CMSC 430 – Compilers

Fall 2018

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
  - Axiomatic semantics
    - Like a logic

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 ⟧σ = 7
    - σ 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

- $b ::= true \mid false \mid b \lor b \mid b \land b \mid e = e$
- $e ::= 0 \mid 1 \mid \ldots \mid x \mid e + e \mid e * e$
- $s ::= e \mid x := e \mid \text{if } b \text{ then } s \text{ else } s \mid \text{while } b \text{ do } s$

Semantics (booleans):

- $[\![\text{true}]\!]\sigma = \text{true}$
- $[\![b1 \lor b2]\!]\sigma = \begin{cases} \text{true} & \text{if } [\![b1]\!] = \text{true or } [\![b2]\!] = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$
- $[\![e1 = e2]\!]\sigma = \begin{cases} \text{true} & \text{if } [\![e1]\!]\sigma = [\![e2]\!]\sigma \\ \text{false} & \text{otherwise} \end{cases}$

Denotational Semantics cont'd

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

- $[\![x := e]\!]\sigma = \sigma[x \mapsto [\![e]\!]\sigma]$

  (remap x to $[\![e]\!]\sigma$ in σ)

- $[\![\text{if } b \text{ then } s1 \text{ else } s2]\!] = \begin{cases} [\![s1]\!]\sigma & \text{if } [\![b]\!]\sigma = \text{true} \\ [\![s2]\!]\sigma & \text{if } [\![b]\!]\sigma = \text{false} \end{cases}$

Complication: Recursion

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

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

  - 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

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)
    - https://www.microsoft.com/en-us/research/publication/some-domain-theory-and-denotational-semantics-in-coq/

Axiomatic Semantics

Can use as a basic for automated reasoning!

- {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:

$$\frac{\{P \land B\}\ S1\ \{Q\} \qquad \{P \land \neg B\}\ S2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{Q\}}$$

- Example proof (simplified)

$$\frac{\{y>3\}\ x := y\ \{x>3\} \qquad \{\neg(y>3)\}\ x := 4\ \{x>3\}}{\{\}\ \text{if } y>3 \text{ then } x := y \text{ else } x := 4\ \{x>3\}}$$

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

Automated Reasoning

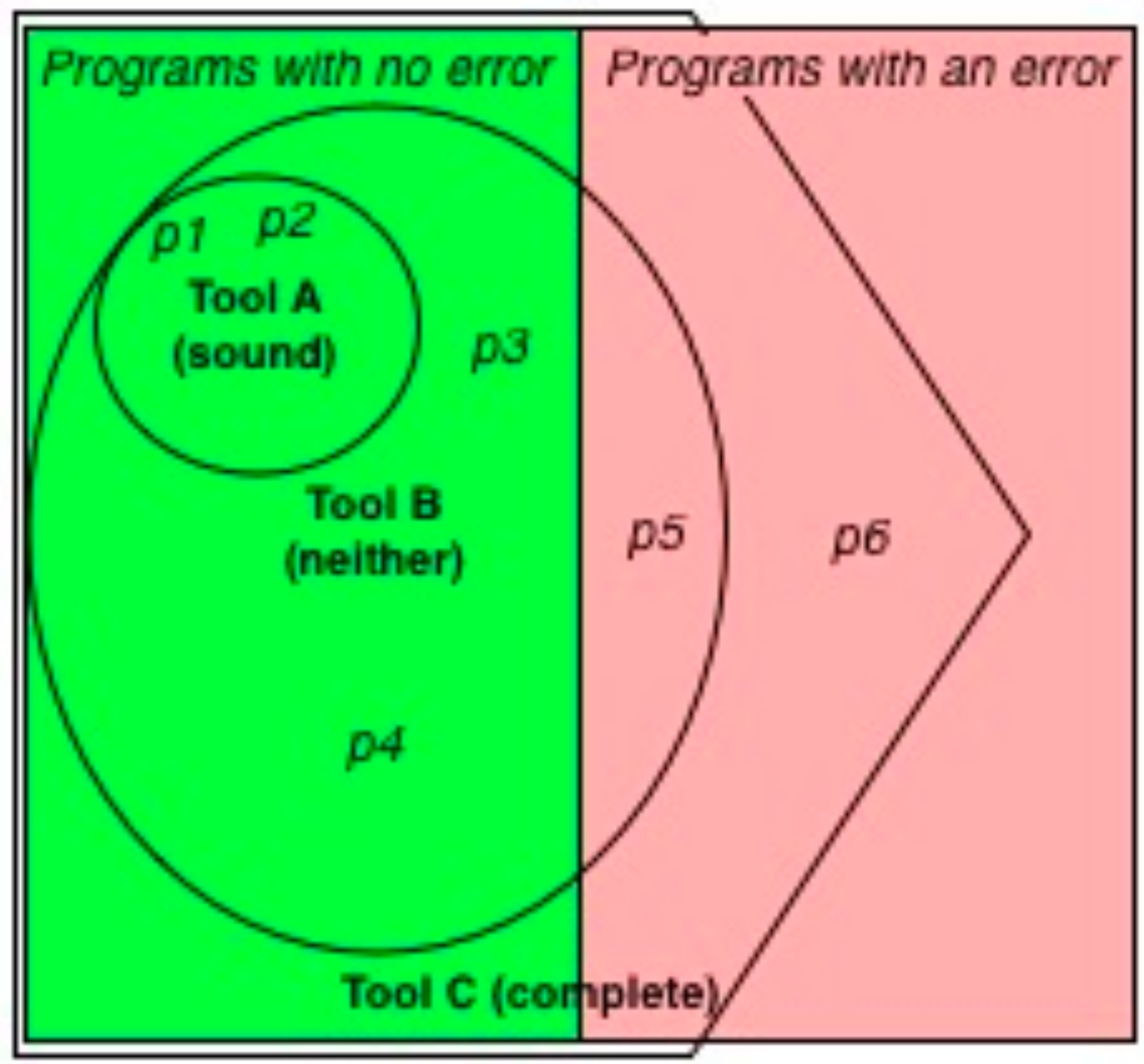
Static Program Analysis

- 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 = “program has no run-time failures”
  - S(P) = true implies P has no run-time failures
- An analysis is **sound** iff
  - for all P, if S(P) = true then P exhibits R
- An analysis is **complete** iff
  - for all P, if P exhibits R then S(P) = true

http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/



Programs with no error
Programs with an error
p1
p2
Tool A
(sound)
p3
Tool B
(neither)
p5
p6
p4
Tool C (complete)

Abstract Interpretation

- Rice's Theorem: Any non-trivial program property is undecidable
  - Never sound *and* complete. Talk about intractable …
- 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
    - I.e., sound (but not complete)
- Seminal papers: Cousot and Cousot, 1977, 1979

Example

e ::= n | e + e

$$\alpha(n) = \begin{cases} - & n < 0 \\ 0 & n = 0 \\ + & n > 0 \end{cases}$$

Abstract semantics:

+	-	0	+
-	-	-	?
0	-	0	+
+	?	+	+

- 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**: $\alpha(\sigma)$ = affine formula over variables in domain of σ, e.g., $x \leq 2y + 5$
    - where σ is a state mapping variables to numbers
    - *relational* domain
- Abstract semantics for standard PL constructs
  - Assignments, sequences, loops, conditionals, etc.

Applications: Abstract Interpretation

- ASTREE (ENS, others) http://www.astree.ens.fr/
  - Detects all possible runtime failures (divide by zero, null pointer deref, array bounds) on embedded code
  - Used regularly on Airbus avionics software
- RacerD (Facebook) https://fbinfer.com/docs/racerd.html
  - Uses Infer.AI framework to reason about memory and pointer use in Java, C, Objective C programs
  - In particular, looks for data races
  - Neither sound nor complete, but very effective

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

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 = α;   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; }

Relating SymExe and AbsInterp

- Symbolic execution is a kind of abstract interpretation, where
  - 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 permits 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
    - https://www.microsoft.com/en-us/security-risk-detection/
- KLEE (Imperial), Angr (UCSB), Triton (Inria), ...
  - Research systems used to enforce security specifications, find vulnerabilities, explore configuration spaces, and more

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/

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

$e ::= x \mid n \mid \lambda x{:}\tau.e \mid e\ e$

$\tau ::= \text{int} \mid \tau \rightarrow \tau$

$A ::= \bullet \mid A, x{:}\tau$

$A \vdash e : \tau$

in type environment A, expression e has type τ

$$\frac{}{A \vdash n : \text{int}}$$

$$\frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$$

$$\frac{A, \tau{:}x \vdash e : \tau'}{A \vdash \lambda x{:}\tau.e : \tau \rightarrow \tau'}$$

$$\frac{A \vdash e1 : \tau \rightarrow \tau' \qquad A \vdash e2 : \tau}{A \vdash e1\ e2 : \tau'}$$

Type Safety

- If $\bullet \vdash e : \tau$ then either
  - there exists a value v of type τ such that $e \to^* v$, or
  - e diverges (doesn't terminate)
- Corollary: e will never get "stuck"
  - never evaluates to a normal form that is not a value
  - i.e., sound (but not complete)
- 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?
  - Introduce type vars, constraints: solve

$$\frac{A, x{:}\alpha \vdash e : t' \qquad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \to t'}$$

$$\frac{A \vdash e1 : t1 \qquad A \vdash e2 : t2 \qquad \boxed{t1 = t2 \to \beta} \qquad \beta \text{ fresh}}{A \vdash e1\ e2 : \beta}$$

"Generated" constraint

Scaling up

- Type inference works well in limited settings
  - Hindley-Milner (polymorphic) 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, …
    - Full polymorphic type inference (System F) undecidable
- Connection:
  - Whole-program type inference = static analysis

Types, Types, Types, Oh my!

- Sums $\tau 1 + \tau 2$
- Products $\tau 1 * \tau 2$
- Unions $\tau 1 \cup \tau 2$
- Intersections $\tau 1 \cap \tau 2$
- References τ ref
- Recursive types $\mu\alpha.\tau$
- Universals $\forall\alpha.\tau$
- Existentials $\exists\alpha.\tau$
- Dependent functions $\Pi x{:}\tau 1.\tau 2$
- Dependent products $\Sigma x{:}\tau 1.\tau 2$

α list =

$\forall\alpha.\mu\beta.\text{unit}+(\alpha * \beta)$

Refinement Types

Back to 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 (next)
- Present in several languages
  - Liquid Haskell, F*

Dependent Types

- Useful for expressing properties of programs
  - [1;2;3] : int list
  - [1;2;3] : int **3** list
  - append: ‘a **n** list -> ‘a **m** list -> ‘a (**m+n**) 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

Dependent Types for Verification

- Dependent types form a practical foundation for the concept of ***propositions as types***
  - A type = a logical proposition
  - A program P with a type T = proof of the proposition corresponding to T
  - So: if P : T then proof of proposition is correct
    - Type checking is proof checking!
- Foundation of proof systems in Coq and Agda
  - coq.inria.fr
  - http://wiki.portal.chalmers.se/agda/pmwiki.php

$$\frac{M : A \qquad N : B}{\langle M, N \rangle : A \times B}\times\text{-I} \qquad \frac{L : A \times B}{\pi_1\, L : A}\times\text{-E}_1 \qquad \frac{L : A \times B}{\pi_2\, L : B}\times\text{-E}_2$$

$$\frac{\begin{matrix}[x : A]^x \\ \vdots \\ N : B\end{matrix}}{\lambda x.\, N : A \to B}\to\text{-I}^x \qquad \frac{L : A \to B \qquad M : A}{L\, M : B}\to\text{-E}$$

Figure 5. Alonzo Church (1935) — Lambda Calculus

$$\frac{\dfrac{\dfrac{[z : B \times A]^z}{\pi_2\, z : A}\times\text{-E}_2 \qquad \dfrac{[z : B \times A]^z}{\pi_1\, z : B}\times\text{-E}_1}{\langle \pi_2\, z, \pi_1\, z \rangle : A \times B}\times\text{-I}}{\lambda z.\, \langle \pi_2\, z, \pi_1\, z \rangle : (B \times A) \to (A \times B)}\to\text{-I}^z$$

Figure 6. A program

https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf

Verification Systems

- Verified software
  - CompCert compiler
    - developed and proved correct in Coq
  - Everest TLS infrastructure
    - developed and proved correct in F*
  - Liquid Haskell (smaller scale)
- Verified mathematical developments (many)
  - E.g., encode type system, semantics, etc. and perform the proof in Coq, LH, Agda, etc.

Applications: Solver-aided languages

- Dafny (Microsoft)
  - Can perform deep reasoning about programs
    - Array out-of-bounds, null pointer errors, failure to satisfy internal invariants; based Hoare logic
  - 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)
  - Project Everest: https://www.microsoft.com/en-us/research/project/project-everest-verified-secure-implementations-https-ecosystem/

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

Rust

- Actively developed by Mozilla
- *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

```rust
// This function takes ownership of the heap allocated memory
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
}
```

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

Dynamic Enforcement

- *Implement* "monitoring" semantics via literally, via instrumentation
  - Accepts more (all!) programs. Defers error checks to run-time (which adds overhead)
- Many examples
  - Phosphor for Java (taint analysis)
  - RoadRunner for Java (data race detector): http://www.cs.williams.edu/~freund/rr/
  - Recent work by Nguyen and Van Horn: Dynamically monitor size-change, which correlates with termination
    - Amazing: Flag non-terminating program at run-time !

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 H to L?
- *Direct flows*

```
h := l;
x := l; y := x; h := y;
```

- *Implicit flows*

```
h := h mod 2;
l := 0;
if h == 1 then l := 1 else skip
```

  – The low order bit of 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, ≤)
    - S is the set of labels
    - s1 ≤ s2 if s1 allowed to flow to s2
      - » e.g., let f (x:s2) = ... in f (y:s1)
    - confidentiality: s1 is "less secret" than s2
    - integrity: s1 is "more trusted" than s2

Preventing Explicit Flows by Typing

- Build a type system that rejects programs with bad explicit flows
  - e ::= x | e op e | n
  - c ::= skip | x := e | if e then c1 else c2 | while e do c
  - t ::= int S   *types tagged with security level*
  - A : vars → t

Preventing Explicit Flows (cont'd)

$\boxed{A \vdash x : t}$

$$\frac{}{A \vdash x : A(x)} \qquad \frac{}{A \vdash n : \text{int } S} \qquad \frac{A \vdash e1 : \text{int } S1 \quad A \vdash e2 : \text{int } S2}{A \vdash e1 \text{ op } e2 : \text{int } (S1 \sqcup S2)}$$

$\boxed{A \vdash c}$

$$\frac{}{A \vdash \text{skip}} \qquad \frac{A \vdash e : \text{int } S \quad A(x) = \text{int } S' \quad S \leq S'}{A \vdash x := e}$$

$$\frac{A \vdash e : \text{int } S \quad A \vdash c1 \quad A \vdash c2}{A \vdash \text{if } e \text{ then } c1 \text{ else } c2} \qquad \frac{A \vdash e : \text{int } S \quad A \vdash c}{A \vdash \text{while } (e) \text{ do } c}$$

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

Implicit Flows

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

```
if h > 0 then l := 1 else l := 0;
```

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

```
l := 0;
if h > 0 then l := 1 else skip;
```

  – Fortunately, we are doing static analysis, so we can look at *both* branches
  – Much harder in a dynamic setting!

Preventing Implicit Flows (cont'd)

$A \vdash x : A(x)$ *(same as before)*

$$\frac{}{A \vdash x : A(x)} \qquad \frac{}{A \vdash n : \text{int } S} \qquad \frac{A \vdash e1 : \text{int } S1 \quad A \vdash e2 : \text{int } S2}{A \vdash e1 \text{ op } e2 : \text{int } (S1 \sqcup S2)}$$

$A, S \vdash c$

$$\frac{}{A, S_{pc} \vdash \text{skip}} \qquad \frac{A \vdash e : \text{int } S \quad A(x) = \text{int } S' \quad S \sqcup S_{pc} \leq S'}{A, S_{pc} \vdash x := e}$$

$$\frac{A \vdash e : \text{int } S \qquad A, S_{pc} \sqcup S \vdash c1 \quad A, S_{pc} \sqcup S \vdash c2}{A, S_{pc} \vdash \text{if } e \text{ then } c1 \text{ else } c2} \qquad \frac{A \vdash e : \text{int } S \quad A, S_{pc} \sqcup S \vdash c}{A, S_{pc} \vdash \text{while } (e) \text{ do } c}$$

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

Application to Haskell

- LIO (Labeled IO)
  - Only reference cells are labeled directly
  - Current expression protected by an ambient "current label"
  - Attempts at IO are checked against the current label
- LWeb: Extension of LIO to web applications
  - Need to protect data stored in DB properly

https://www.cs.umd.edu/~mwh/papers/parker19lweb.html

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

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 — many examples today
  - Parallel work
    - Tobin-Hochstadt and Felleisen. Interlanguage Migration. DLS 2006.
    - Siek and Taha. Gradual Typing for Objects. ECOOP 2007.
  - Focuses on providing sister typed and untyped languages
  - Allows interaction between typed and untyped modules

Gradual Typing Enforcement

- Static types can be used as a compile-time bug-finder, with no run-time effect
  - Relies on underlying language semantics
- … or as a way of designating where type checking should take place
  - I.e., at the boundary between typed/untyped code
  - Creates interesting complication for higher-values based between typed/untyped code
    - Whom to blame when something goes wrong?

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.

Gradual Typing Examples

- Flow (Facebook), Typescript (Microsoft)
  - https://flow.org/
  - https://www.typescriptlang.org/
- Dart (Google)
  - https://www.dartlang.org/dart-2
- Typed Racket (academic)
  - https://docs.racket-lang.org/ts-guide/

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
  - Like gradual (migratory) typing, but no extra checks
- Mechanized proof of blame property in Coq
  - Failures can be blamed on unchecked code
    - Specially designated checked regions of code are internally sound
    - So: Make as many of these as possible

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 ϕ is faster than writing P

correctness: when proving ϕ is easier than proving P

Contracts

- Assertions about inputs/outputs to functions
  - In a sense, a kind of refinement type
- Connection to types brings in connections to automated reasoning
  - Prove contracts will always hold (so-called *contract verification*), and remove those that do
  - Enforce those that remain similarly to gradual typing
- Interesting work here at UMD by David Van Horn and Phil Nguyen

Preparing your language for synthesis

Extend the language with two constructs

spec:

```
int foo (int x) {
    return x + x;
}
```

$\phi(x, y) : y = \texttt{foo}(x)$

sketch:

```
int bar (int x) implements foo {
    return x << ??;
}
```

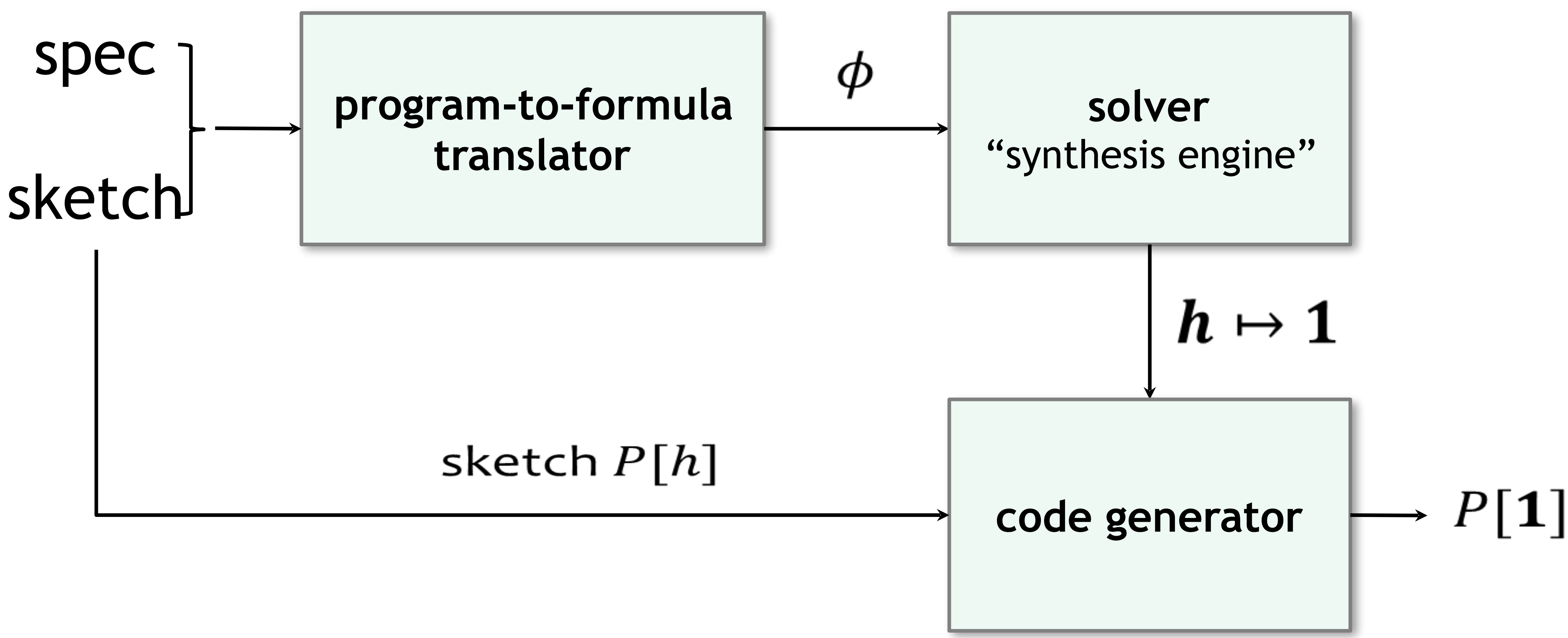
?? substituted with an int constant meeting ϕ

result:

```
int bar (int x) implements foo {
    return x << 1;
}
```

instead of **`implements`**, assertions over safety properties can be used

Synthesis from partial programs



Examples: Sketch (C), JSketch (Java), Flashfill (Excel!)

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

```
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

```
void compareWithNoise(real logScr, real age,
                      bool isFemale, bool isAA) {
  f1 = estimateLogEGFR(logScr, age, isFemale,isAA);
  logScr = logScr + uniformRandom(-0.1, 0.1);
  age = age + uniformRandomInt(-1,1);
  if (flip(0.01))
    isFemale = not( isFemale );
  if (flip(0.01))
    isAA = not( isAA );
  f2 = estimateLogEGFR(logScr, age, isFemale,isAA);
  estimateProbability (f1 - f2 <= 0.1);
  estimateProbability (f2-f1 <= 0.1);
}
```

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

- Lots of other connections between PL and ML
  - Automatic differentiation — better languages than Tensorflow
  - ML for program analysis directly, and for prioritizing alarms
- Performance/feature enhancement
  - Better run-times, GCs, language features, compilers (auto-parallelization!),
- Debugging … oh my!

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