Lessons From Model Checking

- To find bugs, we need specifications
- To convert a program into a model, we need predicates/invariants and a theorem prover.
- Simple algorithms (e.g., depth first search, pushing facts along a CFG) can work well

The Big Lesson

To reason about a program (= “is it doing the right thing? the wrong thing?”), we must understand what the program means!

Semantics = “Meaning”

Syntax

- **Concrete syntax**: The rules by which programs can be expressed as strings of characters
  - Keywords, identifiers, statement separators vs. terminators / comments / indentation
- **Concrete syntax in practice**: For readability, speed, effectiveness of error recovery, clarity of error messages
- Well understood principles
  - Use finite automata and context-free grammars
  - Automatic lexer/parser generators

Semantics

We will focus on the basics and what we need for analysis. See CSCI 5535 for a more thorough treatment.
Abstract Syntax

• We ignore parsing issues and study programs given as abstract syntax trees.

• Abstract syntax tree is (a subset of) the parse tree of the program
  - Ignores issues like comment conventions
  - More convenient for formal and algorithmic manipulation
  - Research papers consider ASTs

Syntactic Entities

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>integer constants</td>
<td>n</td>
</tr>
<tr>
<td>boolean constants</td>
<td>true, false</td>
</tr>
<tr>
<td>variables</td>
<td>x, y, ...</td>
</tr>
<tr>
<td>arithmetic expressions</td>
<td>a</td>
</tr>
<tr>
<td>boolean expressions</td>
<td>b</td>
</tr>
<tr>
<td>commands</td>
<td>s</td>
</tr>
</tbody>
</table>

Set

Meta-Variable

Here, these also encode the types.

Abstract Syntax (AExp)

• Arithmetic expressions (AExp)
  \[ a ::= n \quad \text{for } n \in \mathbb{Z} \]
  \[ | x \quad \text{for } x \in \text{Var} \]
  \[ | a_1 + a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]
  \[ | a_1 - a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]
  \[ | a_1 \times a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]

• Observations?

Abstract Syntax (AExp)

• Arithmetic expressions (AExp)
  \[ a ::= n \quad \text{for } n \in \mathbb{Z} \]
  \[ | x \quad \text{for } x \in \text{Var} \]
  \[ | a_1 + a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]
  \[ | a_1 - a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]
  \[ | a_1 \times a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]

• Notes:
  - Variables are not declared
  - All variables have integer type
  - No side-effects (in expressions)

Abstract Syntax (BExp)

• Boolean expressions (BExp)
  \[ b ::= \text{true} \]
  \[ | \text{false} \]
  \[ | a_1 = a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]
  \[ | a_1 \leq a_2 \quad \text{for } a_1, a_2 \in \text{AExp} \]
  \[ | \neg b \quad \text{for } b \in \text{BExp} \]
  \[ | b_1 \land b_2 \quad \text{for } b_1, b_2 \in \text{BExp} \]
  \[ | b_1 \lor b_2 \quad \text{for } b_1, b_2 \in \text{BExp} \]

“Boolean”

• George Boole
  - 1815-1864
Abstract Syntax (Stmt)

• Statements (Stmt)

\[ s ::= \text{skip} \]
\[ | x := a \quad \text{for } x \in \text{Var and } a \in \text{AExp} \]
\[ | s_1 ; s_2 \quad \text{for } s_1, s_2 \in \text{Stmt} \]
\[ | \text{if } b \text{ then } s_1 \text{ else } s_2 \quad \text{for } s_1, s_2 \in \text{Stmt}, b \in \text{BExp} \]
\[ | \text{while } b \text{ do } s \quad \text{for } s \in \text{Stmt}, b \in \text{BExp} \]

• Observations?

Notes:
- The typing rules have been embedded in the syntax
- Other parts are not context-free and need to be checked separately (e.g., all variables are declared)
- Commands contain all the side-effects in the language

Now What?

• Questions to answer:
  - What is the "meaning" of a given expression/command?
  - How would we go about evaluating expressions and commands?
  - How are the evaluator and the meaning related?

14.20.2 Execution of try-catch-finally

• A try statement with a finally block is executed by first executing the try block. Then there is a choice:
  - If execution of the try block completes normally, then the finally block is executed, and then there is a choice:
    - If the finally block completes normally, then the try statement completes normally.
    - If the finally block completes abruptly for reason \( S \), then the try statement completes abruptly.
  - If execution of the try block completes abruptly because of a throw of a value \( V \), then there is a choice:
    - If the run-time type of \( V \) is assignable to the parameter of any catch clause of the try statement, then the first (leftmost) such catch clause is selected. The value \( V \) is assigned to the parameter of the selected catch clause, and the block of that catch clause is executed. Then there is a choice:
      • If the catch block completes normally, then the finally block is executed. Then there is a choice:
        • If the finally block completes normally, then the try statement completes normally.
        • If the finally block completes abruptly for reason \( S \), then the try statement completes abruptly.
    - If the run-time type of \( V \) is not assignable to the parameter of any catch clause of the try statement, then the try statement completes abruptly.

An Operational Semantics

• Specifies how expressions and commands should be evaluated
  - Depending on the form of the expression
    - 0, 1, 2, ... don’t evaluate any further.
    - They are normal forms or values.
    - \( e_1 + e_2 \) is evaluated by first evaluating \( e_1 \) to \( n_1 \), then evaluating \( e_2 \) to \( n_2 \).
    - The result is the literal representing \( n_1 + n_2 \).

Operational semantics abstracts the execution of a concrete interpreter
Operational Semantics

- The meanings of expressions depend on the values of variables
  - What does \(x + 5\) mean? It depends on \(x\)!
- The value of variables at a given moment is abstracted as a function from Var to \(\mathbb{Z}\) (a state)
  - If \(x = 8\) in our state, we expect \(x + 5\) to mean 13
- The set of all states is \(\Sigma = \text{Var} \rightarrow \mathbb{Z}\)
- We shall use \(\sigma\) to range over \(\Sigma\)
  - \(\sigma\), a state, maps variables to values

Program State

- The state \(\sigma\) is somewhat like "memory"
  - It holds the current values of all variables
  - Formally, \(\sigma : \text{Var} \rightarrow \mathbb{Z}\)

Notation: Evaluation Judgment

- We write: \(<a, \sigma> \Downarrow n\)
  - To mean that \(a\) evaluates to \(n\) in state \(\sigma\).
  - This is a judgment. It asserts a relation between \(a\), \(\sigma\) and \(n\).
  - In this case, we can view \(\Downarrow\) as a function with two arguments (\(a\) and \(\sigma\)).

Notation: Rules of Inference

- We express the evaluation rules as rules of inference for our judgment
  - called the derivation rules for the judgment
  - also called the evaluation rules (for operational semantics)
- In general, we have one rule for each language construct:
  \[
  \frac{<a_1, \sigma> \Downarrow n_1 \quad <a_2, \sigma> \Downarrow n_2}{<a_1 + a_2, \sigma> \Downarrow n_1 + n_2}
  \]
  This is the only rule for \(a_1 + a_2\)

Rules of Inference

Hypothesis\(_1\) ... Hypothesis\(_n\)

Conclusion

- \(A\) is true \quad \(B\) is true
- \(A \land B\) is true

- For any given proof system, a finite number of rules of inference (or schema) are listed somewhere

Evaluation Rules (for AExp)

- \(<n, \sigma> \Downarrow n\)
- \(<x, \sigma> \Downarrow\)
- \(<a_1, \sigma> \Downarrow n_1 \quad <a_2, \sigma> \Downarrow n_2\)
  \[
  \frac{<a_1 + a_2, \sigma> \Downarrow n_1 + n_2}{<a_1, \sigma> \Downarrow n_1 \quad <a_2, \sigma> \Downarrow n_2}
  \]
  \[
  \frac{<a_1 - a_2, \sigma> \Downarrow}{<a_1, \sigma> \Downarrow n_1 \quad <a_2, \sigma> \Downarrow n_2}
  \]
- This is called structural operational semantics
  - rules defined based on the structure of the expression
  - These rules do not impose an order of evaluation!
Evaluation Rules (for AExp)

- \( <n, \sigma> \Downarrow n \)
- \( <x, \sigma> \Downarrow \sigma(x) \)
- \( <a_1, \sigma> \Downarrow n_1 \) and \( <a_2, \sigma> \Downarrow n_2 \)
- \( <a_1 + a_2, \sigma> \Downarrow n_1 + n_2 \)
- \( <a_1 - a_2, \sigma> \Downarrow n_1 - n_2 \)
- \( <a_1 \ast a_2, \sigma> \Downarrow n_1 \ast n_2 \)

This is called structural operational semantics:
- rules defined based on the structure of the expression
- These rules do not impose an order of evaluation!

Derivation

- Apply inferences rules and put in a tree
- Provides proof of a judgment
- "witnesses an element in the relation"
- Conclusion is at the bottom and the leaves at the top are axioms (rules with no hypotheses)

Derivation (Example)

- "Show that 3 + (4 - 5) evaluates to 2"

\[ <3 + (4 - 5), \sigma> \Downarrow 2 \]

Evaluation Rules (for BExp)

How to Read the Rules

- Forward (top-down) = inference rules
  - if we know that the hypothesis judgments hold then we can infer that the conclusion judgment also holds
  - If we know that \( <a_1, \sigma> \Downarrow 5 \) and \( <a_2, \sigma> \Downarrow 7 \), then we can infer that \( <a_1 + a_2, \sigma> \Downarrow 12 \)

How to Read the Rules

- Backward (bottom-up) = evaluation rules
  - Suppose we want to evaluate \( a_1 + a_2 \), i.e., find \( n \) s.t. \( a_1 + a_2 \Downarrow n \) is derivable using the previous rules
  - By inspection of the rules we notice that the last step in the derivation of \( a_1 + a_2 \Downarrow n \) must be the addition rule
  - the other rules have conclusions that would not match \( a_1 + a_2 \Downarrow n \)
Syntax-Directed Evaluation

- Thus we must find \( n_1 \) and \( n_2 \) such that \( a_1 \Downarrow n_1 \) and \( a_2 \Downarrow n_2 \) are derivable
  - This is done recursively
- If there is exactly one rule for each kind of expression we say that the rules are syntax-directed
  - At each step at most one rule applies
  - This allows a simple evaluation procedure as above (recursive tree-walk)

Where are we?

- Defined a big-step operational semantics for arithmetic and boolean expressions.
  - What's "big"?

What about statements?

Small-Step Operational Semantics

- We define a transition relation \( \langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle \)
  - "\( s \) steps to \( s' \) via an atomic rewrite step" 
  - Evaluation terminates when the program has been rewritten to a terminal program
    - one from which we cannot make further progress
- The terminal program is "skip"
  - As long as the statement is not "skip" we can make further progress
    - some statements never reduce to skip (e.g., "while true do skip")

Small-Step Operational Semantics

- Evaluation of a statement as a sequence of rewrites:

  \[
  \langle x := 3; x := 4, \sigma \rangle \\
  \rightarrow
  \]

Let's Define Together