CS 411 Lecture 5 Equivalence of Operational Semantics; Denotational Semantics 10 Sep 2003

Lecturer: Greg Morrisett

In the homework, I asked you to prove:

$$\vdash \mathsf{eval}(e, s) = i \iff \vdash (e, s) \Downarrow i.$$

which turns out to be much harder than it first looks. One direction is easy, but the other requires some cleverness.

Here's a sketch (you can fill in the details):

Definition 1. *$(e, s) \rightarrow^n (e', s)$ means that there exists expressions $e_1, e_2, \ldots, e_n$ such that*

$$(e, s) \rightarrow (e_1, s) \rightarrow (e_2, s) \rightarrow \ldots \rightarrow (e_n, s) = (e', s)$$

Note that if $n = 0$ then $e = e'$.

Definition 2. *$(e, s) \rightarrow^* (e', s)$ means that there exists an $n \geq 0$ such that $(e, s) \rightarrow^n (e', s)$.*

Lemma 1. *$(e, s) \rightarrow^* (i, s)$ iff $\mathsf{eval}(e, s) = i$.*

Proof. In the forward direction, you argue by induction on n, where n is the number of steps that allows us to conclude that $(e, s) \rightarrow^* (i, s)$.

In the backward direction, you argue by induction on the height of the tree that allows you to conclude $\mathsf{eval}(e, s) = i$. □

Lemma 2. *If $(e_1, s) \rightarrow^n (e_1', s)$ then $(e_1 + e_2, s) \rightarrow^n (e_1' + e_2, s)$ and $(e_1 * e_2, s) \rightarrow^n (e_1' * e_2, s)$.*

Proof. By induction on n. □

Lemma 3. *If $(e_2, s) \rightarrow^n (e_2', s)$ then $(i_1 + e_2, s) \rightarrow^n (i_1 + e_2', s)$ and $(i_1 * e_2, s) \rightarrow^n (i_1 * e_2', s)$.*

Proof. By induction on n. □

Lemma 4. *If $(e, s) \Downarrow i$ then $(e, s) \rightarrow^* (i, s)$.*

Proof. By induction on the height of the derivation that allows us to conclude $(e, s) \Downarrow i$. The base cases are easy. There are two inductive cases to consider. We'll deal with the one where the derivation ends with:

$$\frac{(e_1, s) \Downarrow i_1 \qquad (e_2, s) \Downarrow i_2}{(e_1 + e_2, s) \Downarrow i} (i = i_1 + i_2)$$

That is, $e = e_1 + e_2$. By the induction hypothesis, we have $(e_1, s) \rightarrow^* (i_1, s)$ and $(e_2, s) \rightarrow^* (i_2, s)$. By Lemma 2 we have $(e_1 + e_2, s) \rightarrow^* (i_1 + e_2, s)$. By Lemma 3 we have $(i_1 + e_2, s) \rightarrow^* (i_1 + i_2, s)$. And then from the transition rules, we have $(i_1 + i_2, s) \rightarrow (i, s)$. So, $(e_1 + e_2) \rightarrow^* (i_1 + e_2, s) \rightarrow^* (i_1 + i_2, s) \rightarrow (i, s)$. Thus, $(e_1 + e_2, s) \rightarrow^* (i, s)$. □

Corollary 4.1. *If $(e, s) \Downarrow i$ then $\mathsf{eval}(e, s) = i$.*

Lemma 5. *If* $(e, s) \rightarrow^n (e', s)$ *and* $(e', s) \Downarrow i$, *then* $(e, s) \Downarrow i$.

Proof. By induction on n. (Work it out for yourselves). $\square$

Corollary 5.1. *If* $\mathsf{eval}(e, s) = i$ *then* $(e, s) \Downarrow i$.

The theorem is thus established by Corollary 4.1 and Corollary 5.1.

Thus far, we've looked at operational models for our little programming languages. I want to briefly introduce you to two other modelling approaches: denotational semantics and axiomatic semantics. Today, we'll talk about denotational semantics for IMP.

The basic idea is that we're going to translate each IMP command c into a mathematical (partial) function from stores to stores. A partial function $F : A \rightharpoonup B$ is a set of pairs (x, y) such that $x \in A$ and $y \in B$, and for any $x \in A$, there is at most one y such that $(x, y) \in F$.

We'll begin by defining our denotational semantics by giving a partial function for the meaning of integer expressions as a partial function from stores to integers:

$$
\begin{array}{l}
\mathcal{E}[\![\cdot]\!] : \mathsf{Store} \rightarrow \mathsf{Int} \\
\begin{array}{rcl}
\mathcal{E}[\![i]\!] & = & \{(s, i)\} \\
\mathcal{E}[\![x]\!] & = & \{(s, i) \mid s(x) = i\} \\
\mathcal{E}[\![e_1 + e_2]\!] & = & \{(s, i) \mid (s, i_1) \in \mathcal{E}[\![e_1]\!] \wedge (s, i_2) \in \mathcal{E}[\![e_2]\!] \wedge i = i_1 + i_2\} \\
\mathcal{E}[\![e_1 * e_2]\!] & = & \{(s, i) \mid (s, i_1) \in \mathcal{E}[\![e_1]\!] \wedge (s, i_2) \in \mathcal{E}[\![e_2]\!] \wedge i = i_1 * i_2\}
\end{array}
\end{array}
$$

We'll next define a partial function for boolean expressions from stores to booleans:

$$
\begin{array}{l}
\mathcal{B}[\![\cdot]\!] : \mathsf{Store} \rightarrow \{\mathsf{true}, \mathsf{false}\} \\
\begin{array}{rcl}
\mathcal{B}[\![e_1 \leq e_2]\!] & = & \{(s, \mathsf{true}) \mid (s, i_1) \in \mathcal{E}[\![e_1]\!] \wedge (s, i_2) \in \mathcal{E}[\![e_2]\!] \wedge i_1 \leq i_2\} \\
& & \quad \cup \{(s, \mathsf{false}) \mid (s, i_1) \in \mathcal{E}[\![e1]\!] \wedge (s, i_2) \in \mathcal{E}[\![e_2]\!] \wedge i_1 > i_2\} \\
\mathcal{B}[\![\mathsf{true}]\!] & = & \{(s, \mathsf{true})\} \\
\mathcal{B}[\![\mathsf{false}]\!] & = & \{(s, \mathsf{false})\}
\end{array}
\end{array}
$$

Finally, we'll define a partial function from stores to stores to model our commands:

$$
\begin{array}{l}
\mathcal{C}[\![\cdot]\!] : \mathsf{Store} \rightharpoonup \mathsf{Store} \\
\begin{array}{rcl}
\mathcal{C}[\![\mathsf{skip}]\!] & = & \{(s, s)\} \\
\mathcal{C}[\![x := e]\!] & = & \{(s, s[x \mapsto i]) \mid (s, i) \in \mathcal{E}[\![e]\!]\} \\
\mathcal{C}[\![c_1; c_2]\!] & = & \{(s_1, s_2) \mid \exists s.\ (s_1, s) \in \mathcal{E}[\![c_1]\!] \wedge (s, s_2) \in \mathcal{E}[\![c_2]\!]\} \\
\mathcal{C}[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] & = & \{(s_1, s_2) \mid (s_1, \mathsf{true}) \in \mathcal{B}[\![e]\!] \wedge (s_1, s_2) \in \mathcal{C}[\![c_1]\!]\} \\
& & \quad \cup \{(s_1, s_2) \mid (s_1, \mathsf{false}) \in \mathcal{B}[\![e]\!] \wedge (s_1, s_2) \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] & = & \{(s_1, s_2) \mid (s_1, s_2) \in \mathcal{C}[\![\mathsf{if}\ e\ \mathsf{then}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}]\!]\}
\end{array}
\end{array}
$$

This last equation is problematic because if you unwind the definition you see we have:

$$\begin{aligned}\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] \quad = \quad & \{(s_1, s_1) \mid (s_1, \mathsf{false}) \in \mathcal{B}[\![e]\!]\} \\ & \cup \{(s_1, s_2) \mid (s_1, \mathsf{true}) \in \mathcal{B}[\![e]\!] \wedge \exists s.\ (s_1, s) \in \mathcal{C}[\![c]\!] \wedge (s, s_2) \in \mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]\}\end{aligned}$$

So, we're trying to define the meaning of "$\mathsf{while}\ e\ \mathsf{do}\ c$" in terms of the meaning of "$\mathsf{while}\ e\ \mathsf{do}\ c$". But this is not a definition!

To see the potential problem, consider the equation over integers:

$$x = x + 1$$

Note that there is no solution to this equation. So, it's not always the case that a mathematical equation has a solution. Furthermore, even when there is a solution, there's no guarantee that it will be unique. For instance:

$$4 = x^2$$

allows x to be both 2 and -2. If we truly want a mathematical definition that is well-founded, we need to come up with a way of constructing a unique definition for the meaning of while loops (without appealing to while loops.)

So, we somehow need to solve for "$\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]$" and ensure that the solution is unique. Furthermore, the solution should satisfy the equation above (if our denotational model is going to capture our intended semantics.)

One way we can do this is to build up the meaning of a while loop by conceptually unrolling it a bunch of times and then take the limit of the process. In particular, let us define

$$F_{e,c} : (\mathsf{Store} \rightharpoonup \mathsf{Store}) \to (\mathsf{Store} \rightharpoonup \mathsf{Store})$$

as follows:

$$\begin{aligned}F_{e,c}(G) \quad = \quad & \{(s, s) \mid (s, \mathsf{false}) \in \mathcal{B}[\![e]\!]\} \\ & \cup \{(s_1, s_2) \mid (s_1, \mathsf{true}) \in \mathcal{B}[\![e]\!] \wedge \exists s.\ (s_1, s) \in \mathcal{C}[\![c]\!] \wedge (s, s_2) \in G\}\end{aligned}$$

So, $F_{e,c}$ takes in a partial function (G) and acts like one iteration of the while-loop except that, instead of invoking itself when it needs to go around the loop again, it instead calls G.

We can now define:

$$\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] = \bigcup_{i \geq 0} {F_{e,c}}^{i}(\emptyset)$$

That is, we can start with the empty set (which is a partial function that just happens to be defined nowhere) and apply $F_{e,c}$ to it, then apply $F_{e,c}$ to the result of that, and then apply $F_{e,c}$ to the result of that and so on. We then take the union of all of these sets to get the total meaning of the while loop. In particular:

$$\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] = \emptyset \cup F_{e,c}(\emptyset) \cup F_{e,c}(F_{e,c}(\emptyset)) \cup F_{e,c}(F_{e,c}(F_{e,c}(\emptyset))) + \ldots$$

For example, let's consider what "$\mathsf{while}\ x \le y\ \mathsf{do}\ x := x+1$" means by writing down all of the intermediate steps:

$$\begin{array}{rcl}
F_0 & = & \emptyset \\
F_1 & = & \{(s,s) \mid (s, \mathsf{false}) \in \mathcal{B}[\![x \le y]\!]\} \\
& & \quad \cup \{(s_1,s_2) \mid (s_1, \mathsf{true}) \in \mathcal{B}[\![x \le y]\!] \wedge (s_1, s) \in \mathcal{C}[\![x := x+1]\!] \wedge (s, s_2) \in F_0\} \\
& = & \{(s,s) | s(x) > s(y)\} \\
F_2 & = & \{(s,s) \mid (s, \mathsf{false}) \in \mathcal{B}[\![x \le y]\!]\} \\
& & \quad \cup \{(s_1,s_2) \mid (s_1, \mathsf{true}) \in \mathcal{B}[\![x \le y]\!] \wedge (s_1, s) \in \mathcal{C}[\![x := x+1]\!] \wedge (s, s_2) \in F_1\} \\
& = & \{(s,s) \mid s(x) > s(y)\} \\
& & \quad \cup \{(s_1, s_1[x \mapsto s_1(x)+1]) \mid s_1(x) \le s_1(y) \wedge s_1(x)+1 > s_1(y)\} \\
F_3 & = & \{(s,s) \mid s(x) > s(y)\} \\
& & \quad \cup \{(s_1, s_1[x \mapsto s_1(x)+1]) \mid s_1(x) \le s_1(y) \wedge s_1(x)+1 > s_1(y)\} \\
& & \quad \cup \{(s_1, s_1[x \mapsto s_1(x)+2]) \mid s_1(x) \le s_1(y) \wedge s_1(x)+2 > s_1(y)\} \\
F_4 & = & \{(s,s) \mid s(x) > s(y)\} \\
& & \quad \cup \{(s_1, s_1[x \mapsto s_1(x)+1]) \mid s_1(x) \le s_1(y) \wedge s_1(x)+1 > s_1(y)\} \\
& & \quad \cup \{(s_1, s_1[x \mapsto s_1(x)+2]) \mid s_1(x) \le s_1(y) \wedge s_1(x)+2 > s_1(y)\} \\
& & \quad \cup \{(s_1, s_1[x \mapsto s_1(x)+3]) \mid s_1(x) \le s_1(y) \wedge s_1(x)+3 > s_1(y)\}
\end{array}$$

If we take the union of all F_i then we get:

$$\begin{array}{rcl}
\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] & = & \{(s,s) \mid s(x) > s(y)\} \\
& & \quad \cup \{(s_1, s_1[x \mapsto s_1(x)+n]) \mid s_1(x) \le s_1(y) \wedge s_1(x)+n > s_1(y))
\end{array}$$

which makes sense.

Now an interesting fact is that:

$$F_{e,c}(\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]) \subseteq \mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]$$

That is, unrolling the loop any more doesn't do us any good. And, it turns out that we can prove (using induction on i) that our desired equation holds:

$$\begin{array}{rcl}
\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] & = & \{(s_1,s_1) \mid (s_1, \mathsf{false}) \in \mathcal{B}[\![e]\!]\} \\
& & \quad \cup \{(s_1,s_2) \mid (s_1, \mathsf{true}) \in \mathcal{B}[\![e]\!] \wedge \exists s.\ (s_1,s) \in \mathcal{C}[\![c]\!] \wedge (s, s_2) \in \mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]\}
\end{array}$$

I'll leave that as an exercise `:-)`

Homework

Hand in electronically using the course management system `http://cms.csuglab.cornell.edu/`. This is Problem Set 2 (PS2). Hand in one long .txt file with all of your solutions. Wrap SML-style comments `(* ... *)` around non-code solutions so that we can just load the file into SML to test it out. Make sure you test your code carefully. If it doesn't at least type-check, you'll get no credit at all.

Last time, we talked about a simple imperative language (IMP).

$$\begin{aligned} op \in \mathsf{Op} &::= + \mid - \mid * \mid \leq \\ e \in \mathsf{Exp} &::= x \mid i \mid \mathsf{true} \mid \mathsf{false} \mid e_1\ op\ e_2 \\ c \in \mathsf{Com} &::= \mathsf{skip} \mid c_1; c_2 \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \end{aligned}$$

and gave a small-step semantics for the language.

1. Write down a large-step semantics for IMP. You should define two relations using inference rules:
   (a) $(e, s) \Downarrow (i, s)$
   (b) $(c, s) \Downarrow s$
2. Write SML code to implement your large-step semantics. In particular, we take the following datatypes as definitions for the abstract syntax:

```
type var = string
datatype op = Plus | Minus | Times | Lte
datatype exp = Var of var | Int of int | True | False | Op of exp * op * exp

datatype com = Skip | Seq of com * com | Assign of var * exp |
               If of exp * com * com | While of exp * com

type store = var -> int
```

   You need to define the following functions:

```
update_store : store * var * int -> store
eval_int : exp * store -> int * store
eval_bool : exp * store -> bool * store
eval_com : com * store -> store
```

3. Extend your language to support the following new constructs (you don't need to hand in separate code for parts 2 and 3):
   (a) operations ==, !=, <, >, >= for comparing integers
   (b) "$\mathsf{do}\ c\ \mathsf{while}\ e$" — should run the command c until e evaluates to false
   (c) short-circuited boolean operations && and ||
4. Using your big-step semantics, prove that the following commands are equivalent (i.e., $\vdash (c, s_1) \Downarrow s_2 \iff \vdash (c', s_1) \Downarrow s_2$).
   (a) $c_1; (c_2; c_3) = (c_1; c_2); c_3$
   (b) $(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1) = c_1$
   (c) $(y := i; x := 0) = (y := 0; x := i; \mathsf{while}\ (1 \leq x)\ \mathsf{do}\ (y := y + 1; x := x - 1))$