Concepts of Programmig Language Design

Semantics Exercises

Liam O'Connor-Davis, Gabriele Keller

December 5, 2024

1. **Logical Formulae:** Imagine we have a simple propositional expression language[1]:

$$\frac{}{\top\ \textbf{Prop}} \qquad \frac{}{\bot\ \textbf{Prop}} \qquad \frac{e_1\ \textbf{Prop} \quad e_2\ \textbf{Prop}}{e_1 \wedge e_2\ \textbf{Prop}} \qquad \frac{e_1\ \textbf{Prop}}{\neg e_1\ \textbf{Prop}}$$

   (a) The big-step semantics is given as

   - The set of evaluable expressions: $E = \{e \mid e\ \textbf{Prop}\}$
   - The set of values: $V = \{\texttt{True}, \texttt{False}\}$

   and the $\Downarrow$-relation, defined by the following rules:

$$\frac{}{\top \Downarrow \texttt{True}}\textsc{True} \qquad \frac{}{\bot \Downarrow \texttt{False}}\textsc{False}$$

$$\frac{e_1 \Downarrow \texttt{True}}{\neg e_1 \Downarrow \texttt{False}}\textsc{Not}_1 \qquad \frac{e_1 \Downarrow \texttt{False}}{\neg e_1 \Downarrow \texttt{True}}\textsc{Not}_2 \qquad \frac{e_1 \Downarrow \texttt{False}}{e_1 \wedge e_2 \Downarrow \texttt{False}}\textsc{And}_1 \qquad \frac{e_1 \Downarrow \texttt{True} \quad e_2 \Downarrow v}{e_1 \wedge e_2 \Downarrow v}\textsc{And}_2$$

   Determine a small step (SOS) semantics for the language **Prop**.

   i. Identify the set of states Q, the set of initial states I, and the set of final states F.

   ii. Provide inference rules for a relation $\mapsto : Q \times Q$, which performs one step only of the expression evaluation.

   (b) **Long, but not difficult:** We shall now prove that the reflexive, transitive closure of $\mapsto$, $\overset{\star}{\mapsto}$, is implied by the big-step semantics above. $\overset{\star}{\mapsto}$ is defined by the following rules:

$$\frac{}{e_1 \overset{\star}{\mapsto} e_1}\textsc{Refl}^* \qquad \frac{e_1 \mapsto e_2 \quad e_2 \overset{\star}{\mapsto} z}{e_1 \overset{\star}{\mapsto} z}\textsc{Trans}^*$$

   i. First prove the following transitivity lemma:

$$\frac{p \overset{\star}{\mapsto} q \quad q \overset{\star}{\mapsto} r}{p \overset{\star}{\mapsto} r}\textsc{Transitive}$$

   ii. Now prove the following two lemmas about Not:

$$\frac{e_1 \overset{\star}{\mapsto} \top}{\neg e_1 \overset{\star}{\mapsto} \bot}\textsc{Lemma-Not}_1 \qquad \frac{e_1 \overset{\star}{\mapsto} \bot}{\neg e_1 \overset{\star}{\mapsto} \top}\textsc{Lemma-Not}_2$$

   iii. Now prove the following lemmas about And:

$$\frac{e_1 \overset{\star}{\mapsto} \bot}{e_1 \wedge e_2 \overset{\star}{\mapsto} \bot}\textsc{Lemma-And}_1 \qquad \frac{e_1 \overset{\star}{\mapsto} \top}{e_1 \wedge e_2 \overset{\star}{\mapsto} e_2}\textsc{Lemma-And}_2$$

[1] Yes, the grammar is ambiguous, but assume it's just a symbolic representation of abstract syntax.

iv. Using these lemmas or otherwise, show that $E \Downarrow V$ implies $Q_E \overset{\star}{\mapsto} Q_V$, where Q_E is the state corresponding to the expression E and Q_V is the final state corresponding to the value V.

(c) Suppose we wanted to add quantifiers and variables to our logic language:

$$\frac{e\ \textbf{Prop}}{\exists(x.x)\ \textbf{Prop}} \qquad \frac{e\ \textbf{Prop}}{\forall(x.e)\ \textbf{Prop}} \qquad \frac{x\ \textbf{\textit{is a variable name}}}{x\ \textbf{Prop}}$$

It is no longer as easy to write static semantics for this language, as the formula may not be decidable, however we can still write static checkers that perform analysis on a more superficial level. Write a static semantics judgement for this language, written $\vdash e$ ***Ok*** (with whatever context you like before the $\vdash$), that ensures that there are no free variables in a given logical formula. Remember that a free variable is a variable that is not bound by a quantifier or lambda.

2. **Bizarro-Poland:** Imagine we have a reverse Polish notation calculator language. Reverse Polish notation is an old calculator format that does not require the use of parenthetical expressions. To achieve this, it moves all operators to post-fix, rather than in-fix order. E.g $1+2$ becomes $1\ 2\ +$, or $1-(3+2)$ becomes $1\ 3\ 2\ +\ -$. These calculators evaluated these expressions by pushing symbols onto a stack until an operator was encountered, when two symbols would be popped off and the result of the operation pushed on. The grammar is easily defined:

$$\frac{x \in \mathbb{N}}{x\ \textbf{\textit{Symbol}}} \qquad \frac{x \in \{+,-,/,*\}}{x\ \textbf{\textit{Symbol}}}$$

$$\frac{}{\epsilon\ \textbf{\textit{RPN}}} \qquad \frac{x\ \textbf{\textit{Symbol}} \quad xs\ \textbf{\textit{RPN}}}{x\ xs\ \textbf{\textit{RPN}}}$$

(a) The issue is that this grammar allows for invalid programs (such as $1+2$ or $-\ +\ *$).

i. Write some static semantics inference rules for a judgement $\vdash e$ ***Ok*** to ensure that programs are well formed.

ii. Show that $\vdash 1\ 3\ 2\ +\ -$ ***Ok***.

(b) We will now define some big-step evaluation semantics for this calculator. It may be helpful to read the program from right-to-left rather than left-to-right.

i. Identify the set of evaluable expressions E, and the set of result values V.

ii. Define a relation $\Downarrow : E \times V$ which evaluates RPN programs.

(c) Now we will try small-step semantics.

Our states Q will be of the form $s \vdash p$ where s is a stack of natural numbers and p is an RPN program.

Initial states are all states of the form $\circ \vdash p$.

Final states are all states of the form $n \triangleright \circ \vdash \epsilon$.

i. Define a relation $\mapsto : Q \times Q$, which evaluates one step of the calculation.

(d) Now we will prove small step and big step equivalent.

i. Show using rule induction on $\Downarrow$ that, for all expressions e, if $e \Downarrow v$ then $Q_e \overset{!}{\mapsto} Q_v$ (where Q_e and Q_v are initial and final states respectively corresponding to e and v). Hint: You may find it helpful to assume the following lemma (A proof of it is provided in the solutions):

$$\frac{\circ \vdash xs \overset{\star}{\mapsto} v \vdash \epsilon}{\circ \vdash xs\ x \overset{\star}{\mapsto} v \vdash x}\ \textsc{Append}$$

It is worth noting that the lemma Transitive from Question 1 applies here also.

ii. Show using rule induction on $\overset{\star}{\mapsto}$ that, for all expressions e and values v, if $Q_e \overset{\star}{\mapsto} Q_v$ then $e \Downarrow v$. It may be useful to assume the inverse of Append:

$$\frac{\circ \vdash xs\ x \overset{\star}{\mapsto} v \vdash x}{\circ \vdash xs \overset{\star}{\mapsto} v \vdash \epsilon}\ \textsc{Append}^{-1}$$

(e) Show that your static semantics defined in (a) ensure that the program will evaluate to a value. That is, show that $\vdash e\ \textbf{\textit{Ok}} \implies e \Downarrow s$, for some s. You may find it helpful to generalise your proof goal before beginning induction.

3. **Dynamic semantics of a simple robot control language** A robot moves along a grid according to a simple program. The program consists of a possibly empty sequence of the commands move and turn, separated by semicolons. Initially, the robot faces east and starts at the grid coordinates (0,0). The command turn causes the robot to turn 90 degrees counter-clockwise, and move to move one unit in the direction it is facing.

   (a) **Small step semantics:** devise a set of small-step semantics rules for this language. This means determining answers to the following questions:

   - What is the set of states?
   - Which of those states are final states, and which are initial states?
   - What transitions exist between those states?

   (b) **Big step semantics:** what would a suitable a set of big-step evaluation rules for this language look like? In particular:

   - What is the set of evaluable expressions?
   - What is the set of values?
   - How do evaluable expressions evaluate to those values?

4. **TinyC** The semantics we defined for TinyC in the lecture is a call-by-value semantics. That is, we only pass the value to the function.

   Extend TinyC such that the type of a function parameter in a function declaration can be preceeded by the keyword `var` so that this variable is passed by reference. For example, evaluating the following TinyC program would result in the value 20:

```
int swap (var int a; var int b) {
  int tmp = a;
  a = b;
  b = tmp;
  return 0;
}

{
  int x = 10;
  int y = 20;
  swap (x, y);
  return (x);
}
```

   - Which (if any) adjustments to the static semantics are necessary?
   - How can you model this in the operational dynamic semantics?