EEÇS 390 – Special Topics

Program Analysis

2/18/24

The Halting Problem (Short Short Version)

- Can we determine through some fancy analysis whether a unary function `f` halts on some input `x`?
- No! Proof by contradiction:
  - Suppose there exists some function `halts(f, x)` that returns whether `f(x)` halts
  - Define the following:

```
def paradox(g):
    if halts(g, g):
        while True: pass
```

  - What does `paradox(paradox)` do?
    - Suppose `paradox(paradox)` halts. Then by assumption, `halts(paradox, paradox)` returns true. But `paradox(paradox)` calls `halts(paradox, paradox)`, and if it returns true, then `paradox(paradox)` infinitely loops.
    - Suppose `paradox(paradox)` does not halt. Then `halts(paradox, paradox)` returns false. But if that call returns false, then `paradox(paradox)` immediately halts.

Rice's Theorem

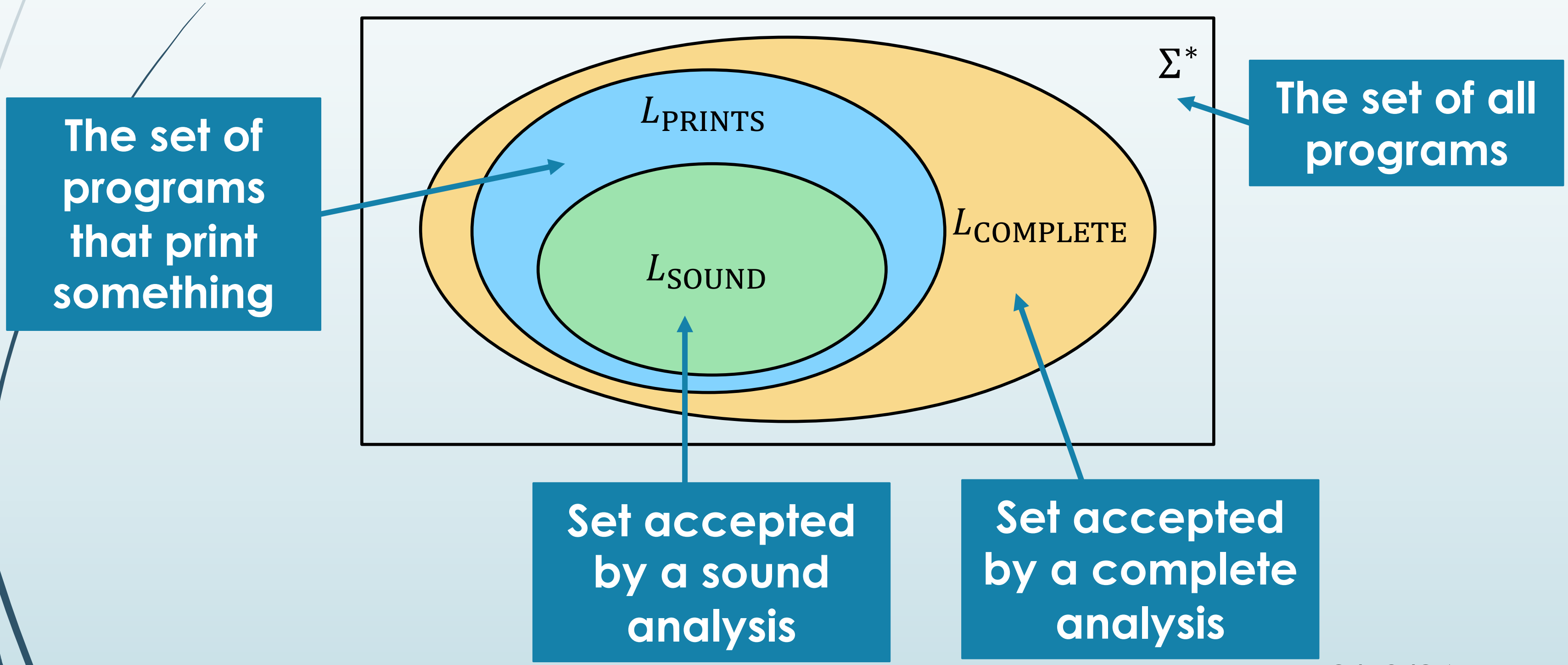
- All nontrivial program analysis is ***undecidable*** (cannot be solved by an algorithm)
- Example: Does `f(x)` print something to standard out?
  - Suppose `prints(f, x)` correctly determines this
  - Then given a function `f` and input `x`, do the following:
    - Remove all print calls from `f` and every function it calls
    - Replace all implicit and explicit returns from `f` with a print call
    - Then run `prints()` on the result and `x`; the answer tells us whether or not `f(x)` halts!
- Any analysis algorithm gets the wrong answer at least some of the time

Soundness vs. Completeness

- Goal of program analysis is often determining whether a program is error-free
- An analysis is ***sound*** if it <u>only</u> accepts correct code – it rejects <u>every</u> erroneous program
  - Suffers from false negatives – sometimes correct code is rejected
- An analysis is ***complete*** if it accepts <u>all</u> correct code – it <u>only</u> rejects incorrect code
  - Suffers from false positives – sometimes incorrect code is accepted
- An analysis may be sound or complete, but it cannot be both
  - It can be neither, but that's undesirable

Soundness vs. Completeness

- An analysis is ***sound*** if it only accepts correct code – it rejects every erroneous program
- An analysis is ***complete*** if it accepts all correct code – it only rejects incorrect code



Examples

- Compiler analyses (e.g. static type checking, control/data-flow analysis, etc.) are typically sound
  - Example: Checking for a return in a non-void function

```
int foo(int x) {
  if (x < 0) return 3;
  if (x >= 0) return 42;
}
```

- Analyses that rely on running a program are designed to be complete (or both unsound and incomplete)
  - Cannot run on every input, and cannot run for an unbounded amount of time
  - Example: Regression testing (e.g. an autograder) – correct programs pass the tests, but some incorrect ones do too

Semantic vs. Syntactic Properties

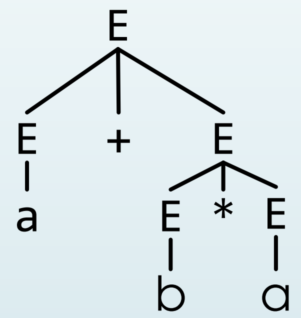
- Questions about the behavior (***semantics***) of a program are undecidable
  - Example: Does program M print a 1 when run on any input?
- Questions about the structure (***syntax***) of a program are decidable
  - Example: Does program M have a print statement?
- Program analysis often approximates a semantic question with a syntactic one
- Relevant details:
  - What is the syntactic structure of a program?
  - How does the behavior of a program relate to its structure?
  - How do we approximate the behavior from the syntax?

Review: Derivation Trees

1) $E \rightarrow E + E$
2) $E \rightarrow E * E$
3) $E \rightarrow a$
4) $E \rightarrow b$

- Recall that a context-free grammar defines a recursive process for matching a string
- A derivation is sequence of rule applications, starting with the start variable and ending with a string of terminals
- Since the rules are recursive, we end up with a tree structure from applying them

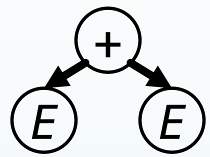
$E \rightarrow E + E$	by rule (1)
$\rightarrow E + E * E$	by rule (2) on 2nd E
$\rightarrow a + E * E$	by rule (3) on 1st E
$\rightarrow a + b * E$	by rule (4) on 1st E
$\rightarrow a + b * a$	by rule (3)



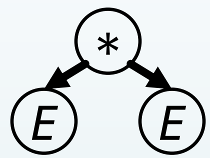
Recall that this grammar is ambiguous, which means that the same string may have multiple derivation trees

Abstract Syntax Trees

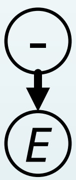
- A grammar's recursive rules can be used to define a related tree structure with constructs as internal nodes
  - $E \rightarrow E + E$



  - $E \rightarrow E * E$

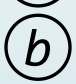


  - $E \rightarrow -E$



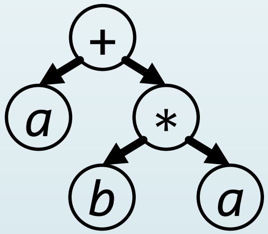
  - $E \rightarrow a$

  - $E \rightarrow b$



Example: $a + b * a$

$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E + E * E \\
&\rightarrow a + E * E \\
&\rightarrow a + b * E \\
&\rightarrow a + b * a
\end{aligned}
$$



- This is an ***abstract syntax tree (AST)***, and it is the output of most parsers

Operational Semantics

- We can formally define the semantics of a program in terms of its structure
- The system we consider is called ***big-step operational semantics***
- The ***state*** of a program is a mapping of variables to values
  - Denoted by σ, and the value of variable v is $\sigma(v)$
- A ***transition*** represents the result of a computation
  - Denoted by $\langle s, \sigma \rangle \Downarrow \langle u, \sigma' \rangle$, which means that s, when evaluated in the state σ, produces a value u and new state σ'

Rules and Axioms

- A ***transition rule*** is an implication – assuming that a set of ***premises*** all hold, then a ***conclusion*** also holds

$$\frac{premise_1 \quad \ldots \quad premise_k}{conclusion}$$

- An ***axiom*** is a result that holds unconditionally

$$\frac{}{conclusion}$$

- A proof of a result is a derivation, starting from axioms and applying transition rules until we reach the result as a conclusion

These have the same structure as rules in ***natural deduction***, which you may have seen in discrete mathematics

Rules and Axioms

- Example: Axioms for a number literal and a variable:

$$\frac{}{\langle n, \sigma \rangle \Downarrow \langle n, \sigma \rangle} \qquad \frac{}{\langle v, \sigma \rangle \Downarrow \langle \sigma(v), \sigma \rangle}$$

  - A number literal n evaluated in a state σ produces the value n and same state σ
  - A variable v evaluated in a state σ produces the value it is mapped to by σ and the same state
- Example: Rule for addition (with possible side effects):

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle n_1, \sigma_1 \rangle \qquad \langle a_2, \sigma_1 \rangle \Downarrow \langle n_2, \sigma_2 \rangle}{\langle (a_1 + a_2), \sigma \rangle \Downarrow \langle n, \sigma_2 \rangle} \quad \text{where } n = n_1 + n_2$$

  - If a_1 evaluated in state σ results in value n_1 and state σ_1, and a_2 evaluated in σ_1 results in value n_2 and state σ_2, then $(a_1 + a_2)$ evaluated in σ results in value $n = n_1 + n_2$ and state σ_2

Derivations and Evaluation

- A ***derivation tree*** for an expression is an application of rules and axioms where the expression is the root, and the leaves are all axioms
- Example: Derivation for $((x+3)*(y-5))$ in a state σ where $\sigma(x)=1$ and $\sigma(y)=2$:

$$\dfrac{\dfrac{\dfrac{}{\langle x,\sigma\rangle \Downarrow \langle 1,\sigma\rangle} \quad \dfrac{}{\langle 3,\sigma\rangle \Downarrow \langle 3,\sigma\rangle}}{\langle (x+3),\sigma\rangle \Downarrow \langle 4,\sigma\rangle} \qquad \dfrac{\dfrac{}{\langle y,\sigma\rangle \Downarrow \langle 2,\sigma\rangle} \quad \dfrac{}{\langle 5,\sigma\rangle \Downarrow \langle 5,\sigma\rangle}}{\langle (y-5),\sigma\rangle \Downarrow \langle -3,\sigma\rangle}}{\big\langle \big((x+3)*(y-5)\big),\sigma\big\rangle \Downarrow \langle -12,\sigma\rangle}$$

- This corresponds to how an interpreter works
  - It keeps track of variable values in a data structure (***environment***)
  - It evaluates a compound expression by recursively evaluating the components and combining results, until it reaches leaf expressions

Interpretation

- Typically, object orientation is used to represent the AST

```
struct ASTNode {
  virtual Value eval(Environment *env) = 0;
};

struct PlusNode : ASTNode {
  Value eval(Environment *env) override {
    return lhs->eval(env) + rhs->eval(env);
  }
  ASTNode *lhs, *rhs;
};
```

- The code for evaluation is just a direct translation of the transition rule for the expression

Syntactic vs. Semantic Correctness

- Recall that a program can be syntactically correct but not make semantic sense
  - Example in Java: `true + 3`

- We need a program analysis that catches erroneous code like this
  - In particular, we want a sound analysis that catches all errors, without running the program

- Since we can't just run the program, we need an analysis that computes some abstraction of the program's semantics

Abstract Interpretation

- ***Abstract interpretation*** is a method for computing an abstraction over a program's semantics
- Rather than computing a concrete value for each expression, we compute an abstract value that represents a set of concrete values the expression may take on
- Examples:
  - **Types:** concrete integer values ⟺ the `int` type
  - **Memory locations:** concrete object addresses ⟺ the line of code that allocates the objects
- ***Abstraction*** maps a set of concrete values to an abstract value
- ***Concretization*** maps an abstract value to a set of concrete values

Example: Type Systems

- Types are an abstraction over concrete values
- Type systems directly map concrete (operational) semantics to abstract semantics over types
- Concrete semantics: ***State*** σ maps variable v to value $\sigma(v)$
  - Abstract semantics: ***Type context*** Γ maps variable v to type $\Gamma(v)$
  - Example: $\sigma(v) = n \Leftrightarrow \Gamma(v) = Int$
- Concrete semantics: ***Transition*** $\langle s, \sigma \rangle \Downarrow \langle u, \sigma' \rangle$ means that s evaluated in the state σ produces the value u and state σ'
  - Abstract semantics: ***Type judgment*** $\Gamma \vdash s : T$ means that s typed in the type context Γ has type T
- Type checking is done with rules and axioms corresponding to those in operational semantics

Typing Rules

- Example: Axioms for a number literal and a variable:

$$\frac{}{\Gamma \vdash n \ : \ Int} \qquad \frac{}{\Gamma \vdash v \ : \ \Gamma(v)}$$

  - A number literal n typed in a context Γ has the type Int
  - A variable v typed in a context Γ has the type $\Gamma(v)$
- Example: Rule for addition:

$$\frac{\Gamma \ \vdash a_1 \ : \ Int \qquad \Gamma \ \vdash a_2 \ : \ Int}{\Gamma \ \vdash \ (a_1 + a_2) \ : \ Int}$$

  - If a_1 typed in context Γ has the type Int, and a_2 typed in context Γ has the type Int, then $(a_1 + a_2)$ typed in the context Γ has the type Int
  - This rule cannot be applied if a_1 has type $Bool$

Type Checking

- Type analysis applies these rules to the AST, performing abstract interpretation on the program structure

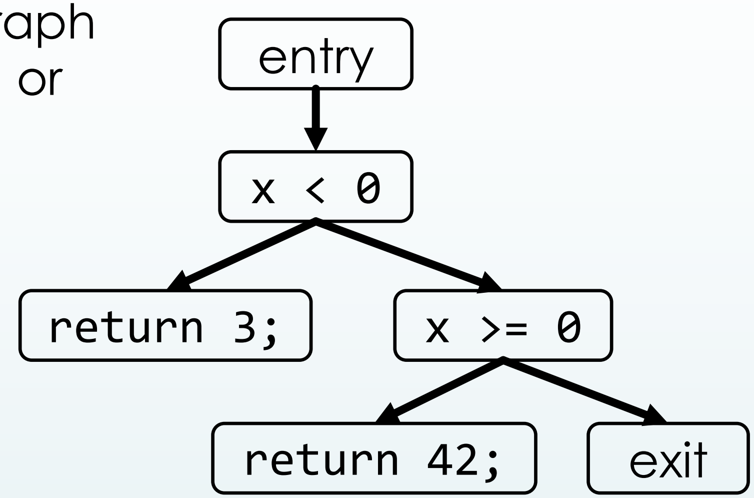
```
struct ASTNode {
  virtual Type check(Context *ctx) = 0;
};

Type PlusNode::check(Context *ctx) {
  if (Type t = lhs->check(ctx); t != Int)
    error("LHS of + has invalid type "s
          + t.str());
  if (Type t = rhs->check(ctx); t != Int)
    error("RHS of + has invalid type "s
          + t.str());
  return Int;
}
```

Data/Control-Flow Analysis

- Many analyses work on a graph structure of the flow of data or control in a program
- Example:

```
int foo(int x) {
  if (x < 0) return 3;
  if (x >= 0) return 42;
}
```



- These analyses use graph algorithms to reason about a program
- Example: Checking for a return in a non-void function
  - Is there a path from function entry to exit that does not go through a return statement?

Summary

- Program analysis approximates the behavior of a program, since computing an exact solution is undecidable
- ***Full Employment Theorem for Compiler Writers:*** There will always be new and better analyses to be developed
- Working on program analysis (and programming languages in general) requires having a background in both theory and software
  - We need mathematical tools to understand, develop, and reason about analyses (hello, EECS 376 and 490!)
  - We need software tools (data structures, object-oriented programming, etc.) to implement analyses