CS655 Lecture 3 — Contextual Operational Semantics — 26 January, 2006

Scribe: Duane Merrill — Lecturer: Wes Weimer

1 Review of Operational Semantics

Today's lecture began with a review of Tuesday's introductory lecture on operational semantics. Operational semantics are a type of **formal semantics**; a way of giving mathematically rigorous meaning to programs by providing models for computation. In operational semantics, meaning of a program is derived from evaluating it as a sequence of computational steps, like an interpreter. Our learning example thus far has been the IMP imperative language.

Rules of inference are the tools in operational semantics that allow you to evaluate programs. The operation of a programming language can be described by a finite set of inference rules, each of which can be expressed a set of **premises** that lead to a **conclusion**:

$$\frac{\begin{array}{c}\textsf{Premise 1}\\ \textsf{Premise 2}\\ \vdots\\ \textsf{Premise}N\end{array}}{\textsf{Conclusion}}$$

For example, the inference rule

$$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)}$$

has no premises and concludes that, in program state σ, the location x evaluates to $\sigma(x)$. Consider another example:

$$\frac{\langle e1, \sigma\rangle \Downarrow n1 \quad \langle e2, \sigma\rangle \Downarrow n2}{\langle e1 - e2, \sigma\rangle \Downarrow n1 \textsf{ minus } n2}$$

There are two premises: "expression $e1$ evaluates to $n1$" and "expression $e2$ evaluates to $n2$". The conclusion is that subtracting these two expressions ($e1 - e2$) evaluates to $n1 - n2$.

In order to derive more interesting **judgements** (for example, what $(x * y) - z$ equates to when $x = 4$, $y = 2$, and $z = 6$), we create trees of interlocking inference rules by pattern-matching them into our desired conclusions in a bottom-up fashion:

$$\frac{\dfrac{\langle x, \sigma\rangle \Downarrow 4 \quad \langle y, \sigma\rangle \Downarrow 2}{\langle x * y, \sigma\rangle \Downarrow 8} \qquad \langle z, \sigma\rangle \Downarrow 6}{\langle (x * y) - z, \sigma\rangle \Downarrow 2}$$

A finite derivation tree, such as the one above, allows us to **prove** conclusions (in polynomial time) provided that it is **well-formed**; every step in the tree is a valid match for one of the inference rules for the given op-sem system. We use the **turnstile symbol** $\vdash$ to mean "infers", "proves" or "concludes", i.e.:

"$\vdash \langle e, \sigma\rangle \Downarrow n$" = "it is provable that expression e in state σ evaluates to n"

We note that meaning is only derived from judgments that can be successfully evaluated; "while (*true*) skip;" will never terminate and therefore has no meaning. Additionally, a failed attempt to locate a valid inference rule when evaluating a judgment (i.e., the op-sem system does not provide for a particular operation in the judgment or a named location/variable is unbound) means that the program has "**gone wrong**".

It is desirable that our op-sem systems allow us to prove all true things and no false things. However, at this point, we're going to punt on a formal definition of truth and appeal to general intuition (i.e., that "$\langle 2, \sigma\rangle \Downarrow 5$" is false, lest it lead to the notion that $2 + 2 = 10$).

We also would like our op-sem systems to be **complete**; that every true judgment is provable. For example, suppose we alter one of the premises in our subtraction inference rule:

$$\frac{\langle e1, \sigma\rangle \Downarrow \mathsf{n1} \quad \langle e2, \sigma\rangle \Downarrow \mathsf{0}}{\langle e1 - e2, \sigma\rangle \Downarrow n1 \text{ minus } n2}$$

This alters the rule so that it is only applicable when subtracting two numerical expressions in which the second evaluates to zero. With this new rule, we can prove the true judgment "$\langle 4 - 0, \sigma\rangle = 4$" but not the true judgment "$\langle 4 - 2, \sigma\rangle = 2$". Although "$4 - 2$" meets the pattern for the conclusion, it does not meet the required premises.

Finally, we note that an op-sem system is **consistent** (a.k.a. **sound**) iff every provable judgement is true. For example, replacing our original subtraction rule with:

$$\frac{\langle e1, \sigma\rangle \Downarrow \mathsf{n1} \quad \langle e2, \sigma\rangle \Downarrow \mathsf{n2}}{\langle e1 - e2, \sigma\rangle \Downarrow n1 \text{ plus } 3}$$

would cause the op-sem system to be **unsound**. (The judgment "$\langle 6 - 1, \sigma\rangle = 9$" is rejected by our intuitive sense of truth, but is provable with the altered subtraction inference rule).

For the purposes of this class, our systems should be complete and consistent (although some research, e.g., Engler, ESP, shows that there unsound systems that are still useful).

2 Introduction to Small-Step Semantics

Up until now, our op-sem system for IMP has been described using **large-step** semantics by providing direct relations between initial and final states. However, there are some drawbacks to such a big-picture view, specifically:

- Cannot express meaning for evaluations that do not terminate. As mentioned in the review portion of lecture, there is no meaning given for such evaluations as "while (*true*) skip;". Yet non-terminating evaluations may be doing interesting work (like sending your personal data to Microsoft).
- No way to express meaning for intermediate states of an evaluation. In general, the op-sem approach is to model execution as a sequence of computational steps, yet large-step semantics can gloss over things like:
  - How the op-sem "interpreter" might interleave the execution of two commands on a parallel machine, notions of race-conditions, etc.
  - The order of evaluation for premises that are not syntax-directed (i.e., the Boolean $\wedge$ operation).

An op-sem alternative to large-step semantics is **small-step** semantics, which models execution as a (possibly infinite) sequence of states.

3 Contextual Semantics

One particular type of small-step semantics is **contextual semantics**, which models execution as a sequence of **atomic rewrites** of state, between each of which some small amount of time passes. Derivations are expressed as sequences that progress with time, rather than as trees of inference that conclude instantaneously. For example, "$x + (y * 3)$" might be derived as:

$$\langle x + (y * 3), \sigma\rangle \to \langle x + (2 * 3), \sigma\rangle \to \langle x + 6, \sigma\rangle \to \langle 1 + 6, \sigma\rangle \to \langle 7, \sigma\rangle$$

As implied above, each atomic rewrite step is expressed as a relation between stages of a derivation:

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

where c' is derived from c during the atomic rewrite step $\to$.

We stop evaluating when we arrive at a "terminal program" (when there is no further progress to be made). For our IMP language, the terminal command is skip. Because some evaluations never reduce to simply skip (i.e., "while (*true*) skip;"), our derivations can be infinite (unlike large-step derivations).

A reducible expression (**redex**) is a syntactic expression or command that can be reduced (transformed) during an atomic rewrite. The transformation of the redex is governed by **reduction rules**. For example, a redex grammar containing "$n1 + n2$" could identify the redex "$5 + 4$" in a program. Then a reduction rule, such as "$\langle n1 + n2, \sigma\rangle \to \langle n, \sigma\rangle$" could be applied during an atomic rewrite to produce "9".

The redex grammar for IMP is:

$$\begin{array}{lll} \text{r} ::= & x & (x \in L) \\ & \mid n1 + n2 & \\ & \mid x := n & \\ & \mid \textsf{skip};\ c & \\ & \mid \textsf{if } \textit{true} \textsf{ then } c1 \textsf{ else } c2 & \\ & \mid \textsf{if } \textit{false} \textsf{ then } c1 \textsf{ else } c2 & \\ & \mid \textsf{while } b \textsf{ do} & \end{array}$$

Note that, according to the grammar, "$(1 + 3) + 2$" is not a redex, but "$1 + 3$" and "$4 + 2$" are redexes.

A reduction rule follows the general form

$$\langle r, \sigma\rangle \to \langle e, \sigma'\rangle$$

where, in state σ, the redex r can be replaced with expression e. (In fact, if e is not a value, e can be a redex called **redex-to-reduce-next**.) The reduction rules for IMP are:

$$\begin{array}{ll} \langle x, \sigma\rangle \to \langle \sigma(x), \sigma\rangle & \\ \langle n1 + n2, \sigma\rangle \to \langle n, \sigma\rangle & \text{where } n = n1 + n2 \\ \langle n1 = n2, \sigma\rangle \to \langle true, \sigma\rangle & \text{if } n1 = n2 \\ \langle x := n, \sigma\rangle \to \langle \text{skip}, \sigma[x := n]\rangle & \\ \langle \textsf{skip};\ c, \sigma\rangle \to \langle c, \sigma\rangle & \\ \langle \textsf{if } \textit{true} \textsf{ then } c1 \textsf{ else } c2, \sigma\rangle \to \langle c1, \sigma\rangle & \\ \langle \textsf{if } \textit{false} \textsf{ then } c1 \textsf{ else } c2, \sigma\rangle \to \langle c2, \sigma\rangle & \\ \langle \textsf{while } b \textsf{ do } c, \sigma\rangle \to \langle \textsf{if } b \textsf{ then } c;\ \textsf{while } b \textsf{ do } c \textsf{ else skip}, \sigma\rangle & \end{array}$$

When iterating through the transformation of an expression (or command), we use the term **context** to denote the portion of the expression other than the current redex. For example, suppose we have the expression "$(1 + 3) + 2$" and the current redex r is "$1 + 3$". The context $\mathsf{H}[r]$ can be written as "$\bullet + 2$", where the $\bullet$ symbol (known as the **hole**) is used as a placemarker where the redex r goes. Contexts are also defined by grammars. Specifically, a context has exactly one $\bullet$ marker. An example context grammar for IMP is:

$$\begin{array}{lll} \text{H} ::= \bullet & \mid n + \text{H} \\ & \mid \text{H} + e \\ & \mid x := \text{H} \\ & \mid \textsf{if H then } c1 \textsf{ else } c2 \\ & \mid \text{H};\ c \end{array}$$

The following is an example of a small-step reduction (progress flowing downward) showing the current redex and context for each reduction step:

	⟨Command, State⟩	Redex	Context
1	$\langle x := 1; x := x + 1, [x := 0]\rangle$	$x := 1$	$\bullet; x := x + 1$
2	$\langle skip; x := x + 1, [x := 1]\rangle$	$\textsf{skip}; x := x + 1$	$\bullet$
3	$\langle x := x + 1, [x := 1]\rangle$	x	$x := \bullet + 1$
4	$\langle x := 1 + 1, [x := 1]\rangle$	$1 + 1$	$x := \bullet$
5	$\langle x := 2, [x := 1]\rangle$	$x := 2$	$\bullet$
6	$\langle skip, [x := 2]\rangle$		

As illustrated above, decomposition is done recursively. In fact, we can often think of the • marker as the program counter: it is the activity that is currently being interpreted.

For another example, suppose a command $c =$ "if b then $c1$ else $c2$". If b is a literal ($true/false$), then

$$\begin{aligned} c &= \mathrm{H}[r] = \text{“}\bullet\text{”} \\ r &= \text{“if } b \text{ then } c1 \text{ else } c2\text{”} \end{aligned}$$

However, if b is another expression (suppose "$2 > 1$"), then

$$\begin{aligned} c &= \mathrm{H'}[r] = \text{“if } \mathrm{H}[r] \text{ then } c1 \text{ else } c2\text{”} \\ \mathrm{H}[r] &= \text{“}\bullet\text{”} \\ r &= \text{“}2 > 1\text{”} \end{aligned}$$

It should be noted that, as long as the current expression\command is not skip, there is always a unique next-redex r and context H. This means there will always be deterministic progress.

Finally, now that we have the power of small-step semantics, we can express things like the short-circuit evaluation of the Boolean "$b1 \wedge b2$" expression. We would do this by adding

$$\begin{array}{ll} \mathrm{H} ::= \ldots \mid \mathrm{H} \wedge b2 & \text{to the context grammar} \\ r ::= \ldots \mid true \wedge b \mid false \wedge b & \text{to the redex grammar} \\ \langle true \wedge b, \sigma\rangle \to \langle b, \sigma\rangle & \\ \langle false \wedge b, \sigma\rangle \to \langle false, \sigma\rangle & \text{to the local reduction rules} \end{array}$$

When $b1$ is false, the reduction rule kicks in, removing $b2$ entirely from the decomposition before it needs to be evaluated.

4 Contextual Semantics Summary

Contextual semantics are good for modeling incremental activities, such as thread execution or intermediate calculations. In general, decomposition in contextual semantics is done by repeating the three following steps: (1) find (2) reduce (3) replace. However, using this approach to implement a language interpreter would be very inefficient because the entire command must be decomposed for each iteration of the three steps.