CS655 Lecture 2 A Simple Imperative Language (Operational Semantics) 24 January, 2006

Scribe: Joseph Carnahan Lecturer: Wes Weimer

1 Administrivia

The results of the beginning-of-class survey have been posted on the course web site. Comments on the students' answers:

- Most students were familiar with induction, which is good – We will be using lots of structural induction in this class, also known as "induction on steroids." You should already be familiar with it, and you should look it up on Wikipedia (or a textbook) if you are not.
- Using LaTeX for typesetting will be mandatory, as it is the language for the vast majority of programming-languages-related publications. It works well with version control systems (such as CVS and Subversion) because it is stored in plain text. (By the way, LaTeX itself is a fully-functional dynamically-scoped programming language in its own right.)
- Students were interested in the following topics:
  - Security
  - Type systems
  - PL basics (Not really in this class, sorry - We will cover "basis", not "basics")
  - New languages (We will not cover a lot of these - Features, not languages)
  - Symbolic execution (Operational semantics by another name)
  - Abstract interpretation
  - Theorem proving
  - Finding a research topic
  - Understand the CQual paper
  - Help with quals
  - Advanced topics (That is *everything* in this class!)

The first homework has been posted online and is due in one week. Note the Wednesday office hours, which means the only office hours for HW1 are tomorrow afternoon.

Today's lecture covers material from Chapter 2 of Winskel. We will be using a simple example language, called IMP (IMperative Programming language).

2 Syntax of IMP

Before discussing its semantics, we must first establish the syntax of IMP. There are two types of syntax to consider: *Concrete syntax* is the set of rules for expressing programs as strings of characters. Concrete syntax defines identifiers, keywords, comments, separators, etc. (As an aside, when developing the concrete syntax for languages of your own, consider using Generalized LR (GLR) parsers like `elkhound` instead of LALR parsers like `bison`. GLR grammars are usually easier to understand and GLR parsers are now just as fast as comprable LALR parsers.) *Abstract syntax* defines a simplified form that discards tokens that do not have meaning (e.g. separators) – We will concern ourselves with the abstract syntax.

IMP has the following syntactic entities:

- int (integers)
- bool (booleans)
- L (stands for "locations", but they are commonly called variables)

- Aexp (aritmetic expressions)
- Bexp (boolean expressions)
- commands (if, while, assignment, and composition)

The textbook and slides use Backus-Naur Form (BNF) to describe the abstract syntax of the language.

Note two major differences from C-like languages: Expressions do not have side effects, and commands (such as assignment) cannot be evaluated as expressions.

Syntax for commands:

$$\begin{array}{rcl} \text{command} & ::= & \mathsf{skip} \parallel \\ & & x{:=}e \parallel \\ & & c1 \ ; \ c2 \parallel \\ & & \mathsf{if}\ b\ \mathsf{then}\ c1\ \mathsf{else}\ c2 \parallel \\ & & \mathsf{while}\ b\ \mathsf{do}\ c \end{array}$$

where

- x is a variable,
- e is an expression,
- c, $c1$, and $c2$ are commands, and
- b is a boolean expression.

We are assuming that all variables are declared, though that can not be expressed in our context-free syntax descrption.

3 Background on Semantics

Why do we want study formal semantics?

- We want to define a language for use by others (denotational).
- We want to perform program verification and/or write correctness proofs (axiomatic).
- We want to implement a language (operational).
- We want to reason about programs.
- We want to be able to write clear program specifications.

There are two papers assigned for reading today that use operational semantics. Throughout proceedings of conferences like PLDI, you will find operational semantics being used.

We considered a Java example that tests our understanding of try, catch, and finally. Consulting the natural-language specification of try-catch-finally blocks in Java is not helpful, as the specification is ugly, long, requires lots of nested "if" statements, and is possibly still ambiguous.

There are three types of formal semantics: *Operational semantics* defines a program by its execution on an interpreter. *Axiomatic semantics* defines a program by making assertions about how the state changes with each step. *Denotational semantics* tries to capture what computation (in a mathematical sense) is going on at each step.

4 Operational Semantics

Rather than providing the source for a real compliler (e.g. `gcc`), operational semantics defines an abstract interpreter. So, no information is given about details like whether the symbol table is an array or a hashtable, but instead a function is provided that maps variables to values at any time. In our example the mapping function is called σ, and we call this function the *state* of the program. The set of all possible states is called Σ.

Structural operational semantics uses syntax to guide rules for evaluation – Think of an interpreter implemented by making a postorder traversal of the abstract syntax tree.

Important notation to learn:

$$\langle e, \sigma \rangle \Downarrow n$$

This means that the expression e in state σ evaluates to the value n. The $\Downarrow$ is called a *judgement*, and it can be considered a function of two arguments, namely the current command or expression and the program's current state.

Now we want to define $e1 + e2$. To do this, we have to evaluate $e1$ and $e2$ first. The rule defining $e1 + e2$ is written as

$$\frac{\langle e1, \sigma \rangle \Downarrow n1 \quad \langle e2, \sigma \rangle \Downarrow n2}{\langle e1 + e2, \sigma \rangle \Downarrow n1 \text{ plus } n2}$$

This may appear circular, but it is not – "plus" means "addition as implemented by your computer," and "+" is "the addition symbol in the language IMP."

Typing rules can also be structured this way:

$$\frac{\Gamma \vdash b : \mathit{bool} \quad \Gamma \vdash e1 : \tau \quad \Gamma \vdash e2 : \tau}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e1\ \mathsf{else}\ e2 : \tau}$$

This means that if b has *bool* type and $e1$ and $e2$ have type τ, then the type of the whole expression is τ.

Derivations with these rules take on a tree format. To see the rules for arithmetic expressions, take a look at page 14 in Winskell, and the rules for boolean expressions can be found on page 17. Note that these rules do not specify evaluation order – You could evaluate either the left-hand operand or right-hand operand first.

Likewise, note that rules are not ordered. An example was given with the rules for or: You can define that false-or-false is false,

$$\frac{\langle b1, \sigma \rangle \Downarrow \mathsf{false} \quad \langle b2, \sigma \rangle \Downarrow \mathsf{false}}{\langle b1 \vee b2, \sigma \rangle \Downarrow \mathsf{false}}$$

but you can not get away with making only one more rule saying that all other $\vee$s are true:

$$\frac{}{\langle b1 \vee b2, \sigma \rangle \Downarrow \mathsf{true}}$$

Don't do this!

The problem is that someone could come along and read the second rule first, and since the second rule does not say that $b1$ and $b2$ are not false, this person could decide that false $\vee$ false is true.

To avoid this problem, you have to enumerate the two cases where one side is true and where the other side is true. Note that you do not need to give a rule for the case where both $b1$ and $b2$ are true, as that will match either of the rules where one side is true and will evaluate correctly to true in both cases.

5 Interpreting Operational Semantics

There are two ways to read the rules: Top-down (forwards) or bottom-up (backwards). *Forwards* means reading the hypotheses first and picking constructions to acheive a desired result. *Backwards* means taking the expression and working back through the rules to evaluate the expression. Backwards evaluation is also called *inversion*.

Backwards evaluation/inversion is done recursively. If the rules are syntax-directed, there is only one rule for each syntactic structure (at least at abstract syntax level). In IMP, the arithmetic expression rules

are syntax-directed, but the boolean rules are not because there are multiple rules for $b1 \wedge b2$ and $b1 \vee b2$. Syntax-directed semantics rules are much easier to figure out/read/implement, and so it is desirable to use syntax-directed semantics if possible.

6 Operational Semantics for Commands

When a command is evaluated, it does not produce a number but a new program state:

$$\langle c, \sigma \rangle \Downarrow \sigma'$$

So, skip does nothing, leaving the state unchanged:

$$\frac{}{\langle \mathsf{skip}, \sigma \rangle \Downarrow \sigma}$$

Sequence takes you to the result of executing the second command in the state that you get by executing the first command in the first state:

$$\frac{\langle c1, \sigma \rangle \Downarrow \sigma' \quad \langle c2, \sigma' \rangle \Downarrow \sigma''}{\langle c1; c2, \sigma \rangle \Downarrow \sigma''}$$

This forces you to execute the first command first, because there is no other way to figure out what the intermediate state σ' will be.

if statements mean that if the boolean is true, then you apply the state change for the first command, and if the boolean is false, you apply the state change from the second command.

$$\frac{\langle b, \sigma \rangle \Downarrow \mathsf{true} \quad \langle c1, \sigma \rangle \Downarrow \sigma'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c1\ \mathsf{else}\ c2, \sigma \rangle \Downarrow \sigma'}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \mathsf{false} \quad \langle c2, \sigma \rangle \Downarrow \sigma''}{\langle \mathsf{if}\ b\ \mathsf{then}\ c1\ \mathsf{else}\ c2, \sigma \rangle \Downarrow \sigma''}$$

Assignment requires us to invent new syntax: We say $\sigma[x := n]$, which is defined as follows:

$$\begin{aligned} \sigma[x := n](x) &= n \\ \sigma[x := n](y) &= \sigma(y) \end{aligned}$$

In other words, if you just assigned to x, then x's value is the new value and all the other variables (such as y) are not changed. Using this notation, assignment is defined as follows:

$$\frac{}{\langle x := n, \sigma \rangle \Downarrow \sigma[x := n]}$$

For while, there are two cases. The easy case is when b is false, because then the end state is the same as the start state.

$$\frac{\langle b, \sigma \rangle \Downarrow \mathsf{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle \Downarrow \sigma}$$

true is harder, and we actually need to do it recursively:

$$\frac{\langle b, \sigma \rangle \Downarrow \mathsf{true} \quad \langle c, \sigma \rangle \Downarrow \sigma' \quad \langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma' \rangle \Downarrow \sigma''}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle \Downarrow \sigma''}$$

The hypothesis is that b is true and that σ' is the result of executing c in the current state σ. Then, the result is the result of re-evaluating the while-loop in state σ'.

Operational semantics is sometimes also called large-step or natural-step semantics because of these sweeping rules (e.g., we just defined the entire while loop in one step with two rules).