td>
<td>( A \vdash e : \text{int } S )  ( A \vdash c )</td>
</tr>
</tbody>
</table>
Notes

• Here we assume all variables have some type in $A$ at the beginning of execution
  – So, essentially this type systems checks whether the annotations in $A$ are correct
• Lets $L$ be assigned to $H$, but not vice-versa (see assignment rule)
• Can be generalized to other types aside from int
  – See type qualifiers papers
• Does not prevent implicit flows
  – Nothing interesting going on for if, while
Proof of Soundness

• Develop an operational semantics that tags data with its security label, and likewise tags storage/channels
  – Track tags through program operations (using $\sqcup$ operator)
  – When storing data, or writing to a channel, make sure tags are compatible; if not program is stuck
  – Similar to Perl, Ruby, etc. taint mode

• Prove that a type-correct program never gets stuck
Aside: Dynamic Enforcement

- *Implement* “monitoring” semantics via instrumentation
  - Accepts more (all!) programs. Defers error checks to run-time (which adds overhead)

- Many examples
  - Phosphor for Java (taint analysis)
Back to InfoFlow: Implicit Flows

• Intuition: The program counter conveys sensitive information if we branch on a high-security value

\[
\text{if } h > 0 \text{ then } l := 1 \text{ else } l := 0;
\]

• Slightly more complicated: information flow depends both on what is done and what is *not* done

\[
\begin{align*}
l & := 0; \\
\text{if } h > 0 \text{ then } l & := 1 \text{ else skip;}
\end{align*}
\]

– Fortunately, we are doing static analysis, so we can look at *both* branches

– Much harder in a dynamic setting!
Preventing Implicit Flows (cont’d)

\[ A \vdash x : A(x) \quad (\text{same as before}) \]

\[ A \vdash x : A(x) \quad A \vdash n : \text{int } S \quad A \vdash e : \text{int } S_1 \quad A \vdash e_2 : \text{int } S_2 \quad A \vdash e_1 \text{ op } e_2 : \text{int } (S_1 \sqcup S_2) \]

\[ A, \ S \vdash c \]

\[ A \vdash e : \text{int } S \quad A(x) = \text{int } S' \quad S \sqcup S_{\text{pc}} \leq S' \]

\[ A, \ S \vdash c \]

\[ A \vdash \text{skip} \quad A \vdash x := e \]

\[ A \vdash e : \text{int } S \quad A, \ S_{\text{pc}} \sqcup S \vdash c_1 \quad A, \ S_{\text{pc}} \sqcup S \vdash c_2 \]

\[ A, \ S_{\text{pc}} \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \]

\[ A \vdash e : \text{int } S \quad A, \ S_{\text{pc}} \sqcup S \vdash c \]

\[ A, \ S_{\text{pc}} \vdash \text{while } (e) \text{ do } c \]
Application to Java

• Jif (Java+Information Flow)
  ▪ Annotate standard types with additional security labels, where type correctness implies correct protection of sensitive data
• Jif is at the core of a number of other projects too
  ▪ Fabric framework, for cloud computing
  ▪ Civitas, secure remote voting system
Proof of Security

• The property that we have no explicit flows is not strong enough for real security.

• Want a property called **noninterference**
  - No matter what the secret values are, the publicly visible ones do not change
  - I.e., secret values do not interfere with visible ones

• Proof is more involved
  - Involves a *logical relation* which defines an equivalence on terms that are indistinguishable to the adversary
Abstraction Safety

- Type safety is about avoiding run-time errors
- Types also provide *abstraction safety*
  - Enforcing invariants on a module’s private data structures
  - Representation independence: Should be able to swap out implementations as long as external behavior unchanged
- Proved using logical relations
Alternatives to Pure Static Typing

• Dynamic Types (Cardelli – CFPL 1985)
  ■ Dynamic-typed values pair typed values with their type
  ■ Dynamic values in typed positions check type at run-time

• Soft Typing (Cartwright, Fagan – PLDI 1991)
  ■ Adds explicit run-time checks where typechecker cannot prove type correctness
  ■ Allows running possibly ill-typed programs

• Gradual Typing
  ■ Parallel work
  ■ Focuses on providing sister typed and untyped languages
  ■ Allows interaction between typed and untyped modules
Key Insight to Gradual Typing

Handle typed/untyped boundary crossings with dynamic checks (e.g., contracts)
Gradual Type Soundness

In a gradual typing system, type soundness looks something like the following:

For all programs, if the typed parts are well-typed, then evaluating the program either

1. produces a value,
2. diverges,
3. produces an error that is not caught by the type system (e.g., division by zero),
4. produces a run-time error in the untyped code, or
5. produces a contract error that blames the untyped code.
Program Synthesis

Find a program $P$ that meets a spec $\phi(input,output)$:

$$\exists P \forall x. \phi(x,P(x))$$

When to use synthesis:

**productivity:** when writing $\phi$ is faster than writing $P$

**correctness:** when proving $\phi$ is easier than proving $P$
Preparing your language for synthesis

Extend the language with two constructs

**spec:**
```c
int foo (int x) {
    return x + x;
}
```

**sketch:**
```c
int bar (int x) implements foo {
    return x << ??;
}
```

**result:**
```c
int bar (int x) implements foo {
    return x << 1;
}
```

Instead of `implements`, assertions over safety properties can be used.
Synthesis from partial programs

spec

sketch

program-to-formula translator

\( \phi \)

solver

“synthesis engine”

\( h \mapsto 1 \)

code generator

\( P[1] \)

\( P[h] \)

sketch
Probabilistic Programming

• Programs operate on random and/or noisy values

• Can interpret such a program as a distribution
  ▪ Each run of the program is a sample from the distribution

• Technical problem: How to get a representation of that distribution to perform inference?
Estimated Glomular Filtration Rate

```c
real estimateLogEGFR(real logScr, int age,
                      bool isFemale, bool isAA) {
    var k, alpha: real;
    var f: real;
    f = 4.94;
    if (isFemale) {
        k = -0.357;
        alpha = -0.329;
    } else {
        k = -0.105;
        alpha = -0.411;
    }
    if (logScr < k) {
        f = alpha * (logScr - k);
    } else {
        f = -1.209 * (logScr - k);
    }
    f = f - 0.007 * age;
    if (isFemale)  f = f + 0.017;
    if (isAA)  f = f + 0.148;
    return f;
}
```
Estimating the possible error

```
1 void compareWithNoise(real logScr, real age,
2     bool isFemale, bool isAA) {
3     f1 = estimateLogEGFR(logScr, age, isFemale,isAA);
4     logScr = logScr + uniformRandom(-0.1, 0.1);
5     age = age + uniformRandomInt(-1,1);
6     if (flip(0.01))
7         isFemale = not( isFemale );
8     if (flip(0.01))
9         isAA = not( isAA );
10    f2 = estimateLogEGFR(logScr, age, isFemale,isAA);
11    estimateProbability (f1 - f2 <= 0.1);
12    estimateProbability (f2-f1 <= 0.1);
13 }
```

Can do this by applying Bayesian machine learning
Many programming languages

- Anglican
- Church
- Fun (with Infer.NET)
- IBAL
- Probabilistic Scheme
- BUGS
- HANSEI
- Factorie
- ...

Other Technologies and Topics

• Safe programming languages for the WWW
  ▪ Ur, Yesod in Haskell

• Safe concurrency
  ▪ Part of Rust, and earlier systems, eg Locksmith

• Talk to me to find something that might be close to your research interests
Conclusion

• PL has a great mix of theory and practice
  ▪ Very deep theory
  ▪ But lots of practical applications

• Recent exciting new developments
  ▪ Focus on program correctness (and security)
    - instead of speed
  ▪ Scalability to large programs
  ▪ In greater use in mainstream development