SmallC Formal Operational Semantics

March 3, 2017

1 Introduction

This document presents a formal semantics of Small C. It has two main parts. The first part is the *expression semantics*. It corresponds to the implementation of your `eval_expr` function. The second part is the *statement semantics*. It corresponds to the implementation of your `eval_stmt` function.

2 Preliminaries

2.1. Environments. Both parts define judgments that make use of environments A. The rules show two operations on environments:

- $A(x)$ means to look up the value of x maps to in the environment A. This operation is undefined if there is no mapping for x in A. We write $\cdot$ for the empty environment. Lookup on this environment is always undefined; i.e., $\cdot(x)$ is undefined for all x.
- The second operation is written $A[x \mapsto v]$. It defines a new environment that is the same as A but maps x to v. It thus overrides any prior mapping for x in A. This is similar to the "concatenation" of environments shown in the lecture notes from Feb. 28. So, $\cdot[x \mapsto 1]$ is an environment that maps x to 1 but is undefined for all other variables. The environment $\cdot[x \mapsto 1][y \mapsto 2]$ is an environment that maps x to 1 and y to 2. As such, if $A = \cdot[x \mapsto 1][y \mapsto 2]$ then $A(x) = 1$. That is, looking up x in the second example environment produces its mapped-to value, 1.

2.2. Syntax. In this document, we have simplified the presentation of the syntax, so it may not correspond exactly to the files that your interpreter will read in. For example, we write `while` e s to represent the syntax of a while-loop, where e is the guard and s is the body. This corresponds to `While of expr * stmt` in the `types.ml` file. Hopefully the connection between what we show here at that file is clear enough from context.

2.3. Error conditions. The semantics here defines only *correct* evaluations. It says nothing about what happens when, say, you have a type error. For example, for the rules below there is no value v for which you can prove the judgment $\cdot; 1 + \textbf{true} \longrightarrow v$. In your actual implementation, erroneous programs will cause an exception to be raised, as indicated in the project README.

3 Expression Semantics

This part of the formal semantics corresponds to the implementation of your `eval_expr` function. That function has type `environment -> expr -> value`. In the semantics, we write the judgment $A; e \longrightarrow v$, where A represents the environment, e represents the expression, and v represents the final value. This follows the same format as the lecture notes from February 28.

3.1. Abstract syntax grammar. Expressions and values are defined by the following grammar:

Integers	n	*is*	any integer
Booleans	b	::=	$\textbf{true} \mid \textbf{false}$
Values	v	::=	$n \mid b$
Expressions	e	::=	$v \mid\ !e \mid e \oplus e \mid e \odot e \mid e == e \mid e\ != e$

Here, we write $\oplus$ to represent any operator involving a pair of integers. This could be addition ($+$), comparison ($\leq$), multiplication ($\times$), etc. We write $\odot$ to represent any operator involving a pair of booleans. This could be boolean-and (&&), boolean-or (||), etc. We do not go into specifics here about what these actually do; they should match your intuition.

3.2. Rules. Here are the axioms.

$$\text{Id}\ \frac{A(x) = v}{A; x \longrightarrow v} \qquad \text{Int}\ \frac{}{A; n \longrightarrow n}$$

$$\text{Bool-True}\ \frac{}{A; \textbf{true} \longrightarrow \textbf{true}} \qquad \text{Bool-False}\ \frac{}{A; \textbf{false} \longrightarrow \textbf{false}}$$

Here are the rest of the rules for expressions.

$$\text{Eq-True}\ \frac{A; e_1 \longrightarrow v \quad A; e_2 \longrightarrow v}{A; e_1 == e_2 \longrightarrow \textbf{true}} \qquad \text{Eq-False}\ \frac{A; e_1 \longrightarrow v_1 \quad A; e_2 \longrightarrow v_2 \quad v_1 \text{ is different than } v_2}{A; e_1 == e_2 \longrightarrow \textbf{false}}$$

$$\text{NotEq-True}\ \frac{A; e_1 \longrightarrow v_1 \quad A; e_2 \longrightarrow v_2 \quad v_1 \text{ is different than } v_2}{A; e_1 \mathrel{!}= e_2 \longrightarrow \textbf{true}} \qquad \text{NotEq-False}\ \frac{A; e_1 \longrightarrow v \quad A; e_2 \longrightarrow v}{A; e_1 \mathrel{!}= e_2 \longrightarrow \textbf{false}}$$

$$\text{BinOp-Int}\ \frac{A; e_1 \longrightarrow n_1 \quad A; e_2 \longrightarrow n_2 \quad v \text{ is } n_1 \oplus n_2}{A; e_1 \oplus e_2 \longrightarrow v} \qquad \text{BinOp-Bool}\ \frac{A; e_1 \longrightarrow b_1 \quad A; e_2 \longrightarrow b_2 \quad b_3 \text{ is } b_1 \odot b_2}{A; e_1 \odot e_2 \longrightarrow b_3}$$

$$\text{Unary-Not}\ \frac{A; e_1 \longrightarrow b_1 \quad b_2 \text{ is } \neg b_1}{A; !e_1 \longrightarrow b_2}$$

4 Statement Semantics

This part of the formal semantics corresponds to the implementation of your `eval_stmt` function. That function has type `environment -> stmt -> environment`. In the semantics, we write the judgment $A; s \longrightarrow A'$, where A represents the input environment, s represents the statement to execute, and A' represents the final output environment. Statements are different from expressions in that they do not "return" anything themselves; instead, their impact occurs by modifying the environment (by assigning to variables).

4.1. Abstract syntax grammar. Statements are defined by the following grammar:

Statements s ::= `skip` | s;s | `if` e s s | `while` e s | `int` x | `bool` x | $x = e$

In the project itself there is also a statement form for printing; we leave that out of the formal semantics.

4.2. Rules.

Variables. In Small C, you have to declare a variable, with its type, before you use it. When declared, it will be initialized to a default value. You cannot declare the same variable twice.

$$\text{Declare-Int}\ \frac{A(x) \text{ is not defined}}{A; \texttt{int}\ x \longrightarrow A[x \mapsto 0]} \qquad \text{Declare-Bool}\ \frac{A(x) \text{ is not defined}}{A; \texttt{bool}\ x \longrightarrow A[x \mapsto \textbf{false}]}$$

Assigning to variables must respect their declared type. In particular, you cannot assign to a variable you have not declared, and you cannot write a boolean to a variable declared as an int, or vice versa.

$$\text{Assign-Int}\ \frac{\begin{array}{c}A(x) = n \text{ (for some } n) \\ A; e \longrightarrow n_1\end{array}}{A; x = e \longrightarrow A[x \mapsto n_1]} \qquad \text{Assign-Bool}\ \frac{\begin{array}{c}A(x) = b \text{ (for some } b) \\ A; e \longrightarrow b_1\end{array}}{A; x = e \longrightarrow A[x \mapsto b_1]}$$

Control Flow. Here are the rules for the different control flow constructs.

$$\text{Nop}\ \frac{}{A; \texttt{skip} \longrightarrow A} \qquad \text{Sequence}\ \frac{\begin{array}{c}A; s_1 \longrightarrow A_1 \\ A_1; s_2 \longrightarrow A_2\end{array}}{A; s_1; s_2 \longrightarrow A_2}$$

$$\text{If-True}\ \frac{\begin{array}{c}A; e \longrightarrow \textbf{true} \\ A; s_1 \longrightarrow A_1\end{array}}{A; \texttt{if}\ e\ s_1\ s_2 \longrightarrow A_1} \qquad \text{If-False}\ \frac{\begin{array}{c}A; e \longrightarrow \textbf{false} \\ A; s_2 \longrightarrow A_2\end{array}}{A; \texttt{if}\ e\ s_1\ s_2 \longrightarrow A_2}$$

$$\text{While-True}\ \frac{\begin{array}{c}A; e \longrightarrow \textbf{true} \\ A; s \longrightarrow A_1 \\ A_1; \texttt{while}\ e\ s \longrightarrow A_2\end{array}}{A; \texttt{while}\ e\ s \longrightarrow A_2} \qquad \text{While-False}\ \frac{A; e \longrightarrow \textbf{false}}{A; \texttt{while}\ e\ s \longrightarrow A}$$