Introduction to Programming Languages, Compilers and Tools

Dmitry Boulytchev

November 19, 2019

1 Introduction: Languages, Semantics, Interpreters, Compilers

1.1 Language and semantics

A language is a collection of programs. A program is an *abstract syntax tree* (AST), which describes the hierarchy of constructs. An abstract syntax of a programming language describes the format of abstract syntax trees of programs in this language. Thus, a language is a set of constructive objects, each of which can be constructively manipulated.

The semantics of a language $\mathscr{L}$ is a total map

$$\llbracket\bullet\rrbracket_{\mathscr{L}} : \mathscr{L} \to \mathscr{D}$$

where $\mathscr{D}$ is some *semantic domain*. The choice of the domain is at our command; for example, for Turing-complete languages $\mathscr{D}$ can be the set of all partially-recursive (computable) functions.

1.2 Interpreters

In reality, the semantics often is described using *interpreters*:

$$eval : \mathscr{L} \to \texttt{Input} \to \texttt{Output}$$

where `Input` and `Output` are sets of (all possible) inputs and outputs for the programs in the language $\mathscr{L}$. We claim *eval* to possess the following property

$$\forall p \in \mathscr{L}, \forall x \in \texttt{Input} : \llbracket p\rrbracket_{\mathscr{L}}\, x = eval\; p\; x$$

In other words, an interpreter takes a program and its input as arguments, and returns what the program would return, being run on that argument. The equality in the definitional property of an interpreter has to be read "if the right hand side is defined, then the left hand side is defined, too, and their values coinside", and vice-versa.

Why interpreters are so important? Because they can be written as programs in a *meta-lanaguge*, or a language of implementation. For example, if we take ocaml as a language of implementation, then an interpreter of a language $\mathscr{L}$ is some ocaml program *eval*, such that

$$\forall p \in \mathscr{L}, \forall x \in \texttt{Input} : [\![p]\!]_{\mathscr{L}}\, x = [\![eval]\!]_{\text{ocaml}}\, p\, x$$

How to define $[\![\bullet]\!]_{\text{ocaml}}$? We can write an interpreter in some other language. Thus, a *tower* of meta-languages and interpreters comes into consideration. When to stop? When the meta-language is simple enough for intuitive understanding (in reality: some math-based frameworks like operational, denotational or game semantics, etc.)

Pragmatically: if you have a good implementation of a good programming language you trust, you can write interpreters of other languages.

1.3 Compilers

A compiler is just a language transformer

$$comp : \mathscr{L} \to \mathscr{M}$$

for two languages $\mathscr{L}$ and $\mathscr{M}$; we expect a compiler to be total and to possess the following property:

$$\forall p \in \mathscr{L} \;\; [\![p]\!]_{\mathscr{L}} = [\![comp\ p]\!]_{\mathscr{M}}$$

Again, the equality in this definition is understood functionally. The property itself is called a *complete* (or full) correctness. In reality compilers are *partially* correct, which means, that the domain of compiled programs can be wider.

And, again, we expect compilers to be defined in terms of some implementation language. Thus, a compiler is a program (in, say, ocaml), such, that its semantics in ocaml possesses the following property (fill the rest yourself).

1.4 The first example: language of expressions

Abstract syntax:

$$\begin{array}{rcll}
\mathscr{X} & = & \{x, y, z, \dots\} & \text{(variables)} \\
\otimes & = & \{+, -, \times, /, \%, <, \leq, >, \geq, =, \neq, \vee, \wedge\} & \text{(binary operators)} \\
\mathscr{E} & = & \mathscr{X} & \text{(expressions)} \\
 & & \mathbb{N} & \\
 & & \mathscr{E} \otimes \mathscr{E} &
\end{array}$$

Semantics of expressions:

- state $\sigma : \mathscr{X} \to \mathbb{Z}$ assigns values to (some) variables;
- semantics $[\![\bullet]\!]$ assigns each expression a total map $\Sigma \to \mathbb{Z}$, where Σ is the set of all states.

Empty state Λ: undefined for any variable.
Denotational style of semantic description:

$$\begin{array}{rcll}
[\![n]\!] & = & \lambda\sigma.n & , \quad n \in \mathbb{N} \\
[\![x]\!] & = & \lambda\sigma.\sigma x & , \quad x \in \mathscr{X} \\
[\![A \otimes B]\!] & = & \lambda\sigma.([\![A]\!]\sigma \oplus [\![B]\!]\sigma) & , \quad A, B \in \mathscr{E}
\end{array}$$

$\otimes$	$\oplus$ in ocaml	
$+$	`+`	
$-$	`-`	
$\times$	`*`	
$/$	`/`	
%	**`mod`**	
$<$	`<`	see note 1 below
$>$	`>`	see note 1 below
$\leq$	`<=`	see note 1 below
$\geq$	`>=`	see note 1 below
$=$	`=`	see note 1 below
$\neq$	`<>`	see note 1 below
$\wedge$	`&&`	see note 2 below
$\vee$	`||`	see note 2 below

Note 1: the result is converted into integers (true $\to$ 1, false $\to$ 0).

Note 2: the arguments are converted to booleans (0 $\to$ false, not 0 $\to$ true), the result is converted to integers as in the previous note.

Important observations:

1. $[\![\bullet]\!]$ is defined *compositionally*: the meaning of an expression is defined in terms of meanings of its proper subexpressions. This is an important property of denotational style.

2. $[\![\bullet]\!]$ is total, since it takes into account all possible ways to deconstruct any expression.

3. $[\![\bullet]\!]$ is deterministic: there is no way to assign different meanings to the same expression, since we deconstruct each expression unambiguously.

4. $\otimes$ is an element of language *syntax*, while $\oplus$ is its interpretation in the metalanguage of semanic description (simpler: in the language of interpreter implementation).

5. This concrete semantics is *strict*: for a binary operator both its arguments are evaluated unconditionally; thus, for example, $1 \vee \mathtt{x}$ is undefined in empty state.

2 Statements, Stack Machine, Stack Machine Compiler

2.1 Statements

More interesting language — a language of simple statements:

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

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, 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:

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

3 Stack Machine

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

$$\langle\sigma, \iota, o\rangle \xRightarrow{\texttt{x := } e} \langle\sigma[x \leftarrow [\![e]\!]_{\mathscr{E}}\,\sigma], \iota, o\rangle \qquad [\textsc{Assign}]$$

$$\langle\sigma, z\iota, o\rangle \xRightarrow{\texttt{read}\,(x)} \langle\sigma[x \leftarrow z], \iota, o\rangle \qquad [\textsc{Read}]$$

$$\langle\sigma, \iota, o\rangle \xRightarrow{\texttt{write}\,(e)} \langle\sigma, \iota, o([\![e]\!]_{\mathscr{E}}\,\sigma)\rangle \qquad [\textsc{Write}]$$

$$\frac{c_1 \xRightarrow{S_1} c' \qquad c' \xRightarrow{S_2} c_2}{c_1 \xRightarrow{S_1\,;S_2} c_2} \qquad [\textsc{Seq}]$$

Figure 1: Big-step operational semantics for statements

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:

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

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

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

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 "⇒" we can define the surface semantics of stack machine:

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

3.1 A Compiler for the Stack Machine

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

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

$$c \xRightarrow{\varepsilon} c \qquad [\textsc{Stop}_{SM}]$$

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

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

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

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

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

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

Figure 2: Big-step operational semantics for stack machine

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 $[\![\bullet]\!]^{\mathscr{E}}_{comp}$:

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

And now the main dish:

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

$$\begin{array}{rcll}
[\![n]\!] & = & \lambda\sigma.n & [\textsc{Const}] \\
[\![x]\!] & = & \lambda\sigma.\sigma x & [\textsc{Var}] \\
[\![A \otimes B]\!] & = & \lambda\sigma.([\![A]\!]\sigma \oplus [\![B]\!]\sigma) & [\textsc{Binop}]
\end{array}$$

(a) Denotational semantics for expressions

$$c \xRightarrow{\varepsilon} c \qquad [\textsc{Stop}_{SM}]$$

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

$$\frac{\langle z :: st, s\rangle \xRightarrow{p} c'}{\langle st, s\rangle \xRightarrow{(\texttt{CONST}\ z)p} c'} \qquad [\textsc{Const}_{SM}]$$

$$\frac{\langle (s\,x) :: st, s\rangle \xRightarrow{p} c'}{\langle st, s\rangle \xRightarrow{(\texttt{LD}\ x)p} c'} \qquad [\textsc{LD}_{SM}]$$

(b) Big-step operational semantics for stack machine

$$\begin{array}{rcll}
[\![x]\!]^{\mathscr{E}}_{comp} & = & [\texttt{LD}\ x] & [\textsc{Var}_{comp}] \\
[\![n]\!]^{\mathscr{E}}_{comp} & = & [\texttt{CONST}\ n] & [\textsc{Const}_{comp}] \\
[\![A \otimes B]\!]^{\mathscr{E}}_{comp} & = & [\![A]\!]^{\mathscr{E}}_{comp} \texttt{@} [\![B]\!]^{\mathscr{E}}_{comp} \texttt{@} [\texttt{BINOP} \otimes]) & [\textsc{Binop}_{comp}]
\end{array}$$

(c) Compilation

Figure 3: All relevant definitions

4 Structural Induction

We have considered two languages (a language of expressions $\mathscr{E}$ and a language of stack machine programs $\mathscr{P}$), and a compiler from the former to the latter. It can be formally proven, that the compiler is (fully) correct in the sense, given in the lecture 1. Due to the simplicity of the languages, the proof technique — *structural induction* — is simple as well.

First, we collect all needed definitions in one place (see Fig. 3). We simplified the description of stack machine semantics a little bit: first, we dropped off all instructions, which cannot be generated by the expression compiler, and then, we removed the input and output streams from the stack machine configurations, since they are never affected by the remaining instructions.

Lemma 1. *(Determinism) Let p be an arbitrary stack machine program, and let c, c_1 and c_2 be arbitrary configurations. Then*

$$c \xRightarrow{p} c_1 \wedge c \xRightarrow{p} c_2 \Rightarrow c_1 = c_2$$

Proof. Induction on the structure of p.

Base case. If $p = \varepsilon$, then, by the rule $\textsc{Stop}_{SM}$, $c_1 = c$ and $c_2 = c$. Since no other rule can be applied, we're done.

Induction step. If $p = \iota p'$, then, by condition, we have

$$\frac{c' \xRightarrow{p'} c_1}{c \xRightarrow{\iota p'} c_1}$$

and

$$\frac{c'' \xRightarrow{p'} c_2}{c \xRightarrow{\iota p'} c_2}$$

where c' and c'' depend only on c and ι. By the case analysis on ι we conclude, that $c' = c''$. Since p' is shorter, than p, we can apply the induction hypothesis, which gives us $c_1 = c_2$. □

Lemma 2. *(Compositionality) Let $p = p_1 p_2$ be an arbitrary stack machine program, subdivided into arbitrary subprograms p_1 and p_2. Then,*

$$\forall c_1, c_2 : \; c_1 \xRightarrow{p} c_2 \;\Leftrightarrow\; \exists c' : \; c_1 \xRightarrow{p_1} c' \wedge c' \xRightarrow{p_2} c_2$$

Proof. Induction on the structure of p.

Base case. The base case $p = \varepsilon$ is trivial: use the rule $\textsc{Stop}_{SM}$ and get $c' = c_2 = c_1$.

Induction step. When $p = \iota p'$, then there are two cases:

- Either $p_1 = \varepsilon$, then $c' = c_1$ trivially by the rule $\textsc{Stop}_{SM}$, and we're done.
- Otherwise $p_1 = \iota p'_1$, and, thus, $p = \iota p'_1 p_2$. In order to prove the lemma, we need to prove two implications:

  1. Let $c_1 \xRightarrow{p = \iota p'_1 p_2} c_2$. Technically, we need here to consider three cases (one for each type of the instruction ι), but in all cases the outcome would be the same: we have the picture

  $$\frac{c'' \xRightarrow{p'_1 p_2} c_2}{c_1 \xRightarrow{p = \iota p'_1 p_2} c_2}$$

  where c'' depends only on ι and c_1. Since $p'_1 p_2$ is shorter, than p, we can apply the induction hypothesis, which gives us a configuration c', such, that $c'' \xRightarrow{p'_1} c'$ and $c' \xRightarrow{p_2} c_2$. The observation $c_1 \xRightarrow{\iota p'_1} c'$ concludes the proof (note, we implicitly use determinism here).

2. Let there exists c', such that $c_1 \xRightarrow{\iota p'_1} c'$ and $c' \xRightarrow{p_2} c_2$. From the first relation we have

$$\frac{c'' \xRightarrow{p'_1} c'}{c_1 \xRightarrow{\iota p'_1} c'}$$

where c'' depends only on ι and c_1. Since $p'_1 p_2$ is shorter, than p, we can apply the induction hypothesis, which gives us $c'' \xRightarrow{p'_1 p_2} c_2$, and, thus, $c_1 \xRightarrow{\iota p'_1 p_2} c_2$ (again, we implicitly use determinism here).

□

Theorem 1. *(Correctness of compilation) Let $e \in \mathscr{E}$ be arbitrary expression, s — arbitrary state, and st — arbitrary stack. Then*

$$\langle st, s\rangle \xRightarrow{[\![e]\!]^{\mathscr{E}}_{comp}} \langle ([\![e]\!]\, s) :: st, s\rangle \textit{ iff } ([\![e]\!]\, s) \textit{ is defined}$$

Proof. Induction on the structure of e.

Base case. There are two subcases:

1. e is a constant z. Then:
   - $[\![e]\!]\, s = z$ for each state s;
   - $[\![e]\!]^{\mathscr{E}}_{comp} = [\texttt{CONST z}]$;
   - $\langle st, s\rangle \xRightarrow{[\texttt{CONST z}]} \langle z :: st, s\rangle$ for arbitrary st and s.

   This concludes the first base case.

2. e is a variable x. Then:
   - $[\![s]\!]\, s = s\,x$ for each state s, such that $s\,x$ is defined;
   - $[\![e]\!]^{\mathscr{E}}_{comp} = [\texttt{LD x}]$;
   - $\langle st, s\rangle \xRightarrow{[\texttt{CONST z}]} \langle (s\,x) :: st, s\rangle$ for arbitrary st and arbitrary s, such that $s\,x$ is defined.

   This concludes the second base case.

Induction step. Let e be $A \otimes B$. Then:

- $[\![A \otimes B]\!]s = [\![A]\!]s \oplus [\![B]\!]s$ for each state s, such that both $[\![A]\!]s$ and $[\![B]\!]s$ are defined;
- $[\![A \otimes B]\!]^{\mathscr{E}}_{comp} = [\![A]\!]^{\mathscr{E}}_{comp} \texttt{@} [\![B]\!]^{\mathscr{E}}_{comp} \texttt{@} [\texttt{BINOP}\oplus]$;

- by the inductive hypothesis, for arbitrary st and s

  $$\langle st, s\rangle \xRightarrow{[\![A]\!]^{\mathscr{E}}_{comp}} \langle ([\![A]\!]s) :: st, s\rangle \text{iff } ([\![A]\!]\, s) \text{ is defined}$$

  and

  $$\langle ([\![A]\!]s) :: st, s\rangle \xRightarrow{[\![B]\!]^{\mathscr{E}}_{comp}} \langle ([\![B]\!]s) :: ([\![A]\!]s) :: st, s\rangle \text{iff } ([\![A]\!]\, s) \text{ and } ([\![A]\!]\, s) \text{ are defined}$$

  Taking into account the semantics of BINOP$\otimes$ and applying the compositionality lemma, the theorem follows.

$\square$

Control Flow Constructs

4.1 Structural Control Flow

We add a few structural control flow constructs to the language:

$$\begin{array}{lcl} \mathscr{S} & += & \texttt{skip} \\ & & \texttt{if}\ \mathscr{E}\ \texttt{then}\ \mathscr{S}\ \texttt{else}\ \mathscr{S} \\ & & \texttt{while}\ \mathscr{E}\ \texttt{do}\ \mathscr{S} \end{array}$$

The big-step operational semantics is straightforward and is shown on Fig. 4.

In the concrete syntax for the constructs we add the closing keywords "**if**" and "**od**" as follows:

$$\begin{array}{c} \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od} \end{array}$$

4.2 Extended Stack Machine

In order to compile the extended language into a program for the stack machine the latter has to be extended. First, we introduce a set of label names

$$\mathscr{L} = \{l_1, l_2, \dots\}$$

Then, we add three extra control flow instructions:

$$\begin{array}{lcl} \mathscr{I} & += & \texttt{LABEL}\ \mathscr{L} \\ & & \texttt{JMP}\ \mathscr{L} \\ & & \texttt{CJMP}_x\ \mathscr{L},\ \text{where } x \in \{\texttt{nz}, \texttt{z}\} \end{array}$$

In order to give the semantics to these instructions, we need to extend the syntactic form of rules, used in the description of big-step operational smeantics. Instead of the rules in the form

$$c \xRightarrow{\texttt{skip}} c \qquad [\textsc{Skip}]$$

$$\frac{[\![e]\!]\,\sigma \neq 0 \qquad c \xRightarrow{S_1} c'}{c = \langle \sigma, \_, \_ \rangle \xRightarrow{\texttt{if}\; e\; \texttt{then}\; S_1\; \texttt{else}\; S_2} c'} \qquad [\textsc{If-True}]$$

$$\frac{[\![e]\!]\,\sigma = 0 \qquad c \xRightarrow{S_1} c'}{c = \langle \sigma, \_, \_ \rangle \xRightarrow{\texttt{if}\; e\; \texttt{then}\; S_1\; \texttt{else}\; S_2} c'} \qquad [\textsc{If-False}]$$

$$\frac{[\![e]\!]\,\sigma \neq 0 \qquad c \xRightarrow{S} c' \qquad c' \xRightarrow{\texttt{while}\; e\; \texttt{do}\; S} c''}{c = \langle \sigma, \_, \_ \rangle \xRightarrow{\texttt{while}\; e\; \texttt{do}\; S} c''} \qquad [\textsc{While-True}]$$

$$\frac{[\![e]\!]\,\sigma = 0}{c = \langle \sigma, \_, \_ \rangle \xRightarrow{\texttt{while}\; e\; \texttt{do}\; S} c} \qquad [\textsc{While-False}]$$

Figure 4: Big-step operational semantics for control flow statements

$$\frac{c \xRightarrow{p} c'}{c' \xRightarrow{p'} c''}$$

we use the following form

$$\frac{\Gamma \vdash c \xRightarrow{p} c'}{\Gamma' \vdash c' \xRightarrow{p'} c''}$$

where Γ, Γ' — *environments*. The structure of environments can be different in different cases; for now environment is just a program. Informally, the semantics of control flow instructions can not be described in terms of just a current instruction and current configuration — we need to take the whole program into account. Thus, the enclosing program is used as an environment.

Additionally, for a program P and a label l we define a subprogram $P[l]$, such that P is uniquely represented as $p'(\texttt{LABEL}\; l)P[l]$. In other words $P[l]$ is a unique suffix of P, immediately following the label l (if there are multiple (or no) occurrences of label l in P, then $P[l]$ is undefined).

All existing rules have to be rewritten — we need to formally add a $P \vdash \dots$ part everywhere. For the new instructions the rules are given on Fig. 5.

Finally, the top-level semantics for the extended stack machine can be redefined as follows:

$$\frac{P \vdash c \xRightarrow{p} c'}{P \vdash c \xRightarrow{(\texttt{LABEL}\ l)p} c'} \qquad [\textsc{Label}_{SM}]$$

$$\frac{P \vdash c \xRightarrow{P[l]} c'}{P \vdash c \xRightarrow{(\texttt{JMP}\ l)p} c'} \qquad [\text{JMP}_{SM}]$$

$$\frac{P \vdash c \xRightarrow{P[l]} c'}{P \vdash \langle z :: st, c\rangle \xRightarrow{(\texttt{CJMP}_{nz}\ l)p} c'},\ z \neq 0 \qquad [\text{CJMP}^{+}_{nz\,SM}]$$

$$\frac{P \vdash c \xRightarrow{p} c'}{P \vdash \langle z :: st, c\rangle \xRightarrow{(\texttt{CJMP}_{nz}\ l)p} c'},\ z = 0 \qquad [\text{CJMP}^{-}_{nz\,SM}]$$

$$\frac{P \vdash c \xRightarrow{P[l]} c'}{P \vdash \langle z :: st, c\rangle \xRightarrow{(\texttt{CJMP}_{z}\ l)p} c'},\ z = 0 \qquad [\text{CJMP}^{+}_{z}{}_{SM}]$$

$$\frac{P \vdash c \xRightarrow{p} c'}{P \vdash \langle z :: st, c\rangle \xRightarrow{(\texttt{CJMP}_{z}\ l)p} c'},\ z \neq 0 \qquad [\text{CJMP}^{-}_{z}{}_{SM}]$$

Figure 5: Big-step operational semantics for extended stack machine

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

4.3 Syntax Extensions

With the structural control flow constructs already implemented, it is rather simple to "saturate" the language with more elaborated control constructs, using the method of *syntactic extension*. Namely, we may introduce the following constructs

```
  if e_1 then s_1
elif e_2 then s_2
...
elif e_k then s_k
[ else s_{k+1} ]
  fi
```

and

```
for s_1 , e , s_2 do s_3 od
```

only at the syntactic level, directly parsing these constructs into the original abstract syntax tree, using the following conversions:

```
  if $e_1$ then $s_1$                 if $e_1$ then $s_1$
elif $e_2$ then $s_2$            else if $e_2$ then $s_2$
...                      $\rightsquigarrow$    ...
elif $e_k$ then $s_k$            else if $e_k$ then $s_k$
else $s_{k+1}$                   else $s_{k+1}$
fi                                    fi
                                      ...
                                      fi
```

```
  if $e_1$ then $s_1$                 if $e_1$ then $s_1$
elif $e_2$ then $s_2$            else if $e_2$ then $s_2$
...                      $\rightsquigarrow$    ...
elif $e_k$ then $s_k$            else if $e_k$ then $s_k$
fi                               else skip
                                      fi
                                      ...
                                      fi
```

```
for $s_1$, $e$, $s_2$ do $s_3$ od      $\rightsquigarrow$     $s_1$;
                                           while $e$ do
                                             $s_3$;
                                             $s_2$
                                           od
```

The advantage of syntax extension method is that it makes it possible to add certain constructs with almost zero cost — indeed, no steps have to be made in order to implement the extended constructs (besides parsing). Note, the semantics of extended constructs is provided for free as well (which is not always desirable). Another potential problem with syntax extensions is that they can easily provide unreasonable results. For example, one may be tempted to implement a post-condition loop using syntax extension:

```
repeat $s$ until $e$                  $\rightsquigarrow$     $s$;
                                           while $e == 0$ do
                                             $s$
                                           od
```

However, for nested **repeat** constructs the size of extended program is exponential w.r.t. the nesting depth, which makes the whole idea unreasonable.

5 Procedures

Procedures are unit-returning functions. We consider adding procedures as a separate step, since introducing full-fledged functions would require essential redefinition of the semantics for expressions; at the same time the code generation for funictions is a little trickier, than for procedures, so it is reasonable to split the implementation in two steps.

At the source level procedures are added as a separate syntactic category — definition $\mathscr{D}$:

$$\begin{array}{rcl}\mathscr{D} & = & \varepsilon \\ & & (\texttt{fun}\ \mathscr{X}\ (\mathscr{X}^*)\ \texttt{local}\ \mathscr{X}^*\ \{\mathscr{S}\})\mathscr{D}\end{array}$$

In other words, a definition is a (possibly empty) sequence of procedure descriptions. Each description consists of a name for the procedure, a list of names for its arguments and local variables, and a body (statement). In concrete syntax a single definition looks like

```
fun name (a_1, a_2, ..., a_k)
  local l_1, l_2, ..., l_n {
  s
}
```

where *name* — a name for the procedure, a_i — its arguments, l_i — local variables, s — body.

We also need to add a call statement to the language:

$$\mathscr{S} += \mathscr{X}\,(\mathscr{E}^*)$$

In a concrete syntax a call to a procedure f with arguments $e_1,\dots,e_k$ looks like

$$f\ (e_1,\dots,e_k)$$

Finally, we have to redefine the syntax category for programs at the top level:

$$\mathscr{L} = \mathscr{D}\mathscr{S}$$

In other words, we extend a statement with a set of definitions.

With procedures, we need to introduce the notion of *scope*. When a procedure is called, its arguments are associated with actual parameter values. A procedure is also in "posession" of its local variables. So, in principle, the context of a procedure execution is a set of arguments, local variables and, possibly, some other variables, for example, the global ones. However, the exact details of procedure context manipilation can differ essentially from language to language.

In our case, we choose a static scoping — each procedure, besides its arguments and local variables, has an access only to global variables. To describe this semantics, we need to change the definition of a state we've used so far:

$$\Sigma = (\mathscr{X} \to \mathbb{Z}) \times 2^{\mathscr{X}} \times (\mathscr{X} \to \mathbb{Z})$$

Now the state is a triple: a *global* state, a set of variables, and a *local* state. Informally, in a new state $\langle\sigma_g, S, \sigma_l\rangle$ S describes a set of local variables, σ_l — their values, and σ_g — the values of all other accessible variables.

We need to redefine all state-manipulation primitives; first, the valuation of variables:

$$\langle\sigma_g, S, \sigma_l\rangle\; x = \left\{\begin{array}{lcl}\sigma_g\, x & , & x \notin S\\ \sigma_l\, x & , & x \in S\end{array}\right.$$

Then, updating the state:

$$\langle\sigma_g, S, \sigma_l\rangle\,[x \leftarrow z] = \left\{\begin{array}{lcl}\langle\sigma_g[x \leftarrow z], S, \sigma_l\rangle & , & x \notin S\\ \langle\sigma_g, S, \sigma_l[x \leftarrow z]\rangle & , & x \in S\end{array}\right.$$

As an empty state, we take the following triple:

$$\Lambda = \langle\Lambda, \varnothing, \Lambda\rangle$$

Finally, we need two transformations for states:

$$\begin{array}{rcl}\textbf{enter}\;\langle\sigma_g, \_, \_\rangle\; S & = & \langle\sigma_g, S, \Lambda\rangle\\ \textbf{leave}\;\langle\sigma_g, \_, \_\rangle\;\langle\_, S, \sigma_l\rangle & = & \langle\sigma_g, S, \sigma_l\rangle\end{array}$$

The first one simlulates entering the new scope with a set of local variables S; the second one simulates leaving from an inner scope (described by the first state) to an outer one (described by the second).

All exising rules for big-step operational semantics have to be enriched by *functional environment* Γ, which binds procedure names to their definitions. As this binding never changes during the program interpretation, we need only to propagate this environment, adding $\Gamma \vdash ...$ for each transition "$... \Longrightarrow ...$". The only thing we need now is to describe the rule for procedure calls:

$$\frac{\texttt{fun}\; f\;(\bar{a})\;\texttt{local}\;\bar{l}\;\{S\} = \Gamma f \qquad \Gamma \vdash \left\langle \textbf{enter}\;\sigma\,(\bar{a}\bar{l})\overline{[a \leftarrow [\![e]\!]\sigma]}, \iota, o\right\rangle \xRightarrow{S} \langle\sigma', \iota', o'\rangle}{\Gamma \vdash \langle\sigma, \iota, o\rangle \xRightarrow{f(\bar{e})} \langle\textbf{leave}\;\sigma'\,\sigma, \iota', o'\rangle}\quad[\text{CALL}]$$

where $\Gamma f = \texttt{fun}\; f\;(\bar{a})\;\texttt{local}\;\bar{l}\;\{S\}$.

6 Extended Stack Machine

In order to support procedures and calls, we enrich the stack machine with three following instructions:

$$\begin{array}{rcl}\mathscr{I} & += & \texttt{BEGIN}\;\mathscr{X}^*\;\mathscr{X}^*\\ & & \texttt{CALL}\;\mathscr{X}\\ & & \texttt{END}\end{array}$$

Informally speaking, instruction `BEGIN` performs entering into the scope of a procedure; its operands are the lists of argument names and local variables; `END` leaves the scope and returns to the call site, and `CALL` performs the call itself.

We need to enrich the configurations for the stack machine as well:

$$\mathscr{C}_{SM} = (\mathscr{P} \times \Sigma) \times \mathbb{Z}^* \times \mathscr{C}$$

Here we added a *control stack* — a stack of pairs of programs and states. Informally, when performing the `CALL` instruction, we put the following program and current state on a stack to use them later on, when corresponding `END` instruction will be encountered. As all other instructions does not affect the control stack, it gets threaded through all rules of operational semantics unchanged.

Now we specify additional rules for the new instructions:

$$\frac{P \vdash \left\langle cs,\, st,\, \left\langle \mathbf{enter}\, \sigma\, (\bar{a}@\bar{l})\overline{[a \leftarrow z]},\, i,\, o \right\rangle \right\rangle \xRightarrow{p} c'}{P \vdash \langle cs,\, \bar{z}@st,\, \langle \sigma,\, i,\, o\rangle\rangle \xRightarrow{(\mathtt{BEGIN}\ \bar{a}\,\bar{l})p} c'} \qquad [\text{Begin}_{SM}]$$

$$\frac{P \vdash \langle (p,\sigma) :: cs,\, st,\, \langle \sigma,\, i,\, o\rangle\rangle \xRightarrow{P[f]} c'}{P \vdash \langle cs,\, st,\, \langle \sigma,\, i,\, o\rangle\rangle \xRightarrow{(\mathtt{CALL}\ f)p} c'} \qquad [\text{Call}_{SM}]$$

$$\frac{P \vdash \langle cs,\, st,\, \langle \mathbf{leave}\, \sigma\sigma',\, i,\, o\rangle\rangle \xRightarrow{p'} c'}{P \vdash \langle (p',\sigma') :: cs,\, st,\, \langle \sigma,\, i,\, o\rangle\rangle \xRightarrow{\mathtt{END}p} c'} \qquad [\text{EndRet}_{SM}]$$

$$P \vdash \langle \varepsilon,\, st,\, c\rangle \xRightarrow{\mathtt{END}p} \langle \varepsilon,\, st,\, c\rangle \qquad [\text{EndStop}_{SM}]$$

7 Functions

7.1 Functions in Expressions

At the syntax level function calls are introduced with the following construct:

$$\mathscr{E} \quad += \quad \mathscr{X}\,\mathscr{E}^*$$

In the concrete syntax function call looks conventional:

$f\ \ (e_1, e_2, \dots, e_k)$

where f — a function name, e_i — its actual parameters.

Surpisingly, such a mild extension results in a complete redefinition of the semantics. Indeed, as function body can perform arbitrary actions on state, input and output streams, expressions with function calls can now have side effects. In order to express these side effects we need to redefine the semantics completely, this time using big-step operational style.

We extend the configuration, used in the semantic description for statements, with a fourth component — an optional integer value:

$$\Phi \vdash \langle \sigma, i, o, \_ \rangle \xRightarrow{n}_{\mathscr{E}} \langle \sigma, i, o, n \rangle \qquad [\textsc{Const}]$$

$$\Phi \vdash \langle \sigma, i, o, \_ \rangle \xRightarrow{x}_{\mathscr{E}} \langle \sigma, i, o, \sigma\, x \rangle \qquad [\textsc{Var}]$$

$$\frac{\Phi \vdash c \xRightarrow{A}_{\mathscr{E}} c' \qquad \Phi \vdash c' \xRightarrow{B}_{\mathscr{E}} c''}{\Phi \vdash c \xRightarrow{A \otimes B}_{\mathscr{E}} \mathbf{ret}\ (\mathbf{val}\ c' \oplus \mathbf{val}\ c'')} \qquad [\textsc{Binop}]$$

$$\frac{\begin{array}{c}\Phi \vdash c_{j-1} \xRightarrow{e_j}_{\mathscr{E}} c_j = \left\langle \sigma_j, i_j, o_j, v_j \right\rangle \\ \Phi\, f = \texttt{fun}\ f\ (\overline{a})\ \texttt{local}\ \bar{l}\ \{s\} \\ \texttt{skip}, \Phi \vdash \left\langle \mathbf{enter}\ \sigma_k\ (\overline{a}@\bar{l})\ [\overline{a_j \leftarrow v_j}], i_k, o_k, \text{—} \right\rangle \xRightarrow{s} \langle \sigma', i', o', n \rangle\end{array}}{\Phi \vdash c_0 = \langle \sigma_0, \_, \_, \_ \rangle \xRightarrow{f(\overline{e_k})}_{\mathscr{E}} \langle \mathbf{leave}\ \sigma'\ \sigma_0, i', o', n \rangle} \qquad [\textsc{Call}]$$

Figure 6: Big-step Operational Semantics for Expressions

$$\mathscr{C} = \Sigma \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^?$$

This component will correspond to an optional return value for function/procedure calls (either integer value n or "—", nothing).

We introduce two primitives to make the semantic description shorter:

$$\mathbf{val}\ \langle s, i, o, v \rangle = v$$
$$\mathbf{ret}\ \langle s, i, o, \_ \rangle\ v = \langle s, i, o, v \rangle$$

The rules themselves are summarized in the Fig. 6. Note the use of double environment for evaluating the body of a function in the rule CALL; note also, that now semantics of expressions and statements are mutually recursive.

7.2 Return Statement

In order to make it possible to return values from procedures we add a return statement to the language of statements:

$$\mathscr{S} += \texttt{return}\ \mathscr{E}^?$$

And, again, this small addition leads to redefinition of the semantics in *continuation-passing style* (CPS).

First, we define the following meta-operator "$\diamond$" on statements:

$$\begin{array}{rcl} s \diamond \texttt{skip} & = & s \\ s_1 \diamond s_2 & = & s_1; s_2 \end{array}$$

Then, we add another environment component, K, in the description of semantic relation "$... \vdash ... \overset{...}{\Longrightarrow} ...$". Informally speaking, now

$$K, \Phi \vdash c \overset{s}{\Longrightarrow} c'$$

is read "the execution of s, immediately followed by K, in the configuration c results in the configuration c'". Statement K is called *continuation*. The rules themselves are shown on Fig. 7. Note, the rule for the call statement is exactly the same, as for the call expression.

8 Arrays and strings

An array can be represented as a pair: the length of the array and a mapping from indices to elements. If we denote $\mathscr{E}$ the set of elements then the set of all arrays $\mathscr{A}(\mathscr{E})$ can be defined as follows:

$$\mathscr{A}(\mathscr{E}) = \mathbb{N} \times (\mathbb{N} \to \mathscr{E})$$

For an array (n, f) we assume $\texttt{dom}\, f = [0\,..\,n-1]$. An element selection function:

$$\bullet[\bullet] : \mathscr{A}(\mathscr{E}) \to \mathbb{N} \to \mathscr{E}$$

$$(n, f)[i] = \left\{ \begin{array}{ccl} f\ i & , & i < n \\ \bot & , & \text{otherwise} \end{array} \right.$$

We represent arrays by *references*. Thus, we introduce a (linearly) ordered set of locations

$$\mathscr{L} = \{l_0, l_1, \dots\}$$

Now, the set of all values the programs operate on can be described as follows:

$$\mathscr{V} = \mathbb{Z} \uplus \mathscr{L}$$

Here, every value is either an integer, or a reference (some location). The disjoint union "$\uplus$" makes it possible to unambiguously discriminate between the shapes of each value. To access arrays, we introduce an abstraction of memory:

$$\mathscr{M} = \mathscr{L} \to \mathscr{A}(\mathscr{V})$$

We now add two more components to the configurations: a memory function μ and the first free memory location l_m, and define the following primitive:

$$\mathbf{mem}\ \langle s, \mu, l_m, i, o, v\rangle = \mu$$

which gives a memory function from a configuration.

$$\texttt{skip}, \Phi \vdash c \xRightarrow{\texttt{skip}} c \qquad [\textsc{SkipSkip}]$$

$$\frac{\texttt{skip}, \Phi \vdash c \xRightarrow{K} c'}{K, \Phi \vdash c \xRightarrow{\texttt{skip}} c'} \qquad [\textsc{Skip}]$$

$$\frac{\Phi \vdash c \xRightarrow{e}_{\mathscr{E}} \langle \sigma, i, o, n\rangle \quad \texttt{skip}, \Phi \vdash \langle \sigma[x \leftarrow n], i, o, \text{—}\rangle \xRightarrow{K} c'}{K, \Phi \vdash c \xRightarrow{x := e} c'} \qquad [\textsc{Assign}]$$

$$\frac{\Phi \vdash c \xRightarrow{e}_{\mathscr{E}} \langle \sigma, i, o, n\rangle \quad \texttt{skip}, \Phi \vdash \langle \sigma, i, o@[n], \text{—}\rangle \xRightarrow{K} c'}{K, \Phi \vdash c \xRightarrow{\texttt{write}\ (e)} c'} \qquad [\textsc{Write}]$$

$$\frac{\texttt{skip}, \Phi \vdash \langle \sigma[x \leftarrow z], i, o, \text{—}\rangle \xRightarrow{K} c'}{K, \Phi \vdash \langle \sigma, z : i, o, \text{—}\rangle \xRightarrow{\texttt{read}\ (x)} c'} \qquad [\textsc{Read}]$$

$$\frac{s_2 \diamond K, \Phi \vdash c \xRightarrow{s_1} c'}{K, \Phi \vdash c \xRightarrow{s_1;\, s_2} c'} \qquad [\textsc{Seq}]$$

$$\frac{\begin{array}{c}\Phi \vdash c \xRightarrow{e}_{\mathscr{E}} \langle \sigma, i, o, n\rangle \quad K, \Phi \vdash \langle \sigma, i, o, \text{—}\rangle \xRightarrow{s_1} c' \\ n \neq 0\end{array}}{K, \Phi \vdash c \xRightarrow{\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2} c'} \qquad [\textsc{IfTrue}]$$

$$\frac{\Phi \vdash c \xRightarrow{e}_{\mathscr{E}} \langle \sigma, i, o, 0\rangle \quad K, \Phi \vdash \langle \sigma, i, o, \text{—}\rangle \xRightarrow{s_2} c'}{K, \Phi \vdash c \xRightarrow{\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2} c'} \qquad [\textsc{IfFalse}]$$

$$\frac{\begin{array}{c}\Phi \vdash c \xRightarrow{e}_{\mathscr{E}} \langle \sigma, i, o, n\rangle \quad \texttt{while}\ e\ \texttt{do}\ s \diamond K, \Phi \vdash \langle \sigma, i, o, \text{—}\rangle \xRightarrow{s} c' \\ n \neq 0\end{array}}{K, \Phi \vdash c \xRightarrow{\texttt{while}\ e\ \texttt{do}\ s} c'} \qquad [\textsc{WhileTrue}]$$

$$\frac{\Phi \vdash c \xRightarrow{e}_{\mathscr{E}} \langle \sigma, i, o, 0\rangle \quad \texttt{skip}, \Phi \vdash \langle \sigma, i, o, \text{—}\rangle \xRightarrow{K} c'}{K, \Phi \vdash c \xRightarrow{\texttt{while}\ e\ \texttt{do}\ s} c'} \qquad [\textsc{WhileFalse}]$$

$$\frac{\begin{array}{c}\Phi \vdash c_{j-1} \xRightarrow{e_j}_{\mathscr{E}} c_j = \left\langle \sigma_j, i_j, o_j, v_j \right\rangle \\ \Phi\, f = \texttt{fun}\ f\ (\overline{a})\ \texttt{local}\ \overline{l}\ \{s\} \\ \texttt{skip}, \Phi \vdash \left\langle \textbf{enter}\ \sigma_k\ (\overline{a}@\overline{l})\ [\overline{a_j \leftarrow v_j}], i_k, o_k, \text{—}\right\rangle \xRightarrow{s} \langle \sigma', i', o', \text{—}\rangle \\ \texttt{skip}, \Phi \vdash \langle \textbf{leave}\ \sigma'\ \sigma_0, i', o', \text{—}\rangle \xRightarrow{K} \langle \sigma'', i'', o'', n\rangle\end{array}}{K, \Phi \vdash c_0 = \langle \sigma_0, \_, \_, \_\rangle \xRightarrow{f(\overline{e_k})} \langle \sigma'', i'', o'', n\rangle} \qquad [\textsc{Call}]$$

$$K, \Phi \vdash c \xRightarrow{\texttt{return}} c \qquad [\textsc{ReturnEmpty}]$$

$$\frac{\Phi \vdash c \xRightarrow{e}_{\mathscr{E}} c'}{K, \Phi \vdash c \xRightarrow{\texttt{return}\ e} c'} \qquad [\textsc{Return}]$$

Figure 7: Continuation-passing Style Semantics for Statements

$$\frac{\begin{array}{cc} \Phi \vdash c \xRightarrow{e}_{\mathscr{E}} c' & \Phi \vdash c' \xRightarrow{j}_{\mathscr{E}} c'' \\ l = \textbf{val}\, c' & j = \textbf{val}\, c'' \\ l \in \mathscr{L} & j \in \mathbb{N} \\ (n, f) = \textbf{mem}\, l & j < n \end{array}}{\Phi \vdash c \xRightarrow{e[j]}_{\mathscr{E}} \textbf{ret}\, c''(f\ j)} \quad [\text{ArrayElement}]$$

$$\frac{\begin{array}{c} \Phi \vdash c_j \xRightarrow{e_j}_{\mathscr{E}} c_{j+1},\ j \in [0..k] \\ \langle s, \mu, l_m, i, o, \_\rangle = c_{k+1} \end{array}}{\Phi \vdash c_0 \xRightarrow{[e_0, e_1, ..., e_k]}_{\mathscr{E}} \langle s, \mu[l_m \leftarrow (k+1, \lambda n.\textbf{val}\, c_n)], l_{m+1}, i, o, l_m\rangle} \quad [\text{Array}]$$

$$\frac{\begin{array}{c} \Phi \vdash c \xRightarrow{e}_{\mathscr{E}} c' \\ l = \textbf{val}\, c' \\ l \in \mathscr{L} \\ (n, f) = (\textbf{mem}\, c')\, l \end{array}}{\Phi \vdash c \xRightarrow{e.\texttt{length}}_{\mathscr{E}} \textbf{return}\, c'\ n} \quad [\text{ArrayLength}]$$

Figure 8: Big-step Operational Semantics for Array Expressions

8.1 Adding arrays on expression level

On expression level, abstractly/concretely:

$\mathscr{E} +=$		$\mathscr{E}[\mathscr{E}]$	$(a[e])$	taking an element
	$\mid$	$[\mathscr{E}^*]$	$([e_1, e_2, .., e_k])$	creating an array
	$\mid$	$\mathscr{E}.\texttt{length}$	$(\texttt{e.length})$	taking the length

The semantics of enriched expressions is modified as follows. First, we add two additional premises to the rule for binary operators:

$$\frac{\begin{array}{cc} \Phi \vdash c \xRightarrow{A}_{\mathscr{E}} c' & \Phi \vdash c' \xRightarrow{B}_{\mathscr{E}} c'' \\ \textbf{val}\, c' \in \mathbb{Z} & \textbf{val}\, c'' \in \mathbb{Z} \end{array}}{\Phi \vdash c \xRightarrow{A \otimes B}_{\mathscr{E}} \textbf{ret}\, c''\ (\textbf{val}\, c' \oplus \textbf{val}\, c'')} \quad [\text{Binop}]$$

These two premises ensure that both operand expressions are evaluated into integer values. Second, we have to add the rules for new kinds of expressions (see Figure 8).

8.2 Adding arrays on statement level

On statement level, we add the single construct:

$$\mathscr{S} += \mathscr{E}[\mathscr{E}]\texttt{:=}\mathscr{E}$$

This construct is interpreted as an assignment to an element of an array. The semantics of this construct is described by the following rule:

$$\frac{\begin{array}{c}\begin{array}{ccc} \Phi \vdash c \xRightarrow{e}_{\mathscr{E}} c' & \Phi\vdash c' \xRightarrow{j}_{\mathscr{E}} c'' & \Phi\vdash c'' \xRightarrow{g}_{\mathscr{E}} \langle s,\mu,l_m,i,o,v\rangle \\ l = \textbf{val}\, c' & i=\textbf{val}\,c'' & \\ l\in\mathscr{L} & i\in\mathbb{N} & \end{array}\\ \\ (n,f)=\mu\, l \\ i<n \\ \texttt{skip}, \Phi \vdash \langle s, \mu[l\leftarrow(n,f[i\leftarrow x])], l_m, i, o, —\rangle \xRightarrow{K} \widetilde{c}\end{array}}{K,\Phi\vdash c \xRightarrow{e[j]:=g} \widetilde{c}}$$

[ArrayAssign]

8.3 Strings

With arrays in our hands, we can easily add strings as arrays of characters. In fact, on the source language the strings can be introduced as a syntactic extension:

1. we add a character constants `'c'` as a shortcut for their integer codes;
2. we add a string literals "`abcd` ..." as a shortcut for arrays `['a', 'b', 'c', 'd', ...]` .

Nothing else has to be done — now we have mutable reference-representable strings.

9 S-expressions and Pattern Matching

S-expressions can be considered as arrays with additionally attached *tags* (of constructors). We denote the set of all constructors as $\mathscr{C}$:

$$\mathscr{C} = \{C_1, C_2, \dots\}$$

Thus, the set of all S-expressions can be described as

$$\mathscr{S}_{exp} = \mathscr{C} \times \mathscr{A}(\mathscr{V})$$

In the concrete syntax constructors are represented by identifiers started with capital letters (A-Z). As being represented as augmented arrays, the values of S-expressions can be manipulated by array primitives (in particular, their elements can be accessed and assigned using square brackets); like arrays they are stored in abstract memory and accessed via locations.

9.1 S-expressions on expression level

At the expression level S-expressions are represented by a single additional syntactic form:

$$\mathscr{E} += \mathscr{C}\mathscr{E}^*$$

In the concrete syntax this form is represented by expressions "`C (e`$_1$`, e`$_1$`, ..., e`$_k$`)`" or just "`C`" if no constructor arguments are specified. The semantics of this form of expression is straightforward: all arguments are sequentially evauated left-to right and their values are packed into an array with appropriate tag attached:

$$\frac{\begin{array}{c}\Phi \vdash c_j \xrightarrow{e_j}_{\mathscr{E}} c_{j+1},\ j \in [0..k] \\ \langle s, \mu, l_m, i, o, \_\rangle = c_{k+1}\end{array}}{\Phi \vdash c_0 \xrightarrow{C(e_0, e_1, ..., e_k)}_{\mathscr{E}} \langle s, \mu[l_m \leftarrow (C, (k+1, \lambda n.\mathbf{val}\, c_n))], l_{m+1}, i, o, l_m\rangle} \quad [\mathrm{S}_{exp}]$$

9.2 Patterns Matching

Pattern-matching is a new control construct which allows to discriminate between different constructors of S-expressions. Pattern-matching is added at the statement level, its abstract syntax is as follows:

$$\mathscr{S} += \mathbf{case}\ \mathscr{E}\,(\mathscr{P} \times \mathscr{S})^*$$

Here $\mathscr{P}$ stands for a new syntactic category — *patterns* (see below). In the concrete syntax:

`case` e **`of`** $p_1 \ \rightarrow s_1$ | ... | $p_k \ \rightarrow s_k$ **`esac`**

where e — expression (*scrutinee*), p_i — patterns, s_i — statements.

The syntactic category of patterns is defined as follows:

$$\mathscr{P} = \_ \mid \mathscr{X} \mid C\mathscr{P}^*$$

In the concrete syntax: "`_`", "`x`", "`C (`$p_1, p_2, ..., p_k$`)`" or just "`C`", where x — variable, C — constructor, p_i — patterns. Informally speaking, each pattern describes a set of S-expressions and a set of bindings of variables to sub sub-expression of each of these expressions. To specify this precisely we introduce the following semantics for patterns: for a pattern p and abstract memory function μ we define

$$[\![p]\!]^\mu : \mathscr{V} \rightarrow \bot \uplus (2^{\mathscr{X}} \times (\mathscr{X} \rightarrow \mathscr{V}))$$

$[\![p]\!]^\mu$ defines a procedure to inspect a value v; as a result either $\bot$ (which can be interpreted as a witness that v does not belong to the set of values defined by p) or a set of variables in the patterns with their bindings is returned. The definition is as follows:

$$\begin{array}{rcl} [\![\_]\!]^{\mu}\, v & = & \langle \varnothing, \text{empty state}\rangle \\ [\![x]\!]^{\mu}\, v & = & \langle \{x\}, [x \leftarrow v]\rangle \\ [\![C(p_1, p_2, \dots, p_k)]\!]^{\mu}\, l & = & \bigoplus_i ([\![p_i]\!]^{\mu}\, v_i) \quad , \quad \mu(l) = C(v_1, v_2, \dots, v_k) \\ [\![p]\!]^{\mu}\, v & = & \bot \quad , \quad \text{otherwise} \end{array}$$

where $\bigoplus$ unions the bindings:

$$\begin{array}{rcl} \bot \oplus \_ & = & \bot \\ \_ \oplus \bot & = & \bot \\ \langle S_1, \sigma_1\rangle \oplus \langle S_2, \sigma_2\rangle & = & \left\langle S_1 \cup S_2, \lambda x. \begin{cases} \sigma_2\, x & , \quad x \in S_2 \\ \sigma_1\, x & , \quad x \in S_1 \setminus S_2 \end{cases} \right\rangle \end{array}$$

As pattern matching may introduce new bindings, we need also to modify the definition of state:

$$\Sigma = (\mathscr{X} \to \mathscr{V}) \times (2^{\mathscr{X}} \times (\mathscr{X} \to \mathscr{V}))^*$$

Now we have one global state and a list of local scopes with sets of variables and their bindings. The redefined state primitives look like the follows. Evaluating in the state:

$$\begin{array}{rcl} \langle \sigma_g, \varepsilon\rangle\; x & = & \sigma_g\, x \\ \langle \sigma_g, \langle S, \sigma_l\rangle :: t\rangle\; x & = & \begin{cases} \sigma_l\, x & , \quad x \in S \\ \langle \sigma_g, t\rangle\; x & , \quad x \notin S \end{cases} \end{array}$$

Updating the state:

$$\langle \sigma_g, s\rangle\, [x \leftarrow z] = \textbf{update}\ \langle \sigma_g, s\rangle\ \varepsilon\, x\, z$$

where **update** is defined as follows:

$$\begin{array}{rcl} \textbf{update}\ \langle \sigma_g, \varepsilon\rangle\ s\, x\, z & = & \langle \sigma_g[x \leftarrow z], s\rangle \\ \textbf{update}\ \langle \sigma_g, \langle S, \sigma_l\rangle :: t\rangle\ s\, x\, z & = & \begin{cases} \langle \sigma_g, s @ \langle S, \sigma_l[x \leftarrow z]\rangle\, t\rangle & , \quad x \in S \\ \textbf{update}\ \langle \sigma_g, t\rangle\ s @ \langle S, \sigma_l\rangle\ x\, z & , \quad x \notin S \end{cases} \end{array}$$

Empty state:

$$\Lambda = \langle \Lambda, \varepsilon\rangle$$

Primitives **enter** /**leave** :

$$\begin{array}{rcl} \textbf{enter}\ \langle \sigma_g, \_\rangle\ S & = & \langle \sigma_g, \langle S, \Lambda\rangle\rangle \\ \textbf{leave}\ \langle \sigma_g, \_\rangle\ \langle \_, t\rangle & = & \langle \sigma_g, t\rangle \end{array}$$

We will also need two additional primitives to push/drop new scopes:

$$\begin{array}{rcl} \textbf{push}\ \langle \sigma_g, t\rangle\ s & = & \langle \sigma_g, s :: t\rangle \\ \textbf{drop}\ \langle \sigma_g, \_ :: t\rangle & = & \langle \sigma_g, t\rangle \end{array}$$

$$\frac{\texttt{skip}, \Phi \vdash \textbf{drop}\ c \xrightarrow{K}_{\mathscr{E}} c'}{K, \Phi \vdash c \xrightarrow{\textbf{leave}}_{\mathscr{E}} c'} \quad [\textsc{Leave}]$$

$$\frac{\Phi \vdash c \xrightarrow{e}_{\mathscr{E}} c' \qquad K, \Phi \vdash c' \xrightarrow{\langle \textbf{val}\ c',\ ps\rangle}_{\mathscr{P}} c''}{K, \Phi \vdash c \xrightarrow{\texttt{case}\ e\ \ ps}_{\mathscr{E}} c''} \quad [\textsc{Case}]$$

$$\frac{[\![p]\!]^{\textbf{mem}\ c}\ v = r\ (\neq \bot) \qquad \textbf{leave}\ \diamond K, \Phi \vdash \textbf{push}\ c\ r \xrightarrow{s}_{\mathscr{P}} c'}{K, \Phi \vdash c \xrightarrow{\langle v,\ \langle p,\ s\rangle\ ::\ t\rangle}_{\mathscr{P}} c'} \quad [\textsc{PatternMatched}]$$

$$\frac{[\![p]\!]^{\textbf{mem}\ c}\ v = \bot \qquad K, \Phi \vdash c \xrightarrow{\langle v,\ t\rangle}_{\mathscr{P}} c'}{K, \Phi \vdash c \xrightarrow{\langle v,\ \langle p,\ s\rangle\ ::\ t\rangle}_{\mathscr{P}} c'} \quad [\textsc{PatternNotMatched}]$$

Figure 9: Continuation-passing Style Semantics for Pattern Matching

We will also use these primitives for the whole configurations, meaning that they apply only to their state parts and leave all other components intact.

To specify big-step semantics for pattern-matching we need a yet another "intrinsic" statement **leave** (remember, the first one was "$\diamond$"). The rules are shown on Fig. 9; note, we need an additional transition relation "$\Rightarrow_{\mathscr{P}}$" to reflect the top-down pattern-matching discipline.