SmallC Formal Operational Semantics

Fall 2019

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.

The semantics have been extended to add function declaration statements and function call expressions.

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	::=	**true** | **false**
Values	v	::=	$n \mid b$
Expressions	e	::=	$v \mid !e \mid e \oplus e \mid e \odot e \mid e == e \mid e \mathrel{!}= e$

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. Functions and Lazy evaluation. In languages like ocaml functions are first class objects, that it can be a value. In `C` there are function pointers that allow references to functions to be passed. In this language, however, they are are not values, instead function declarations update a global environment for functions, F.

While the language lacks first class functions it has lazy evaluation. In C and other languages, functions use call-by-value semantics, which evaluates each argument before passing the resulting values. Languages like Haskell use call-by-name, pass each expression, which is only evaluated if the corresponding parameter is used.

Listing 1: Lazy evaluation example

```
int infiniteLoop() {
  while(true){}
  return 0;
}

int russianRoulette(int a, int b){
  if (a == 0){
    return b;
  }
  return a;
}

int main() {
  int x;
  x = russianRoulette(1, infiniteLoop());
}
```

In the example above, with call-by-value semantics when calling `russianRoulette`, the `infiniteLoop` function call expression would be called and the program would crash. However, with call-by-name the expression is passed to `russianRoulette` and can be referred to with `b`. Since `a != 0`, `b` is never evaluated and the function returns `a`. OCaml does provide a `Lazy` module, which allows expressions to explicitly become lazy and must be forced to evaluate in the future.

3.2.1 Thunks

To allow for calling by name a runtime value stores the environment needed to evaluate the expression and the code that needs to be executed. This construct is called a thunk, which is a executable block that stores the expression and its context. The programming language Haskell notably uses thunks This means that while the evaluation of an expression can be delayed there is extra memory overhead. Haskell is a statically typed language, which means that before the program runs every expression is typed checked. However, since Small C is an interpreted language the types are not checked until the expression is evaluated.

For example, consider the following

```
bool foo(int x) {
  bool y;
  y = x;
  return y;
}
int main(){
  foo(true);
}
```

Since `true` becomes a thunk the expression isn't evaluated until it is referenced by the assign which expects it to be a boolean. However, since `x` was declared to be an `int` this is incorrect. To solve this the expected type of the thunk must be part of the data structure and checked when the thunk is evaluated.

3.3. Updated AST. We need to add function calls and Thunks. There is also a separate global environment, F that contains the function declarations.

DataTypes	typ	::=	$int \mid bool$
Thunk	t	::=	$environment * e * typ$
Values	v	::=	$\ldots \mid t$
Expressions	e	::=	$\ldots \mid \texttt{id}(e_1, \ldots, e_n)$

3.4. Rules. Here are the axioms.

$$\text{Id}\ \frac{A(x) = v}{A; x \longrightarrow v} \qquad \text{Id-Thunk}\ \frac{\begin{array}{c}A(x) = t \\ A; t \rightarrow v\end{array}}{A; x \longrightarrow v} \qquad \text{Int}\ \frac{}{A; n \longrightarrow n}$$

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

Here are the rest of the rules for expressions.

$$\text{Thunk-Int}\ \frac{A'; e \longrightarrow n}{A; (A', e, \texttt{int}) \longrightarrow n} \qquad \text{Thunk-Bool}\ \frac{A'; e \longrightarrow b}{A; (A', e, \texttt{bool}) \longrightarrow b}$$

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

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

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

$$\text{Unary-Not}\ \frac{\begin{array}{c} A; e_1 \longrightarrow b_1 \\ b_2 \text{ is } \neg b_1 \end{array}}{A; !e_1 \longrightarrow b_2}$$

$$\text{FunctionCall-Int}\ \frac{\begin{array}{c} F(\texttt{id}) = (\texttt{int}, [(id_1, typ_1), \ldots, (id_n, typ_n)], s) \\ A' = id_1 : (A, e_1, typ_1), \ldots, id_n : (A, e_n, typ_n) \\ A'; s \to A'[\text{“}{\sim}ret\text{”} \mapsto n] \end{array}}{F, A; \texttt{id}(e_1, \ldots, e_n) \longrightarrow n}$$

$$\text{FunctionCall-Bool}\ \frac{\begin{array}{c} F(\texttt{id}) = (\texttt{bool}, [(id_1, typ_1), \ldots, (id_n, typ_n)], s) \\ A' = id_1 : (A, e_1, typ_1), \ldots, id_n : (A, e_n, typ_n) \\ A'; s \to A'[\text{“}{\sim}ret\text{”} \mapsto b] \end{array}}{F, A; \texttt{id}(e_1, \ldots, e_n) \longrightarrow b}$$

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:

$$\begin{array}{lrcl} \text{Statements} & s & ::= & \texttt{skip} \mid s;s \mid \texttt{if}\ e\ s\ s \mid \texttt{while}\ e\ s \mid \texttt{int}\ x \mid \texttt{bool}\ x \mid x = e \\ & & & \mid\ \texttt{int id}(\texttt{typ}_1\ x_1, \ldots, \texttt{typ}_n\ x_n))\ \{s\} \\ & & & \mid\ \texttt{bool id}(\texttt{typ}_1\ x_1, \ldots, \texttt{typ}_n\ x_n))\ \{s\} \end{array}$$

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.

$$\text{Nop}\ \frac{}{A;\ \texttt{skip} \longrightarrow A} \qquad \text{Sequence}\ \frac{A;\ s_1 \longrightarrow A_1 \quad A_1;\ s_2 \longrightarrow A_2}{A;\ s_1;\ s_2 \longrightarrow A_2}$$

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

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

$$\text{DoWhile-True}\ \frac{A;\ s \longrightarrow A_1 \quad A_1;\ e \longrightarrow \textbf{true} \quad A_1;\ \texttt{dowhile}\ s\ e \longrightarrow A_2}{A;\ \texttt{dowhile}\ s\ e \longrightarrow A_2} \qquad \text{DoWhile-False}\ \frac{A;\ s \longrightarrow A_1 \quad A_1;\ e \longrightarrow \textbf{false}}{A;\ \texttt{dowhile}\ s\ e \longrightarrow A_1}$$

$$\text{For-Go}\ \frac{\begin{array}{c}A;\ e_1 \longrightarrow n_1 \quad A;\ e_2 \longrightarrow n_2 \quad A[x \mapsto n_1];\ s \longrightarrow A_1 \\ A_1, x = x + 1 \longrightarrow A_2 \quad A_2;\ x \longrightarrow n_3 \quad A_2;\ \texttt{for}\ x\ n_3\ n_2\ s \longrightarrow A' \quad n_1 \leq n_2\end{array}}{A;\ \texttt{for}\ x\ e_1\ e_2\ s \longrightarrow A'} \qquad \text{For-Stop}\ \frac{A;\ e_1 \longrightarrow n_1 \quad A;\ e_2 \longrightarrow n_2 \quad n_1 > n_2}{A;\ \texttt{for}\ x\ e_1\ e_2\ s \longrightarrow A[x \mapsto n_1]}$$

Functions. Here are the rules for function declarations and function calls

$$\text{FunctionDeclaration-Int}\ \frac{f = (\texttt{int}, [(x_1, \texttt{typ}_1) \ldots (x_n, \texttt{typ}_n)], s)}{F, A;\ \texttt{int id}(\texttt{typ}_1\ x_1, \ldots, \texttt{typ}_n\ x_n))\ \{s\} \longrightarrow F[\texttt{id} \mapsto f], A}$$

$$\text{FunctionDeclaration-Bool}\ \frac{f = (\texttt{bool}, [(x_1, \texttt{typ}_1) \ldots (x_n, \texttt{typ}_n)], s)}{F, A;\ \texttt{bool id}(\texttt{typ}_1\ x_1, \ldots, \texttt{typ}_n\ x_n))\ \{s\} \longrightarrow F[\texttt{id} \mapsto f], A}$$