Statements, Stack Machine, Stack Machine Compiler (the first draft)

Dmitry Boulytchev

1 Statements

More interesting language — a language of simple statements:

$$\begin{array}{rcl} \mathscr{S} & = & \mathscr{X} := \mathscr{E} \\ & & \text{read} \ \ (\mathscr{X}) \\ & & \text{write} \ \ (\mathscr{E}) \\ & & \mathscr{S}\,;\mathscr{S} \end{array}$$

Here $\mathscr{E}, \mathscr{X}$ stand for the sets of expressions and variables, as in the previous lecture.

Again, we define the semantics for this language

$$[\![\bullet]\!]_{\mathscr{S}} : \mathscr{S} \mapsto \mathbb{Z}^* \to \mathbb{Z}^*$$

with the semantic domain of partial functions from integer strings to integer strings. This time we will use big-step operational semantics: we define a ternary relation "$\Rightarrow$"

$$\Rightarrow \subseteq \mathscr{C} \times \mathscr{S} \times \mathscr{C}$$

where $\mathscr{C} = \Sigma \times \mathbb{Z}^* \times \mathbb{Z}^*$ — a set of all configurations during a program execution. We will write $c_1 \overset{S}{\Rightarrow} c_2$ instead of $(c_1, S, c_2) \in \Rightarrow$ and informally interpret the former as "the execution of a statement S in a configuration c_1 completes with the configuration c_2". The components of a configuration are state, which binds (some) variables to their values, and input and output streams, represented as (finite) strings of integers.

The relation "$\Rightarrow$" is defined by the following deductive system (see Fig. 1). The first three rules are axioms as they do not have any premises. Note, according to these rules sometimes a program cannot do a step in a given configuration: a value of an expression can be undefined in a given state in rules Assign and Write, and there can be no input value in rule Read. This style of a semantics description is called big-step operational semantics, since the results of a computation are immediately observable at the right hand side of "$\Rightarrow$" and, thus,

$$\langle s,i,o\rangle \xRightarrow{\text{x := e}} \langle s[x \leftarrow [\![e]\!]_{\mathscr{E}}\, s], i, o\rangle \qquad [\text{Assign}]$$

$$\langle s, z :: i, o\rangle \xRightarrow{\text{read (x)}} \langle s[x \leftarrow z], i, o\rangle \qquad [\text{Read}]$$

$$\langle s,i,o\rangle \xRightarrow{\text{write (e)}} \langle s,i,o@[\![e]\!]_{\mathscr{E}}\, s\rangle \qquad [\text{Write}]$$

$$\frac{c_1 \xRightarrow{s_1} c',\ c' \xRightarrow{s_2} c_2}{c_1 \xRightarrow{s_1;s_2} c_2} \qquad [\text{Seq}]$$

Figure 1: Big-step operational semantics for statements

the computation is performed in a single "big" step. And, again, this style of a semantic description can be used to easily implement a reference interpreter.

With the relation "$\Rightarrow$" defined we can abbreviate the "surface" semantics for the language of statements:

$$\forall S \in \mathscr{S}, \forall i \in \mathbb{Z}^*\ :\ [\![S]\!]_{\mathscr{S}} i = o \Leftrightarrow \langle \bot, i, \varepsilon\rangle \xRightarrow{S} \langle \_, \_, o\rangle$$

2 Stack Machine

Stack machine is a simple abstract computational device, which can be used as a convenient model to constructively describe the compilation process.

In short, stack machine operates on the same configurations, as the language of statements, plus a stack of integers. The computation, performed by the stack machine, is controlled by a program, which is described as follows:

$$\begin{array}{rcl} \mathscr{I} & = & \text{BINOP} \otimes \\ & & \text{CONST}\ \mathbb{N} \\ & & \text{READ} \\ & & \text{WRITE} \\ & & \text{LD}\ \mathscr{X} \\ & & \text{ST}\ \mathscr{X} \\ \mathscr{P} & = & \varepsilon \\ & & \mathscr{I}\,\mathscr{P} \end{array}$$

Here the syntax category $\mathscr{I}$ stands for instructions, $\mathscr{P}$ — for programs; thus, a program is a finite string of instructions.

The semantics of stack machine program can be described, again, in the form of big-step operational semantics. This time the set of stack machine configurations is

$$\mathscr{C}_{SM} = \mathbb{Z}^* \times \mathscr{C}$$

$$c \xrightarrow{\varepsilon} c \qquad [\text{Stop}_{SM}]$$

$$\frac{\langle (x \oplus y) :: st, c\rangle \xrightarrow{p} c'}{\langle y :: x :: st, c\rangle \xrightarrow{(\text{BINOP}\ \otimes) p} c'} \qquad [\text{Binop}_{SM}]$$

$$\frac{\langle z :: st, c\rangle \xrightarrow{p} c'}{\langle st, c\rangle \xrightarrow{(\text{CONST}\ z) p} c'} \qquad [\text{Const}_{SM}]$$

$$\frac{\langle z :: st, \langle s, i, o\rangle\rangle \xrightarrow{p} c'}{\langle st, \langle s, z :: i, o\rangle\rangle \xrightarrow{(\text{READ}) p} c'} \qquad [\text{Read}_{SM}]$$

$$\frac{\langle st, \langle s, i, o @ z\rangle\rangle \xrightarrow{p} c'}{\langle z :: st, \langle s, i, o\rangle\rangle \xrightarrow{(\text{WRITE}) p} c'} \qquad [\text{Write}_{SM}]$$

$$\frac{\langle (s\ x) :: st, \langle s, i, o\rangle\rangle \xrightarrow{p} c'}{\langle st, \langle s, i, o\rangle\rangle \xrightarrow{(\text{LD}\ x) p} c'} \qquad [\text{LD}_{SM}]$$

$$\frac{\langle st, \langle s[x \leftarrow z], i, o\rangle\rangle \xrightarrow{p} c'}{\langle z :: st, \langle s, i, o\rangle\rangle \xrightarrow{(\text{ST}\ x) p} c'} \qquad [\text{ST}_{SM}]$$

Figure 2: Big-step operational semantics for stack machine

where the first component is a stack, and the second — a configuration as in the semantics of statement language. The rules are shown on Fig. 2; note, now we have one axiom and six inference rules (one per instruction).

As for the statement, with the aid of the relation "$\Rightarrow$" we can define the surface semantics of stack machine:

$$\forall p \in \mathscr{P}, \forall i \in \mathbb{Z}^* \ :\ [\![p]\!]_{SM}\ i = o \Leftrightarrow \langle \varepsilon, \langle \bot, i, \varepsilon\rangle\rangle \xrightarrow{p} \langle \_, \langle \_, \_, o\rangle\rangle$$

3 A Compiler for the Stack Machine

A compiler of the statement language into the stack machine is a total mapping

$$[\![\bullet]\!]_{comp} : \mathscr{S} \mapsto \mathscr{P}$$

We can describe the compiler in the form of denotational semantics for the source language. In fact, we can treat the compiler as a static semantics, which maps each program into its stack machine equivalent.

As the source language consists of two syntactic categories (expressions and statments), the compiler has to be "bootstrapped" from the compiler for expressions $[\![\bullet]\!]^{\mathscr{E}}_{comp}$:

$$\begin{array}{rcl} [\![x]\!]^{\mathscr{E}}_{comp} & = & [\mathrm{LD}\ x] \\ [\![n]\!]^{\mathscr{E}}_{comp} & = & [\mathrm{CONST}\ n] \\ [\![A \otimes B]\!]^{\mathscr{E}}_{comp} & = & [\![A]\!]^{\mathscr{E}}_{comp} @ [\![B]\!]^{\mathscr{E}}_{comp} @ [\mathrm{BINOP}\ \otimes] \end{array}$$

And now the main dish:

$$\begin{array}{rcl} [\![x{:=}e]\!]_{comp} & = & [\![e]\!]^{\mathscr{E}}_{comp} @ [\mathrm{ST}\ x] \\ [\![\mathrm{read}\ \ (x)]\!]_{comp} & = & [\mathrm{READ};\ \mathrm{ST}\ x] \\ [\![\mathrm{write}\ \ (e)]\!]_{comp} & = & [\![e]\!]^{\mathscr{E}}_{comp} @ [\mathrm{WRITE}] \\ [\![S_1; S_2]\!]_{comp} & = & [\![S_1]\!]_{comp} @ [\![S_2]\!]_{comp} \end{array}$$