In our last exciting episode ...

Lessons From Model Checking

• To find **bugs**, we need **specifications**
  - What are some good specifications?
• To **convert** a program into a **model**, we need **predicates/invariants** and a **theorem prover**.
  - What are important predicates? Invariants?
  - What should we track when reasoning about a program and what should we abstract?
  - How does a theorem prover work?
• **Simple** algorithms (e.g., depth first search, pushing facts along a CFG) can work well
  - ... under what circumstances?

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**"
Language design step 1: Syntax

• What is the syntax of a language? What are some issues that come up?
  • Symbols, seq of words
  • Identifiers, separators

Concrete syntax

• The rules by which programs can be expressed as strings of characters
  • Keywords, identifiers, statement separators vs. terminators, comments, indentation

• Concrete syntax is important in practice
  • For readability, familiarity, parsing speed, effectiveness of error recovery, clarity of error messages

• Well understood principles
  • Use finite automata and context-free grammars
  • Automatic lexer/parser generators

 Syntax

• Concrete syntax
  • Concrete syntax is important in practice

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

Abstract Syntax (Aexp)

• Arithmetic expressions (Aexp)
  e ::= n for n ∈ Z
  | x for x ∈ L
  | e₁ + e₂ for e₁, e₂ ∈ Aexp
  | e₁ - e₂ for e₁, e₂ ∈ Aexp
  | e₁ * e₂ for e₁, e₂ ∈ Aexp

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

Abstract Syntax (Bexp)

• Boolean expressions (Bexp)
  b ::= true
  | false
  | e₁ = e₂ for e₁, e₂ ∈ Aexp
  | e₁ ≤ e₂ for e₁, e₂ ∈ Aexp
  | ¬ b for b ∈ Bexp
  | b₁ ∧ b₂ for b₁, b₂ ∈ Bexp
  | b₁ ∨ b₂ for b₁, b₂ ∈ Bexp

For IMP, these also encode the types.
“Boolean”
- George Boole
- 1815–1864
- Recall boolean algebra:

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P ∧ Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Abstract Syntax (Com)
- Commands (Com)
  ∙ c ::= skip
  ∙ | x := e for x ∈ L and e ∈ Aexp
  ∙ | c₁; c₂ for c₁, c₂ ∈ Com
  ∙ | if b then c₁ else c₂ for c₁, c₂ ∈ Com, b ∈ Bexp
  ∙ while b do c for c ∈ Com, b ∈ Bexp

- 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

Abstract Syntax (Com)
- What’s missing?
  ∙ Functions!
  ∙ Strings
  ∙ Pointers/Heap
  ∙ Exceptions
  ∙ Objects
  ∙ threads

Why study formal semantics?
- Language design (denotational)
- Proofs of correctness (axiomatic)
- Language implementation (operational)
- Reasoning about programs
- Providing a clear behavioral specification
- “All the cool people are doing it.”
  - You need this to understand PL research

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

14.20.2 Execution of try-catch-finally
- If the statement with a finally block is executed by first executing the try block. Then there is a choice:
  1. If execution of the try block completes normally, then the finally block is executed, and then there is a choice:
    1. If the finally block completes normally, then the try statement completes normally.
    2. If the finally block completes abruptly because of a throw of a value V, then there is a choice:
      1. If the run-time type of V is assignable to the parameter of any catch clause of the try statement, then the try statement completes abruptly.
      2. If the run-time type of V is not assignable to the parameter of any catch clause of the try statement, then there is a choice:
        1. If the finally block completes normally, then the finally block completes abruptly because of a throw of the value V.
        2. If the finally block completes abruptly for any other reason R, then the finally block completes abruptly for reason R.

- If execution of the try block completes abruptly by any other reason R, then the finally block is executed. Then there is a choice:
  1. If the finally block completes normally, then the try statement completes abruptly.
  2. If the finally block completes abruptly for reason R, then the try statement completes abruptly for reason R.
Ouch! Confusing

- Wouldn’t it be nice if we had some way of describing what a language (feature or program) means …
  - More precisely than English
  - More compactly than English
  - So that you might build a compiler
  - So that you might prove things about programs

Analysis of IMP

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

Three Canonical Approaches

- Operational
  - How would I execute this?
- Axiomatic
  - What is true after I execute this?
- Denotational
  - What is this trying to compute?

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 $n_1 + n_2$.
  - Similar for $e_1 * e_2$

Operational semantics abstracts the execution of a concrete interpreter

Semantics of IMP

- The meanings of IMP 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 $L$ to $\mathbb{Z}$ (a state)
  - If $x = 8$ in our state, we expect “$x+5$” to mean 13
- The set of all states is $\Sigma = L \rightarrow \mathbb{Z}$
- 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, $\sigma : L \rightarrow \mathbb{Z}$

“What’s L again?”
Notation: Evaluation Judgment

• We write:
  \[ \langle e, \sigma \rangle \Downarrow n \]

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

Flavors of Operational Semantics

• This formulation is called natural operational semantics
  - or big-step operational semantics
  - the \( \Downarrow \) judgment relates the expression and its "meaning"

• How should we define
  \[ \langle e_1 + e_2, \sigma \rangle \Downarrow \ldots ? \]

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:
  \[
  \frac{
  \langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2
  }{
  \langle e_1 + e_2, \sigma \rangle \Downarrow \, n_1 + n_2
  }
  \]

Rules of Inference

<table>
<thead>
<tr>
<th>Hypothesis_1, ... Hypothesis_n</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>A is true</td>
<td>B is true</td>
</tr>
<tr>
<td>A \land B is true</td>
<td></td>
</tr>
</tbody>
</table>

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

Derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Hypothesis</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \Gamma(x) = \text{int} ] \quad \Gamma \vdash x : \text{int}</td>
<td>\Gamma \vdash \text{var}</td>
<td>\Gamma \vdash \text{var}</td>
</tr>
<tr>
<td>[ \Gamma \vdash x &gt; 3 : \text{bool} ] \quad \Gamma \vdash x : \text{int}</td>
<td>\Gamma \vdash \text{var}</td>
<td>\Gamma \vdash \text{var}</td>
</tr>
<tr>
<td>[ \Gamma \vdash \text{while } x &gt; 3 , \text{do } x := x - 1 ] \quad \Gamma \vdash x := x - 1 : \text{int}</td>
<td>\Gamma \vdash \text{while } x &gt; 3 , \text{do } x := x - 1 ] \quad \Gamma \vdash \text{assign}</td>
<td></td>
</tr>
</tbody>
</table>

• Tree-structured (conclusion at bottom)
• May include multiple sorts of rules-of-inference
• Provides proof of a judgment
Evaluation Rules (for Aexp)

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

\[
\begin{align*}
<n, \sigma> & \Downarrow n \\
<e_1, \sigma> \Downarrow n_1 & \quad <e_2, \sigma> \Downarrow n_2 \\
<e_1 + e_2, \sigma> & \Downarrow n_1 + n_2 \\
<e_1 - e_2, \sigma> & \Downarrow n_1 - n_2 \\
<e_1 \ast e_2, \sigma> & \Downarrow n_1 \ast n_2
\end{align*}
\]

- What does this say about evaluating \( \mathcal{A} \)?

Non-Short-Circuiting Rule for \( \land \)

\[
\begin{align*}
<b_1, \sigma> & \Downarrow \text{true} \\
<b_2, \sigma> & \Downarrow \text{true} \\
<b_1 \land b_2, \sigma> & \Downarrow \text{true}
\end{align*}
\]

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
    \[
    \begin{align*}
    <e_1, \sigma> & \Downarrow 5 \\
    <e_2, \sigma> & \Downarrow 7
    \end{align*}
    \]
    then we can infer that
    \[
    <e_1 + e_2, \sigma> \Downarrow 12
    \]

- Backward (bottom-up) = evaluation rules
  - Suppose we want to evaluate \( e_1 + e_2 \), i.e., find \( n \) s.t. \( e_1 + e_2 \Downarrow n \) is derivable using the previous rules.
  - By inspection of the rules we notice that the last step in the derivation of \( e_1 + e_2 \Downarrow n \) must be the addition rule.
  - This is called reasoning by inversion on the derivation rules.
Syntax-Directed Evaluation
• Thus we must find $n_1$ and $n_2$ such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ 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)

Evaluation of Commands
• The evaluation of a Com may have side effects but has no direct result
  - What is the result of evaluating a command?
• The “result” of a Com is a new state: $<c, \sigma> \Downarrow \sigma'$
  But the evaluation of a Com might not terminate!?!?

Evaluation Rules (for Com)

Evaluation Rules (for Com, assignment)

Evaluation Rules (for Com, while)
• Let’s do while together:
Evaluation Rules (for Com, while)

- Let's do while together:

\[ \langle b, \sigma \rangle \uparrow \text{false} \]
\[ \langle \text{while } b \text{ do } c, \sigma \rangle \uparrow \sigma' \]
\[ \langle b, \sigma \rangle \uparrow \text{true} \]
\[ \langle c; \text{while } b \text{ do } c, \sigma \rangle \uparrow \sigma' \]

Observations: Command Evaluation

- Order of evaluation?

- The order of evaluation is important:
  - \( c_1 \) is evaluated before \( c_2 \) in \( c_1; c_2 \)
  - \( c_2 \) is not evaluated in "if true then \( c_1 \) else \( c_2 \)"
  - \( c \) is not evaluated in "while false do \( c \)"
  - \( b \) is evaluated first in "if \( b \) then \( c_1 \) else \( c_2 \)"
  - This is explicit in the evaluation rules

- Conditional constructs have multiple evaluation rules:
  - but only one can be applied at one time

Observations: Command Evaluation

- The evaluation rules are not syntax-directed
  - See the rules for while
  - The evaluation might not terminate

- Recall: the evaluation rules suggest an interpreter

For Next Time

- Homework 1 out today, due Feb 2
- Reading and Forum, as usual