Questions from last week?

Announcements

- Homework 0 is due Thu at 11:55pm
  - How’s it going?
  - Can’t get BLAST to work?
    - Try the CSEL/ECES Linux machines with the BLAST binaries
- Homework 1 out Thu
  - Will need OCaml. Resources on website.
    - OCaml installed on CSEL machines.
  - Fantastic forum participation: keep it up!

What is Counterexample Guided Abstraction Refinement (CEGAR)?

Verification by...

- Model Checking?
  - Designs, Specifications, Synthesis, Verification
- Theorem Proving?
  - Logical Formulas, Decidability, Completeness, Proofs
- Program Analysis?
  - Abstraction
Verification

Is this program correct?

What does correct mean?

How do we determine if a program is correct?

Verification by Model Checking

Example

1. (Finite State) Program
2. State Transition Graph
3. Reachability
- Loop invariants
- Multi-threaded programs
- Behaviors encoded in logic
- Decision procedures
Precise [SPIN,SMV,Bandera,]FIP

Verification by Theorem Proving

Example

1. Loop Invariants
2. Logical Formulas
3. Check Validity

Invariant:

\[ \text{lock} \land \text{new} = \text{old} \lor \neg \text{lock} \land \text{new} \neq \text{old} \]

Verification by Program Analysis

Example

1. Dataflow Facts
2. Constraint System
3. Solve Constraints

- Imprecision: fixed facts
- Abstraction
- Type/flow analyses

Scalable [Qud,ESP]
Lessons From Model Checking

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

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”

Semantics

Medium-Range Plan

- Study a simple imperative language IMP
  - Abstract syntax
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics
  - and relationships between various semantics
- Today: Operational semantics
  - Follow along in Winskel, Chapter 2

Language design step 1: Syntax

- What is the syntax of a language? What are some issues that come up?
  - Keywords, variables, operators
  - block structure, comment structure, whitespace
  - “Step 1 in the language = concrete syntax

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 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*

* Note: recent research: consider GLR (elkhound) instead of LALR(1) (bison) parsers.
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

**IMP Syntactic Entities**

| **int** | integer constants | \( n \in \mathbb{Z} \) |
| **bool** | boolean constants | true, false |
| **L** | locations (variables) | \( x, y, \ldots \) |
| **Aexp** | arithmetic expressions | \[ e \] |
| **Bexp** | boolean expressions | \( b \) |
| **Com** | commands | \( c \) |

For IMP, these also encode the types.

Abstract Syntax (Aexp)

- Arithmetic expressions (Aexp)
  
  \[
  e ::= \begin{cases} \text{n} & \text{for } n \in \mathbb{Z} \\ \text{x} & \text{for } x \in L \\ e_1 + e_2 & \text{for } e_1, e_2 \in \text{Aexp} \\ e_1 - e_2 & \text{for } e_1, e_2 \in \text{Aexp} \\ e_1 \times e_2 & \text{for } e_1, e_2 \in \text{Aexp} \end{cases}
  \]

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

Abstract Syntax (Bexp)

- Boolean expressions (Bexp)
  
  \[
  b ::= \begin{cases} \text{true} & \text{false} \\ e_1 = e_2 & \text{for } e_1, e_2 \in \text{Aexp} \\ e_1 \leq e_2 & \text{for } e_1, e_2 \in \text{Aexp} \\ \neg
  \neg \neg \neg b & \text{for } b \in \text{Bexp} \\ b_1 \land b_2 & \text{for } b_1, b_2 \in \text{Bexp} \\ b_1 \lor b_2 & \text{for } b_1, b_2 \in \text{Bexp} \end{cases}
  \]

Abstract Syntax (Com)

- Commands (Com)
  
  \[
  c ::= \text{skip} \\ x := e & \text{for } x \in L \text{ and } e \in \text{Aexp} \\ c_1 ; c_2 & \text{for } c_1, c_2 \in \text{Com} \\ \text{if } b \text{ then } c_1 \text{ else } c_2 & \text{for } c_1, c_2 \in \text{Com}, b \in \text{Bexp} \\ \text{while } b \text{ do } c & \text{for } c \in \text{Com}, b \in \text{Bexp} \end{cases}
  \]

- 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

---

"Boolean"

- George Boole
  - 1815-1864
- Recall boolean algebra:
  
<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P \land 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>

"Boole's friend"
Abstract Syntax (Com)

- What's missing?

```
c ::= skip
    | x := e
    | c1; c2
    | if b then c1 else c2
    | while b do c
```

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?

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

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 the finally block completes normally, then the try statement completes normally.
  - If the try block completes abruptly for reason \( R \), then the try statement completes abruptly for reason \( R \).

- 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 for reason \( S \).
    - If the catch block completes abruptly for reason \( R \), then the finally block completes abruptly for reason \( R \).
      - If the try block completes abruptly for reason \( R \), then the try statement completes abruptly for reason \( R \).

- If execution of the try block completes abruptly for any other reason \( R \), then the finally block is executed. Then there is a choice:
  - If the catch block completes abruptly for reason \( R \), then the finally block completes abruptly for reason \( R \).
  - If the try 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 \( e_1 \cdot e_2 \)

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, \( \sigma : L \rightarrow \mathbb{Z} \)

Notation: Evaluation Judgment

- We write: \(<e, \sigma> \downarrow n\)
- 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
  \[<e_1 + e_2, \sigma> \Downarrow \ldots ?]\n
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**:

  \[
  \frac{<e_1, \sigma> \Downarrow n_1 \quad <e_2, \sigma> \Downarrow n_2}{<e_1 + e_2, \sigma> \Downarrow n_1 + n_2}
  \]

  This is the only rule for \( e_1 + e_2 \)

Rules of Inference

\[
\begin{array}{c}
A \quad \text{Hypothesis}_1 \ldots \text{Hypothesis}_n \\
\hline
A \quad \text{Conclusion}
\end{array}
\]

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

Evaluation Rules (for \( A\text{exp} \))

\[
\begin{align*}
<\text{e}_1, \sigma> & \Downarrow n_1 & & <\text{x}, \sigma> \Downarrow \text{sx} \\
<\text{e}_1 + \text{e}_2, \sigma> & \Downarrow n_1 + n_2 & & <\text{e}_1, \sigma> \Downarrow n_1 - n_2 \\
<\text{e}_1 \ast \text{e}_2, \sigma> & \Downarrow n_1 \ast n_2 & & <\text{e}_1, \sigma> \Downarrow n_1 & <\text{e}_2, \sigma> \Downarrow n_2
\end{align*}
\]

• 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

\[
\begin{align*}
<\text{e}_1, \sigma> & \Downarrow n_1 & & <\text{x}, \sigma> \Downarrow \text{sx} \\
<\text{e}_1 + \text{e}_2, \sigma> & \Downarrow n_1 + n_2 & & <\text{e}_1, \sigma> \Downarrow n_1 - n_2 \\
<\text{e}_1 \ast \text{e}_2, \sigma> & \Downarrow n_1 \ast n_2 & & <\text{e}_1, \sigma> \Downarrow n_1 & <\text{e}_2, \sigma> \Downarrow n_2
\end{align*}
\]

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