Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 35–47
www.elsevier.com/locate/entcs

Rule-Based Operational Semantics for an Imperative Language

Florent Kirchner[1,2] François-Régis Sinot[1,3]

LIX, École Polytechnique, 91128 Palaiseau, France

Abstract

Operational semantics for programming languages usually come in two flavours: big-step and small-step. Both are defined using *deduction* rules with a congruence rule allowing reduction in certain contexts. For a description based on *rewrite* rules, known approaches consist in deriving an abstract machine, which is very close to implementation. In this paper, we describe the operational semantics of an imperative language in a rule-based style, arguably as abstract as the other kinds of semantics. Moreover, we combine the approach with the store-based semantics, which puts the focus on memory states rather than values, which is more appropriate for imperative languages.

Keywords: Programming Languages, Operational Semantics, Rewriting.

1 Introduction

Structural operational semantics (SOS) [9] has become the standard way of defining the operational meaning (or semantics) of programming languages. Still, SOS allows two different styles of specification: the *big-step style* defining a relation between programs and return values; and the *small-step style* defining a relation between program states. The big-step style is often preferred in the definition of the semantics of a programming language, but the small-step formulation is sometimes more convenient for certain applications (*e.g.*, the proof of language properties [5,2]).

The small-step semantics is intended to give a more fine-grained account of evaluation. Informally, a single small-step derivation is expected to be atomic, in the sense that it should be implementable at a cost independent of the size of the program. Still, conventional presentations of small-step semantics are done as deduction systems, allowing arbitrary high application of deduction rules. In

[1] Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

[2] Email: florent.kirchner@inria.fr

[3] Email: frs@lix.polytechnique.fr

doi:10.1016/j.entcs.2006.10.023

practice, small-step presentations of programming languages are indeed done in this way: the truly atomic steps are described as axioms, and a deduction rule allows reduction in a certain class of contexts. In general, this rule may effectively be applied an arbitrary number of times. Moreover, the class of contexts in which reduction should be allowed may be very tricky to define; this is the main difficulty that arises when one tries to derive the small-step semantics from the big-step one. Thereafter, we call *contextual semantics* this kind of presentation that heavily relies on contexts.

The second author has recently proposed to address these problems,for various strategies of the λ-calculus, by describing small-step semantics as a plain term rewrite system (TRS), instead of a deduction system or a contextual term rewrite system, where the evaluation flow is given an explicit status, as a standard symbol in the syntax of the TRS, called the *evaluation token*. In particular, it is made clear at the syntactic level where reduction will occur, *i.e.* there is exactly one redex at each step, which is marked by the evaluation token. This approach, called the *token-passing semantics* [10,11,12], is adapted to an imperative language in Section 3.

Another defect of conventional descriptions of semantics is the emphasis on values, which may prove cumbersome when the object of study is instead a memory state. This is the case in imperative languages, but also for instance in proof languages where the state is the proof-tree under development. In this situation, the framework is usually adapted to deal with pairs of a program and a memory state: the programs no longer compute a value but a final memory state, so the definition of normal forms needs to be adapted. The use of pairs also creates an artificial asymmetry in the deduction rules, reflecting the fact that subterms of pairs are not pairs themselves.

This problem has been addressed by the first author, resulting in the *store-based semantics* [4], which puts the memory state or *store* at the centre of the semantics, so that reduction is directly defined on memory states. This solves the problem of the normal form of programs, and allows to express the rules for the sequence and the identity in a quite intuitive fashion. Interestingly, this also allows the deletion of the context rule and generates a system of rewrite rather than deductive rules. Some words are spent on the subject in Section 4.

We show that the two approaches can be combined in Section 5, resulting in a truly rule-based semantics of an imperative language, where evaluation flow (reduction order) is made explicit at the syntactic level, and with emphasis on memory states. This could prove particularly useful to express the semantics of languages centred on state manipulation and for which the evaluation order is tricky to define in terms of contexts. It could also be useful to derive in a more or less automatic way small-step semantics and abstract machines from big-step specifications. Moreover, this allows to borrow from well-known rewriting theory results, technologies and tools.

2 Background

2.1 Syntax

We consider a minimal imperative language in the style of IMP [13]. In this note, the focus is on commands (the approaches described in this note can also be applied a given set of expressions).

$$
\begin{array}{rlll}
\mathit{Ref} \ni & x & & \\
\mathit{Int} \ni & n & ::= & 0 \mid 1 \mid 2 \mid \dots \\
\mathit{Val} \ni & v & ::= & \texttt{true} \mid \texttt{false} \mid n \mid \dots \\
\mathit{Expr} \ni & e & ::= & x \mid v \mid \dots \\
\mathit{Com} \ni & c & ::= & \texttt{skip} \\
 & & \mid & x := e \\
 & & \mid & c_1 ; c_2 \\
 & & \mid & \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \\
 & & \mid & \texttt{while } e \texttt{ do } c \\
\mathit{Instr} \ni & i & ::= & e \mid c
\end{array}
$$

x ranges over a finite set of *references*, and e ranges over an arbitrary set of *expressions*, which is assumed to contain at least the *values* v. The *commands* c include the *assignment* ':=' of a reference to an expression, the *sequence* ';' for successive application of two commands, and the usual neutral '`skip`', conditional '`if then else`' and loop '`while do`'. The union of both expressions and commands will be referred to as *instructions*.

2.2 Stores

At the relatively high level of description permitted by the use of semantical frameworks, we will consider that memory states (or stores) are partial mappings from references to values:

$$\sigma \in \mathit{Store} = \mathit{Ref} \rightarrow \mathit{Val}$$

We call *support* of σ the set of references where σ is defined:

$$\mathit{Supp}(\sigma) = \{x \in \mathit{Ref} \mid \exists v \in \mathit{Val},\ \sigma(x) = v\}$$

We also define $\preccurlyeq$ as:

$$\sigma \preccurlyeq \sigma' \iff \forall x \in \mathit{Supp}(\sigma),\ \sigma(x) = \sigma'(x)$$

We call the *empty store* the mapping $\omega \in Store$ whose support is empty and we define the following operation on stores, called *extension*:

$$\sigma\{x \mapsto v\}(y) = \begin{cases} v & \text{if } y = x, \\ \sigma(y) & \text{otherwise.} \end{cases}$$

We remark that any store of finite support may be described as successive extensions of the empty store.

2.3 Big-step semantics

The most intuitive way to describe the behaviour of a program is the big-step (or natural) semantics, which is simply a binary relation between programs and their normal forms. It is given in the form of deduction rules, as follows:

$$\frac{\langle i_1, \sigma_1\rangle \Downarrow \langle v_1, \sigma'_1\rangle \ \cdots \ \langle i_n, \sigma_n\rangle \Downarrow \langle v_n, \sigma'_n\rangle}{\langle i, \sigma\rangle \Downarrow \langle v, \sigma'_n\rangle}$$

where $i_1, \ldots, i_n$ are sub-instructions of i, and each σ_n being either σ or one of the σ'_n.

Figure 1 presents the big-step semantics of IMP.

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma\rangle} \quad \text{(A)}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma'\rangle \quad \langle c_2, \sigma'\rangle \Downarrow \langle \texttt{skip}, \sigma''\rangle}{\langle c_1; c_2, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma''\rangle} \quad \text{(B)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle v, \sigma\rangle}{\langle x := e, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma\{x \mapsto v\}\rangle} \quad \text{(C)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \texttt{true}, \sigma\rangle \quad \langle c_1, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma'\rangle}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma'\rangle} \quad \text{(D)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \texttt{false}, \sigma\rangle \quad \langle c_2, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma'\rangle}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma'\rangle} \quad \text{(E)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \texttt{false}, \sigma\rangle}{\langle \texttt{while } e \texttt{ do } c, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma\rangle} \quad \text{(F)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \texttt{true}, \sigma\rangle \quad \langle c, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma'\rangle \quad \langle \texttt{while } e \texttt{ do } c, \sigma'\rangle \Downarrow \langle \texttt{skip}, \sigma''\rangle}{\langle \texttt{while } e \texttt{ do } c, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma''\rangle} \quad \text{(G)}$$

Fig. 1. Big-Step Semantics of IMP

As an example, consider the evaluation of the program $x := 0; x := 1$ in the initial memory state $\omega\{x \mapsto 7\}$. Then the following derivation yields the result of the program:

$$\dfrac{\dfrac{\overline{\langle 0, \omega\{x \mapsto 7\}\rangle \Downarrow \langle 0, \omega\{x \mapsto 7\}\rangle}}{\langle x := 0, \omega\{x \mapsto 7\}\rangle \Downarrow \langle \mathtt{skip}, \omega\{x \mapsto 0\}\rangle}\ \text{by (C)} \qquad \dfrac{\overline{\langle 1, \omega\{x \mapsto 0\}\rangle \Downarrow \langle 1, \omega\{x \mapsto 0\}\rangle}}{\langle x := 1, \omega\{x \mapsto 0\}\rangle \Downarrow \langle \mathtt{skip}, \omega\{x \mapsto 1\}\rangle}\ \text{by (C)}}{\langle x := 0; x := 1, \omega\{x \mapsto 7\}\rangle \Downarrow \langle \mathtt{skip}, \omega\{x \mapsto 1\}\rangle}\ \text{by (B)}$$

Observe that this formalism reflects a very coarse view of a program's execution: the execution path is not observable in a proposition of the form $t \Downarrow v$, but in the proof tree of this proposition. In some cases, for example when errors are introduced, this does not constitute enough information. Smaller steps within the execution may thus need to be explicitly stated.

2.4 Contextual semantics

The traditional approach to express the semantics of a language in more atomic steps is called *small-step semantics* [7], but we prefer the term *contextual semantics* here, as the formalisms we introduce can arguably also be called small-step semantics. It will even be argued that the steps are somewhat smaller. In this section, we expose the traditional approach, adapted from [7].

Expressions may need to look up variables in the store, and commands may modify the store. A homogeneous notation for reduction is thus of the form: a pair $\langle$*instruction, store*$\rangle$ reduces to another pair $\langle$*instruction, store*$\rangle$. Here witness that $\langle\,,\rangle$ is an instruction evaluator of type $\mathit{Instr} \to \mathit{Store} \to \mathit{Instr} \times \mathit{Store}$; and a series of reductions is written as a succession of pairs $\langle$*instruction, store*$\rangle$:

$$\langle i, \sigma\rangle \to \langle i_1, \sigma_1\rangle \to \langle i_2, \sigma_2\rangle \to \cdots \to \langle i_n, \sigma_n\rangle$$

In particular, if e is an expression, then repeatedly applying the reduction rules will result in its normal form (provided evaluation of expressions does not fail):

$$\langle e, \sigma\rangle \to^* \langle v, \sigma\rangle \not\to$$

where the state is unchanged and v is a value. In this note, the focus is on the evaluation of commands. Hence from now on we assume expressions are correctly evaluated without giving more details; the interested reader may refer to [7].

If c is a command, then, either its reduction does not terminate, or:

$$\langle c, \sigma\rangle \to^* \langle \mathtt{skip}, \sigma'\rangle \not\to$$

The detailed reduction rules for commands are given in Figure 2. The $[\,]$ notation is used to represent the usual notion of hole, and $\Theta[i]$ is the context Θ in which the hole $[\,]$ is replaced by i.

The example used in the previous section here evaluates into:

$$\begin{aligned}
\langle x := 0; x := 1, \omega\{x \mapsto 7\}\rangle &\to \langle \mathtt{skip}; x := 1, \omega\{x \mapsto 0\}\rangle && \text{by (a) + Context.}\\
&\to \langle x := 1, \omega\{x \mapsto 0\}\rangle && \text{by (b)}\\
&\to \langle \mathtt{skip}, \omega\{x \mapsto 1\}\rangle && \text{by (a)}
\end{aligned}$$

$$\frac{\langle i, \sigma\rangle \to \langle i', \sigma'\rangle}{\langle \Theta[i], \sigma\rangle \to \langle \Theta[i'], \sigma'\rangle}$$

$$\begin{array}{rcl} \Theta & ::= & [\,] \\ & | & \Theta; c \\ & | & x := \Theta \\ & | & \texttt{if } \Theta \texttt{ then } c_1 \texttt{ else } c_2 \\ & | & \texttt{while } \Theta \texttt{ do } c \end{array}$$

$$\frac{}{\langle x := n, \sigma\rangle \to \langle \texttt{skip}, \sigma\{x \mapsto n\}\rangle} \quad \text{(a)}$$

$$\frac{}{\langle \texttt{skip}; c, \sigma\rangle \to \langle c, \sigma\rangle} \quad \text{(b)}$$

$$\frac{}{\langle \texttt{if true then } c_1 \texttt{ else } c_2, \sigma\rangle \to \langle c_1, \sigma\rangle} \quad \text{(c)}$$

$$\frac{}{\langle \texttt{if false then } c_1 \texttt{ else } c_2, \sigma\rangle \to \langle c_2, \sigma\rangle} \quad \text{(d)}$$

$$\frac{}{\langle \texttt{while } b \texttt{ do } c, \sigma\rangle \to \langle \texttt{if } b \texttt{ then } (c \,;\, \texttt{while } b \texttt{ do } c) \texttt{ else skip}, \sigma\rangle} \quad \text{(e)}$$

Fig. 2. Contextual Semantics of IMP

A few remarks on this system:

- Because of the asymmetry of the congruence rule with respect to the elements of the pair $\langle$*instruction, store*$\rangle$, and of the role of `skip` in the definition of a normal form, this framework is often considered as being an *ad hoc* extension of that used for purely functional languages.
- This is not a rewrite system, because reduction is not allowed to occur everywhere in the representation of a program. This is rather a deduction system, where outermost rewrite rules are represented as axioms, and there is a deduction rule for contextual reduction.
- Note that only one reduction is possible at a time (*cf.* [7]). The important remark though is that this is far from obvious. This relies on a tight adequacy between the definition of the valid contexts and the restriction of the reduction rules to particular instances (*i.e.* when some subexpressions are already in normal form — values or `skip`).

The token-passing semantics addresses some of these shortcomings.

3 Token-Passing Semantics

In this section, we adapt the token-passing approach to operational semantics [10,11,12] from the λ-calculus to our imperative language. The idea is to get rid of the con-

textual rule by allowing the distinguished marker of reduction (namely $\langle\,,\rangle$ in the previous section) to move up and down in the syntax-tree of the program at the level of reductions instead of deductions. In other words, we put the context rule in the reduction relation, in an explicit manner.

To this end, we replace constructions of the form $\langle i, \sigma\rangle$ by $\Downarrow_\sigma\ i$ and $\Uparrow_\sigma\ i$, depending if we are trying to evaluate something (going down in the context) or if we are returning a value, after some evaluation has been completed (going up in the context). Type-wise, both operators belong to $\mathit{Instr} \to \mathit{Instr} \times \mathit{Store}$.

This gives the set of rewrite rules of Figure 3.

$$\Downarrow_\sigma (x := e) \to x := (\Downarrow_\sigma e) \quad \text{(I)}$$
$$x := (\Uparrow_\sigma n) \to \Uparrow_{\sigma\{x \mapsto n\}} \texttt{skip} \quad \text{(II)}$$
$$\Downarrow_\sigma \texttt{skip} \to \Uparrow_\sigma \texttt{skip} \quad \text{(III)}$$
$$\Downarrow_\sigma (c_1; c_2) \to (\Downarrow_\sigma c_1); c_2 \quad \text{(IV)}$$
$$(\Uparrow_\sigma \texttt{skip}); c \to (\Downarrow_\sigma c) \quad \text{(V)}$$
$$\Downarrow_\sigma (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) \to \texttt{if } (\Downarrow_\sigma e) \texttt{ then } c_1 \texttt{ else } c_2 \quad \text{(VI)}$$
$$\texttt{if } (\Uparrow_\sigma \texttt{true}) \texttt{ then } c_1 \texttt{ else } c_2 \to \Downarrow_\sigma c_1 \quad \text{(VII)}$$
$$\texttt{if } (\Uparrow_\sigma \texttt{false}) \texttt{ then } c_1 \texttt{ else } c_2 \to \Downarrow_\sigma c_2 \quad \text{(VIII)}$$
$$\Downarrow_\sigma (\texttt{while } e \texttt{ do } c) \to \Downarrow_\sigma (\texttt{if } e \texttt{ then } (c\,;\, \texttt{while } e \texttt{ do } c) \texttt{ else skip}) \quad \text{(IX)}$$

Fig. 3. Token-Passing Semantics of IMP

The previous small example illustrates how token-passing rewrites unfold:

$$\begin{aligned}
\Downarrow_{\omega\{x\mapsto 7\}} (x := 0; x := 1) &\to (\Downarrow_{\omega\{x\mapsto 7\}} x := 0); x := 1 && \text{by (IV)}\\
&\to (\Uparrow_{\omega\{x\mapsto 0\}} \texttt{skip}); x := 1 && \text{by (I) + (II)}\\
&\to \Downarrow_{\omega\{x\mapsto 0\}} (x := 1) && \text{by (V)}\\
&\to \Uparrow_{\omega\{x\mapsto 1\}} \texttt{skip} && \text{by (I) + (II)}
\end{aligned}$$

We note the following for this system:

- The system is presented as a plain TRS rather than a deduction system.
- The contextual rule has been put in the reduction relation, in an explicit way. All steps are thus truly atomic, and the number of steps gives a more reasonable account of the cost of evaluation. For instance, a compiled program will most probably not need to perform any context steps.
- As in the conventional approach, there is only one reduction possible at a time. But now, this does not rely on a fine tuning of a class of contexts. This is rather enforced syntactically at the level of terms, in an explicit and extensible way.

In fact, our approach is quite comparable to some previous works [3,8,1], where abstract machines are derived in a more or less systematic way from big or small-step semantics. However, our presentation is somehow more abstract, which facilitates reasoning and still leaves some freedom with respect to implementation.

Although, upwards tokens ($\Uparrow_\sigma$) closely correspond to unfolding successive applications of the contextual rule of contextual semantics, this is obviously something that we would not mind get rid of, as it gives the impression to carry some inefficiency, in the sense that its job is only to move around and not to perform actual computations. This will quite happily be the case of the final system given in Section 5.

The emphasis is still on values: for instance the rule for assignment has to generate a dummy `skip` command. This looks very much like patching a framework that does not quite fit. Fortunately, the issue is addressed by store-based semantics.

4 Store-Based Semantics

In this section, we review the store-based approach to operational semantics [4]. In store-based semantics the memory state is at the centre of the formalism. We consider the $\langle\,,\rangle$ symbol as a store constructor, and will perform reduction directly on stores. In other words, we define two syntactic categories: the atomic stores, which are the equivalent of values but for stores instead of expressions, and the stores, which are either atomic or built with $\langle\,,\rangle$.

$$\mathit{AtStore} \ni \quad \sigma^\star \quad ::= \quad \omega \mid \sigma^\star\{x \mapsto v\}$$

$$\mathit{Store} \ni \quad \sigma \quad ::= \quad \sigma^\star \mid \langle c, \sigma\rangle$$

Here the $\langle\,,\rangle$ evaluator has type $\mathit{Com} \to \mathit{Store} \to \mathit{Store}$. In other words we have the embedding $\mathit{Store} = \mathit{Com} \times \mathit{Store}$. A sequence of reductions is a series of memory states:

$$\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$$

where some of the σ_i are syntactically built using $\langle\,,\rangle$, others are not. The complete evaluation of a program then results in an atomic store:

$$\sigma \to^* \sigma^\star \not\to$$

We can now get rid of the context rule by restricting rewrite rules to atomic stores, which imposes a reduction strategy at the level of the rewrite system, so that only one reduction is possible at a time. Intermediate notations, such as $\underline{:=}$ and $\underline{\texttt{if}}$ ... $\underline{\texttt{then}}$ ... $\underline{\texttt{else}}$, are introduced to mark that evaluation of an expression has already been triggered, and the next step should happen after we get a value. This trick contributes to getting rid of the context rule. Finally, we use the $[\,,]$ notation as a blackbox evaluator for expressions, of type $\mathit{Expr} \to \mathit{Store} \to \mathit{Expr}$. We allow reduction of subexpressions inside expressions as well, although the details are omitted. The store-based semantics of IMP is the set of rewrite rules provided in Figure 4.

$$\langle x := e, \sigma^\star\rangle \to \langle x \underline{:=} [e, \sigma^\star], \sigma^\star\rangle \quad \text{(i)}$$
$$\langle x \underline{:=} n, \sigma^\star\rangle \to \sigma^\star\{x \mapsto n\} \quad \text{(ii)}$$
$$\langle \texttt{skip}, \sigma^\star\rangle \to \sigma^\star \quad \text{(iii)}$$
$$\langle c_1; c_2, \sigma^\star\rangle \to \langle c_2, \langle c_1, \sigma^\star\rangle\rangle \quad \text{(iv)}$$
$$\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma^\star\rangle \to \langle \underline{\texttt{if}}\ [e, \sigma^\star]\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, \sigma^\star\rangle \quad \text{(v)}$$
$$\langle \underline{\texttt{if}}\ \texttt{true}\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, \sigma^\star\rangle \to \langle c_1, \sigma^\star\rangle \quad \text{(vi)}$$
$$\langle \underline{\texttt{if}}\ \texttt{false}\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, \sigma^\star\rangle \to \langle c_2, \sigma^\star\rangle \quad \text{(vii)}$$
$$\langle \texttt{while}\ e\ \texttt{do}\ c, \sigma^\star\rangle \to \langle \texttt{if}\ e\ \texttt{then}\ (c\ ;\ \texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else skip}, \sigma^\star\rangle \quad \text{(viii)}$$

Fig. 4. Store-Based Semantics of IMP

Here again, our example provides an insight on the reduction system:

$$\begin{aligned}
\langle x := 0; x := 1, \omega\{x \mapsto 7\}\rangle &\to \langle x := 1, \langle x := 0, \omega\{x \mapsto 7\}\rangle\rangle && \text{by (iv)}\\
&\to \langle x := 1, \omega\{x \mapsto 0\}\rangle && \text{by (i) + (ii)}\\
&\to \omega\{x \mapsto 1\} && \text{by (i) + (ii)}
\end{aligned}$$

We insist that the resulting system is a plain term rewrite system, as opposed to a deduction system as in Section 2.4. In particular, reduction is not restricted to a given class of contexts. Not quite unlike [6], this system deals with stores as first-class objects, which avoids introducing dummy `skip` commands at some points. This solves some of the problems of Section 2.4. However, as there might be several instances of the $\langle\,,\rangle$ operator, the evaluation order is not enforced syntactically but by restrictions to atomic stores. While this restriction is in the philosophy of the conventional approach, thus perfectly acceptable, it can be improved upon, using the token-passing mindset.

5 Rule-Based Operational Semantics

Both of the previous approaches have their clear merits. It is thus a natural idea to combine them into a semantics where the evaluation order is syntactically explicit and the emphasis is on stores.

5.1 A first shot

One important property of the token-passing approach is to ensure unicity of the token. If one looks more closely at the rules for sequence and `skip` in Figure 4 (store-based semantics), it appears that the first one would duplicate a token, while the second one would erase it. This does not meet the token-passing spirit.

We would instead like every rule to produce exactly one evaluation token. This gives a somewhat eager behaviour to the token: the result of evaluation is an evaluator. Tentative rules are given in Figure 5. The rule for `skip` has to give back a token, so there is very little choice. Then the rule for sequence should give control to the first command, and be such that it behaves transparently with the rule for `skip` when evaluation of the first command is finished. The sequence (semi-colon) is thus simply replaced by an application, so that evaluation will go on in a straightforward way after dealing with its first component. Formally, the syntax of instructions is thus extended with both tokens and with an implicit application.

$$\Downarrow_\sigma \texttt{skip} \to \Downarrow_\sigma$$
$$\Downarrow_\sigma (c_1; c_2) \to (\Downarrow_\sigma c_1)\, c_2$$
$$x := (\Uparrow_\sigma n) \to \Downarrow_{\sigma\{x \mapsto n\}}$$

Fig. 5. Rule-Based Operational Semantics of IMP, first try

Now to evaluate expressions, we cannot just forget the returned value, so we can combine this with the other token-passing-style rules. We do not develop further this issue, as an alternative will be given below.

5.2 Rule-based operational semantics

If we want to go further in the same direction, we would like to get completely rid of the upwards token, as it is already the case for sequence. The idea is then very simple: we know how to do it for commands but not for expressions, so let us transform expressions into commands. More precisely, we transform expressions into assignments: when we reach a return value, this value should be stored in the memory, as a distinguished variable, and evaluation should go on. The transformation is quite clear in Figure 6.

Note that this way of dealing with expressions, *i.e.* as commands, is not as far-fetched as it may seem: the contextual semantics of Section 2.4 has a dual approach, in that it considered commands as a special kind of expressions (functions returning `skip`).

Also observe that the token has the recursive type $\mathit{Eval} = \mathit{Instr} \to \mathit{Eval}$. Following these specifications, a (terminating) series of reductions is of the form:

$$\Downarrow_{\sigma_1} c_1 \to \cdots \to (\Downarrow_{\sigma_i} c_i^1)\, c_i^2 \ \ldots\ c_i^{j_i} \to \cdots \to \Downarrow_{\sigma_{n-1}} c_{n-1} \to \Downarrow_{\sigma_n}$$

$$\Downarrow_\sigma (x := e) \to (\Downarrow^x_\sigma e) \tag{1}$$
$$\Downarrow^x_\sigma n \to \Downarrow_{\sigma\{x\mapsto n\}} \tag{2}$$
$$\Downarrow^x_\sigma \texttt{true} \to \Downarrow_{\sigma\{x\mapsto \texttt{true}\}} \tag{3}$$
$$\Downarrow^x_\sigma \texttt{false} \to \Downarrow_{\sigma\{x\mapsto \texttt{false}\}} \tag{4}$$
$$\Downarrow_\sigma \texttt{skip} \to \Downarrow_\sigma \tag{5}$$
$$\Downarrow_\sigma (c_1; c_2) \to (\Downarrow_\sigma c_1)\ c_2 \tag{6}$$
$$\Downarrow_\sigma (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \to (\Downarrow^x_\sigma e)\ (\underline{\texttt{if}}\ x\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2) \quad x \text{ new} \tag{7}$$
$$\Downarrow_\sigma (\underline{\texttt{if}}\ x\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2) \to \Downarrow_\sigma c_1 \qquad \sigma(x) = \texttt{true} \tag{8}$$
$$\Downarrow_\sigma (\underline{\texttt{if}}\ x\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2) \to \Downarrow_\sigma c_2 \qquad \sigma(x) = \texttt{false} \tag{9}$$
$$\Downarrow_\sigma (\texttt{while}\ e\ \texttt{do}\ c) \to \Downarrow_\sigma (\texttt{if}\ e\ \texttt{then}\ (c\,;\ \texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else skip}) \tag{10}$$

Fig. 6. Rule-Based Operational Semantics of IMP

Running our favourite example in this formalism creates the following derivation:

$$\begin{aligned}
\Downarrow_{\omega\{x\mapsto 7\}} (x := 0; x := 1) &\to (\Downarrow_{\omega\{x\mapsto 7\}} x := 0)\ (x := 1) && \text{by (6)}\\
&\to (\Downarrow^x_{\omega\{x\mapsto 7\}} 0)\ (x := 1) && \text{by (1)}\\
&\to \Downarrow_{\omega\{x\mapsto 0\}} (x := 1) && \text{by (2)}\\
&\to \Downarrow^x_{\omega\{x\mapsto 0\}} 1 && \text{by (1)}\\
&\to \Downarrow_{\omega\{x\mapsto 1\}} && \text{by (2)}
\end{aligned}$$

Remark that we could add a special command `end` with reduction rule $\Downarrow_\sigma \texttt{end} \to \sigma$, to extract the store from the evaluation token at the end of the evaluation. This makes sense from an implementation point of view, if $\Downarrow$ is implemented as a function.

Our formalism is correct and complete in the following sense:

Proposition 5.1 *$\langle c, \sigma\rangle \Downarrow \langle \texttt{skip}, \sigma'\rangle$ if and only if $\Downarrow_\sigma c \to^* \Downarrow_{\sigma''}$ such that $\sigma' \preccurlyeq \sigma''$.*

The proof is omitted but the interested reader may easily check that it is actually much simpler than for contextual semantics [7]. The rule-based operational semantics indeed addresses every aforementioned dissatisfaction of the vintage contextual semantics:

- The system is a plain TRS (no context rule).
- We avoid the need to generate dummy `skip` commands, in particular as part of the normal form of a program.
- Only one reduction is possible at a time. Moreover, the unique redex is explicitly marked in the syntax. Contrasting with the genuine token-passing semantics, the token is here always in the leftmost position.

- As a consequence, it is quite easy to implement. It is also more likely to be automatically derived from the big-step semantics.

Hence in some sense the rule-based semantics may be considered as dual to the contextual semantics, being centered on stores (instead of values), and expressed as rewrite rules (instead of deduction rules).

6 Conclusion

We have presented three original semantical frameworks for a minimalistic imperative language. While being at a higher level than abstract machines, they have many desirable properties compared to conventional approaches. These formalisms rely on a simpler model (term rewriting instead of deduction system), one of them gives an explicit status to the evaluation flow, another one improves the balance between memory state and values and the last one combines these benefits.

Moreover, the approach should not be too difficult to extend to richer languages. For instance, the semantics of procedural proof languages require dealing with a rich memory state, but also with exceptions, data structures, constraints and modules. This is the subject of undergoing work.

The approach has many other potential applications, still to be perceived. Among others, results and tools from rewriting theory may be easier to apply to semantics. It is also a better candidate than regular approaches to bridge the gap between big-step specifications and implementations.

References

[1] Mads Sig Ager. From natural semantics to abstract machines. In Sandro Etalle, editor, *LOPSTR*, volume 3573 of *Lecture Notes in Computer Science*, pages 245–261. Springer, 2004.

[2] Catherine Dubois. Proving ML Type Soundness Within Coq. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 126–144. Springer-Verlag, 2000.

[3] John Hannan and Dale Miller. From operational semantics for abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, December 1992.

[4] Florent Kirchner. Store-based operational semantics. In *Seizièmes Journées Francophones des Langages Applicatifs*. INRIA, 2005.

[5] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 2006. To appear.

[6] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer-Verlag, 2002.

[7] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[8] Andrew M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002.

[9] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[10] François-Régis Sinot. Call-by-name and call-by-value as token-passing interaction nets. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer-Verlag, April 2005.

[11] François-Régis Sinot. Token-passing nets: Call-by-need for free. *Electronic Notes in Theoretical Computer Science*, 2005.

[12] François-Régis Sinot. Call-by-need in token-passing nets. *Mathematical Structures in Computer Science*, 16(4), 2006.

[13] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1993.