# Store-based Operational Semantics

Florent Kirchner*

*Laboratoire d'Informatique de l'École Polytechnique,*
*91128 Palaiseau CEDEX, France*
florent.kirchner@inria.fr

**Résumé**

Imperative programming consists in using side effects on a memory state in order to perform some computation. A language of that kind is usually given an operational semantics *à la Plotkin*, focusing on the values of the program and integrating the concept of memory states and side effects through an explicit extension of the syntax of programs. This article exposes this formalism and intends to show how an alternative, memory state-centered operational semantics can be designed to deal specifically with imperative languages. It also provides an overview of how the depicted frameworks can be adapted to more complex extensions of these languages.

## 1. Introduction

Small-step operational semantics [6] expresses the semantics of a language through the use of reduction rules such as:

$$e \quad \longrightarrow \quad e'$$

where $e$ and $e'$ are *expressions* of the language. The length of the step implemented by each reduction is arbitrarily defined and the reductions are repeated until a normal form, the result or *value* of the program, is reached. This way of expressing the semantics allows for a keen description of the evaluation process; indeed small-step operational semantics were used in many occasions to prove a number of properties for various programming languages [2, 5, 10].

Unlike functional languages that carry out computations via $\beta$-reduction of a program, imperative languages heavily rely on side effects, in other words the alteration of a *memory state* by an *instruction*, to achieve some computation. Reflecting this particularity usually requires a few changes to the aforementioned semantical formalism. For example, the small-step operational semantics of the imperative fragment of the programming language OCaml uses reduction rules over pairs [8]. These are constituted by a program $i$, made of instructions of the language, and a memory state $s$ of the computer:

$$\langle i, s\rangle \quad \longrightarrow \quad \langle i', s'\rangle$$

However a number of problems arise from the introduction of pairs as objects of the reductions. Since imperative programs no longer compute a value, the normal forms of the programs must be adapted. Also, subterms of a pair are not pairs themselves, thus making the expression of a congruence rule quite ticklish.

* Projet LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

Besides, any imperative program can be translated to a functional one. This *functional translation* can serve, for example, to prove properties of imperative programs [3, 4].

But as we shall see, the process of considering a program as a function can also spawn the design of a formalism slightly different from the one presented above, for which the central element is the memory state rather than the program values. Although the resulting formalism has some similarities with the one over pairs, it is generally simpler in several respects: for instance, the definition of normal forms no longer depends on a special value such as `skip`, and a simple congruence rule replaces the more sophisticated context rules. These properties make this formalism well-suited for expressing the semantics of an imperative language, as a straightforward rewrite system. It also proves interesting when studying purely imperative languages, such as proof languages [1], that do not have a notion of returned value; this is the subject of ongoing work.

Section 2 exposes the IMP language, which is the subject of our work, and the notations used. In Section 3 we present the usual small step operational semantics of this language, as laid out in [8], and in Section 4.1 we review the principle of functionalization of imperative languages. Based on this system we expose in Section 4.2 the store-based operational semantics of IMP, a slight deviation of the operational semantics better suited to accommodate the particularities of imperative programming. A second aspect of this work is to study the evolution of these formalisms when some expressions of the language, not only compute a value, but also trigger side effects (Section 5).

## 2. Syntax and notations

### 2.1. Syntax of IMP

The syntax of the imperative language IMP [9] can be divided into three parts defining arithmetic expressions $a$, boolean expressions $b$ and commands (or instructions) $c$:

$$\begin{array}{lll} a & ::= & n \mid X \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\ b & ::= & \mathtt{true} \mid \mathtt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\ c & ::= & \mathtt{skip} \\ & \mid & X := a \\ & \mid & c_1 ; c_2 \\ & \mid & \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \\ & \mid & \mathtt{while}\ b\ \mathtt{do}\ c \end{array}$$

$X$ ranges over a finite set whose elements are called *references*, and $n$ is an integer. IMP is Turing-complete.

### 2.2. Notations

The letters $s$ and $s'$ denote memory states (also called *stores*) of our semantics, which are association lists $X_1 \rightsquigarrow n_1 :: \ldots :: X_m \rightsquigarrow n_m$, and we note $s(X_i) = n_i$. We write $s\{X \rightsquigarrow n\}$ for the memory state where every reference $Y$ is associated to $s(Y)$ except for the reference $X$ which indexes the integer $n$.

The letter $e$ refers to the union of arithmetic and boolean expressions

$$e \quad ::= \quad a \mid b$$

and $v$ is a value

$$v \quad ::= \quad n \mid \mathtt{true} \mid \mathtt{false}$$

# 3. Small-Step Operational Semantics

## 3.1. Formalism

A first definition of the semantics of IMP is derived from the operational semantics of functional languages, where reduction rules read $c \to c'$. When dealing with an imperative language one must also take into account the side effects, which is done by adding stores to terms as objects of the rewrite rules.

A pair $\langle$*instruction, store*$\rangle$ is thus reduced to another pair $\langle$*instruction, store*$\rangle$. With such a formalism, a reduction rule reads:

$$\langle c, s\rangle \quad \to \quad \langle c', s'\rangle$$

The detailed reduction rules are given in Section 3.2. However, we need to define a notion of context in order to deal with in-depth reductions of instructions:

$$\begin{array}{rcl} \Xi & ::= & [\,] \\ & | & \Xi; c \end{array}$$

where $[\,]$ represents the usual notion of hole, and $\Xi[c]$ is the context $\Xi$ in which the hole $[\,]$ is replaced by $c$. The congruence rule follows:

$$\frac{\langle c, s\rangle \to \langle c', s'\rangle}{\langle \Xi[c], s\rangle \to \langle \Xi[c'], s'\rangle}$$

In addition, another notion of pairing $(\!|$*expression, store*$|\!)$ is needed to allow the computation of arithmetic and boolean expressions in a given memory state. In order to allow the reduction of expressions within instructions, we define the congruence rule:

$$\frac{(\!| e, s |\!) \to e'}{\langle \Xi[e], s\rangle \to \langle \Xi[e'], s\rangle}$$

and we enrich the context:

$$\begin{array}{rcl} \Xi & ::= & [\,] \\ & | & \Xi; c \\ & | & X := \Theta \\ & | & \texttt{if } \Theta \texttt{ then } c_1 \texttt{ else } c_2 \\ \text{with } \Theta & ::= & [\,] \\ & | & \Theta + a \mid v + \Theta \mid \Theta - a \mid v - \Theta \mid \Theta \times a \mid v \times \Theta \\ & | & \Theta = a \mid v = \Theta \mid \Theta \leq a \mid v \leq \Theta \mid \Theta \wedge a \mid v \wedge \Theta \mid \Theta \vee a \mid v \vee \Theta \end{array}$$

This defines a left to right, lazy reduction strategy (for example the subterms of a conditional instruction are not reduced *a priori*).

## 3.2. Evaluation rules

**Expressions.** The evaluation rules for expressions are not thoroughly exposed here, but they are quite straightforward. For example, the reduction rule for the sum of integers 4 and 3 or the boolean conjunction of `true` and `false`:

$$\begin{array}{rcl} (\!| 4 + 3, s |\!) & \to & 7 \\ (\!| \texttt{true} \wedge \texttt{false}, s |\!) & \to & \texttt{false} \end{array}$$

Access to memory is as usual:

$$(\!| X, s |\!) \quad \to \quad s(X)$$

**Instructions.** Memory management is done by the rule:

$$\langle X := n, s\rangle \quad \to \quad \langle \texttt{skip}, s\{X \rightsquigarrow n\}\rangle \tag{i}$$

The instructions for sequence and conditional are evaluated by:

$$\langle \texttt{skip}; c, s\rangle \quad \to \quad \langle c, s\rangle \tag{ii}$$

$$\langle \texttt{if true then } c_1 \texttt{ else } c_2, s\rangle \quad \to \quad \langle c_1, s\rangle \tag{iii}$$
$$\langle \texttt{if false then } c_1 \texttt{ else } c_2, s\rangle \quad \to \quad \langle c_2, s\rangle \tag{iv}$$

Finally the loop reuses the definition of the conditional combined with a recursive call:

$$\langle \texttt{while } b \texttt{ do } c, s\rangle \quad \to \quad \langle \texttt{if } b \texttt{ then (} c \texttt{ ; while } b \texttt{ do } c\texttt{) else skip}, s\rangle \tag{v}$$

A series of reductions is thus written as a succession of pairs ⟨*instruction, store*⟩:

$$\langle c, s\rangle \to \langle c_1, s_1\rangle \to \langle c_2, s_2\rangle \to \cdots \to \langle c_n, s_n\rangle$$

In this formalism, the normal form for a well-formed program is a pair $\langle \texttt{skip}, s'\rangle$, with $s'$ being the memory state at the end of execution.

# 4. Store-Based Operational Semantics

## 4.1. Functional Translation

### 4.1.1. Formalism.

A different approach consists in using the concept of program functionalization [4], where an imperative program is treated as a function mapping memory states to memory states.

The resulting functional language can be given some operational semantics, as described in [4], which we do not fully expose here. Yet it is interesting to understand the intuitive signification of the instructions once they have been functionalized.

### 4.1.2. Evaluation Rules.

Instructions are simply treated as functions over stores. Actually:

- the `skip` instruction is simply the identity function $\lambda s.s$ for stores;
- the sequence instruction ; has the semantics of a functional composition operator for two instructions $f$ and $g$: $\circ = \lambda f \lambda g \lambda s.(g\ (f\ s))$ ;
- the loop instruction `while` $b$ `do` $c$ can be seen as the fixpoint of the function: $\lambda f \lambda s.\texttt{if}\ b\ \texttt{then}\ \langle c \circ f, s\rangle\ \texttt{else}\ s$.

Moreover the affectation instruction is similar to a function for store manipulation and the instruction `if` refers to the usual conditional functionality. Note that one could also use monads to express in a functional way the dynamics of side effects. The latter, called monadic translation, is equivalent to the one developed here, the reader can refer to [4] for further reading on this topic.

### 4.1.3. Windup: reasoning on types.

Define $I$ as the type of instructions, $E$ the type of expressions, $V$ the type of values and $S$ the type of stores.

Section 3 presented reduction rules over objects of type $I \times S$. The $\langle\,,\rangle$ symbol was considered as a constructor of cartesian products, and the notation $(\!|\,,|\!)$ was used to evaluate expressions.

On the contrary, in the present section, the interpretation of an imperative program as a function over memory states implies that $I$ is identical to $S \to S$ and that $E$ is similar to $S \to V$. Here the symbol $\langle\,,\rangle$ of type $I \times S \to S$, as well as the notation $(\!|\,,|\!)$ of type $E \times S \to V$, denote the application function. Note how this allows to build programs with nested $\langle\,,\rangle$ and $(\!|\,,|\!)$.

Blending these two concepts yields an alternative solution: a set of reduction rule interpreting the $\langle\,,\rangle$ symbol as an operator for store construction, that is, as a symbol of type $I \to S \to S$. We will see in the next section how this translates with regards to simplifying the semantical framework, but the attentive reader can already foresee the goal that is being aimed for: amalgaming the simplicity of the definition of operational semantics with the fitness of the functional translation's store-centred approach.

## 4.2. Store-based semantics

### 4.2.1. Formalism.

The semantics of IMP is now expressed as a set of relations between memory states. Since we consider the $\langle\,,\rangle$ symbol as a store constructor, the normal form of $\langle c, s\rangle$ is a store, namely the final state resulting from the execution of $c$ in $s$. Thus we write reduction rules in the form:

$$s \quad\to\quad s'$$

And the congruence rule reads:

$$\frac{s \to s'}{\langle c, s\rangle \to \langle c, s'\rangle}$$

where $c$ is an instruction of IMP. Its simplicity triggers the expression of the reduction strategy at the level of the reduction rules. Finally, we reuse the $(\!|\,,|\!)$ notation for expression evaluation, and allow reduction of subexpressions inside instructions.

### 4.2.2. Evaluation rules.

Intermediate notations, such as `aff` and `cond`, are introduced to allow the reduction of subexpressions by the congruence rule.

**Expressions.** The evaluation rules for expressions are the same as in Section 3, only the context rules need to be specified. For example, for the addition, the context rule reads:

$$(\!|a_1 + a_2, s|\!) \quad\to\quad (\!|\texttt{add}\ (\!|a_1, s|\!)\ (\!|a_2, s|\!)\ , s|\!)$$

**Instructions.** Reduction rules for instructions are provided in Figure 1.

Now a sequence of reductions is a series of memory states:

$$s_1 \to s_2 \to \cdots s_n$$

where some of the $s_i$ are syntactically built using $\langle\,,\rangle$, others are not.

$$\langle X := a, s\rangle \rightarrow \langle \texttt{aff}\ X\ (\!|a, s|\!), s\rangle \quad (1)$$
$$\text{and } \langle \texttt{aff}\ X\ n, s\rangle \rightarrow s\{X \rightsquigarrow n\} \quad (2)$$

$$\langle \texttt{skip}, s\rangle \rightarrow s \quad (3)$$

$$\langle c_1; c_2, s\rangle \rightarrow \langle c_2, \langle c_1, s\rangle\rangle \quad (4)$$

$$\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, s\rangle \rightarrow \langle \texttt{cond}\ (\!|b, s|\!)\ c_1\ c_2, s\rangle \quad (5)$$
$$\langle \texttt{cond true}\ c_1\ c_2, s\rangle \rightarrow \langle c_1, s\rangle \quad (6)$$
$$\langle \texttt{cond false}\ c_1\ c_2, s\rangle \rightarrow \langle c_2, s\rangle \quad (7)$$

$$\langle \texttt{while}\ b\ \texttt{do}\ c, s\rangle \rightarrow \langle \texttt{if}\ b\ \texttt{then (c ; while}\ b\ \texttt{do}\ c\texttt{) else skip}, s\rangle \quad (8)$$

Figure 1: Evaluation rules for IMP

#### 4.2.3. Example.

Consider the evaluation of the program $(X := 4; X := 5)$ in the initial memory state $X \rightsquigarrow 8 :: Y \rightsquigarrow 12$, using the formalism of the usual operational semantics (Figure 2) and of our store-based approach (Figure 3). Remark how, in these reduction sequences otherwise very similar to each other, the

| Rules | State |
|---|---|
| | $\langle(X := 4; X := 5), X \rightsquigarrow 8 :: Y \rightsquigarrow 12\rangle$ |
| (i) | $\langle(\texttt{skip}; X := 5), X \rightsquigarrow 4 :: Y \rightsquigarrow 12\rangle$ |
| (ii) | $\langle(X := 5), X \rightsquigarrow 4 :: Y \rightsquigarrow 12\rangle$ |
| (i) | $\langle(\texttt{skip}), \texttt{X} \rightsquigarrow 5 :: \texttt{Y} \rightsquigarrow 12\rangle$ |

Figure 2: Reduction of an imperative program – Operational semantics

| Rules | State |
|---|---|
| | $\langle(X := 4; X := 5), X \rightsquigarrow 8 :: Y \rightsquigarrow 12\rangle$ |
| (4) | $\langle(X := 5), \langle(X := 4), X \rightsquigarrow 8 :: Y \rightsquigarrow 12\rangle\rangle$ |
| (1), (2) | $\langle(X := 5), X \rightsquigarrow 4 :: Y \rightsquigarrow 12\rangle$ |
| (1), (2) | $X \rightsquigarrow 5 :: Y \rightsquigarrow 12$ |

Figure 3: Reduction of an imperative program – Store-based operational semantics

store-based semantics results in a simplification of both the normal form and the intermediate steps. Also note that in the case of Figure 3, another reduction sequence is possible, and yields the same result. If too itching, this alternative derivation can be suppressed simply by specifying that innermost reductions should be given precedence.

#### 4.2.4. Comparison.

In Section 3 of this note we exposed the usual operational semantics of IMP. Although it is widespread, some of its features are somewhat peculiar: the congruence rule, asymmetric with respect to the elements of the pairs $\langle$*instruction, store*$\rangle$; or the role of the `skip` instruction in the definition of a normal form. Therefore this semantical framework is often considered as being rather *ad hoc.*

In the present section we have first seen how the functional translation puts the memory state in the midst of the instructions' semantics, a thing that is not well mirrored by the usual operational semantics. This led us to reformulate the small-step semantics of IMP, using a store-centered formalism in which the congruence rule is direct and the `skip` instruction no longer plays any peculiar role. The possibility of writing terms such as $\langle c_2, \langle c_1, s\rangle\rangle$ in this formalism allows for the fairly natural expression of the concept of sequence, as the result of the application of an instruction to the by-product of the evaluation of an other. Eventually the evaluation rules of this store-based operational semantics are quite close to the ones of the usual operational semantics. One can recognize in (2) the affectation rule of Section 3, as well as the rules (6), (7), and (8) for the conditional and the loop. The rules (1) and (5) are the alternatives to the context rule. Acknowledge the very intuitive rules (3) and (4) which take full advantage of the formalism, also note that these rules describe exactly the semantics of the functionalized instructions: indeed, (4) and (3) are related to the composition and identity over stores.

Finally and as a conclusion for this first part of our work, we can underline the link that this formalism establishes between the functional translation and the operational semantics of imperative languages.

## 5. Extensions of IMP

The previous sections showed how, for an imperative language, operational semantics and functional translation articulate. Now we study the case, occuring in some programming languages, where expressions can trigger side effects.

To illustrate this event, consider an arithmetic expression with side effects : **r**eturn **n**and **u**pdate (`rnu`).

$$a \quad ::= \quad \ldots \mid \mathbf{rnu}\ X\ a$$

This expression is a memory affectation that returns the integer previously affected in that slot. In that way, if a memory state $s$ contains the two references $Y \rightsquigarrow 3$ and $Z \rightsquigarrow 7$, evaluating the expression $2 + \texttt{rnu Y (Z} + 1)$ assigns the integer 8 to the reference $Y$ and returns the sum of 2 and 3, namely 5. In this section we express the semantics of this new expression `rnu` in each of the formalisms of the previous section.

### 5.1. Operational Semantics

#### 5.1.1. Formalism.

Recall that the formalism of operational semantics is about manipulating pairs $\langle$*instruction, store*$\rangle$ and reduction rules such as:

$$\langle c, s\rangle \quad \rightarrow \quad \langle c', s'\rangle$$

In order to deal with expressions such as `rnu`, one must provide more than just a new context $\Xi \quad ::= \quad \ldots \mid \texttt{rnu}\ X\ \Xi$. Indeed, because expressions now trigger side effects, the corresponding rules of Section 3.2 no longer fit: as for instructions, we need to rely on pairs to record changes on stores. Hence reduction rules over pairs read: $\langle c, s\rangle \to \langle c', s'\rangle$ for instructions, and $(\!|e, s|\!) \to (\!|e', s'|\!)$ for expressions. This also requires the modification of the context rules:

$$\frac{\langle c, s\rangle \to \langle c', s'\rangle}{\langle \Xi[c], s\rangle \to \langle \Xi[c'], s'\rangle} \qquad \frac{(\!|e, s|\!) \to (\!|e', s'|\!)}{\langle \Xi[e], s\rangle \to \langle \Xi[e'], s'\rangle}$$

#### 5.1.2. Evaluation Rules.

**Expressions.** The reduction rules for the usual expressions simply return the store unchanged along with the result of the computation:

$$(\!|4+3, s|\!) \quad \to \quad (\!|7, s|\!)$$

And the rule for `rnu` reads:

$$(\!|\mathtt{rnu}\ \ X\ \ n, s|\!) \quad \to \quad (\!|s(X), s\{X \leadsto n\}|\!)$$

**Instructions.** The reduction rules for instructions are identical to those of the corresponding paragraph in Section 3.2.

## 5.2. Functional Translation

Like in Section 4.1, the notations $\langle\, , \rangle$ and $(\!|\, , |\!)$ can be considered as simple applications, instructions as functions over stores, and expressions as functions of stores returning pairs (*value, store*).

## 5.3. Store-based semantics

#### 5.3.1. Formalism.

Here as in Section 5.1, the addition of the `rnu` $X$ $a$ expression in IMP entails the creation of a new operator to evaluate the expressions: $(\!|\, , |\!)$, of type $E \to S \to V \times S$. This operator for cartesian product construction yields proper expression evaluation, which results in the creation of a pair (*value, store*) witnessing both the side effects and the value of the computation. We will use *fst* and *snd* as the usual projection operators over these pairs.

#### 5.3.2. Evaluation Rules.

**Expressions.** The rules handling arithmetic and boolean expressions, when bringing "imperative" expressions in, become more space-consuming than in Section 4.2.2: the reduction of subexpressions has an impact on stores that needs to be mirrored. For example, the addition has for context rule:

$$\begin{gathered}(\!|a_1 + a_2, s|\!) \to \\ (\!|\mathtt{add}\ \ (\mathit{fst}\ (\!|a_1, s|\!))\ (\mathit{fst}\ (\!|a_2, \mathit{snd}\ (\!|a_1, s|\!)|\!)), \mathit{snd}\ (\!|a_2, \mathit{snd}\ (\!|a_1, s|\!)|\!)|\!)\end{gathered}$$

and the usual reduction rule:

$$(\!|\mathtt{add}\ \ 4\ \ 3, s|\!) \quad \to \quad (7, s)$$

The "imperative" expression `rnu` evaluates into:

$$\begin{aligned} (\!|\mathtt{rnu}\ \ X\ \ a, s|\!) \quad &\to \quad (\!|\mathtt{ru}\ X\ (\mathit{fst}\ (\!|a, s|\!)), \mathit{snd}\ (\!|a, s|\!)|\!) \\ \text{and } (\!|\mathtt{ru}\ \ X\ \ n, s|\!) \quad &\to \quad (s(X), s\{X \leadsto n\}) \end{aligned}$$

An evaluation sequence for an expression $e$ and a memory state $s$ would then read as a sequence of pairs:

$$(\!|e, s|\!) \rightarrow (\!|e_1, s_1|\!) \rightarrow \ldots \rightarrow (\!|e_n, s_n|\!) \rightarrow (v, s')$$

**Instructions.** The reduction rules for instructions are close to those of Section 4.2 where, for expression evaluation, calls are made to the first or second component of $(\!|\,,|\!)$. For example, for store management, affectation reads:

$$\begin{array}{rcl} \langle X := a, s\rangle & \rightarrow & \langle \texttt{aff}\ X\ (\mathit{fst}\ (\!|a, s|\!)), \mathit{snd}\ (\!|a, s|\!)\rangle \\ \text{and } \langle \texttt{aff}\ \ X\ \ n, s\rangle & \rightarrow & s\{X \rightsquigarrow n\} \end{array}$$

## 5.4. Expression languages

Eventually, let us explore the following point of view: we can consider all elements of the language IMP as expressions, that return a value and possibly trigger side effects. Hence `skip` or $:=$, in addition to their usual semantics, return an arbitrarily defined value: for example, `unit` or **end**. As a consequence, the notation $\langle\,,\rangle$ previously used to evaluate instructions is useless here, only the symbol for expression evaluation $(\!|\,,|\!)$ remains.

The framework for the operational semantics was originally developed using this approach [8], therefore it comprises the same rules as in Section 5.1, modulo the pair naming convention.

For the store-based semantics, the addition of return values to the instructions entail just a few modifications in the reduction rules. Figure 4 presents the store-based rules for IMPe, *i.e.*, IMP seen as an expression language.

$$\begin{array}{rcl} (\!|X := a, s|\!) & \rightarrow & (\!|\texttt{aff}\ X\ (\mathit{fst}\ (\!|a, s|\!)), \mathit{snd}\ (\!|a, s|\!)|\!) \\ \text{and } (\!|\texttt{aff}\ \ X\ \ n, s|\!) & \rightarrow & s\{X \rightsquigarrow n\} \end{array}$$

$$(\!|\texttt{skip}, s|\!) \rightarrow (\textbf{end}, s)$$

$$(\!|c_1; c_2, s|\!) \rightarrow (\!|c_2, \mathit{snd}\ (\!|c_1, s|\!)|\!)$$

$$\begin{array}{rcl} (\!|\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, s|\!) & \rightarrow & (\!|\texttt{cond}\ (\mathit{fst}\ (\!|b, s|\!))\ c_1\ c_2, \mathit{snd}\ (\!|b, s|\!)|\!) \\ (\!|\texttt{cond true}\ c_1\ \ c_2, s|\!) & \rightarrow & (\!|c_1, s|\!) \\ (\!|\texttt{cond false}\ c_1\ \ c_2, s|\!) & \rightarrow & (\!|c_2, s|\!) \end{array}$$

$$(\!|\texttt{while}\ b\ \texttt{do}\ c, s|\!) \rightarrow (\!|\texttt{if}\ b\ \texttt{then (c ; while}\ b\ \texttt{do}\ c\texttt{) else skip}, s|\!)$$

Figure 4: Evaluation rules for IMPe

Here again, this semantical framework allows for the natural expression of the concept of sequence, while avoiding the shortcomings of the usual semantical framework. Indeed, the points hilighted in Section 4.2.4, such as the appropriateness of the congruence rule over the assymetric context definition or the intuitiveness of the store-based approach, still apply here and contrast with the *ad-hoc* extension of the usual operating semantics to imperative language.

## 6. Conclusion

This work presents a semantical formalism for a minimal imperative language, which intends to be a compromise between the operational and functional approaches. By placing the memory state at the center of the semantics of instructions, it allows for the simple and intuitive formulation of the reduction rules of IMP. To the best of our knowledge, this is the first time it is being used as a standalone formal framework for the operational semantics of an imperative language.

These results can be applied without much foreseen difficulties to larger languages. This formalization should also allow for a more natural semantics for data manipulation languages that make extensive use of side effects: for example, the proof languages of procedural provers.

## 7. Acknowledgments

Many thanks to Gilles Dowek, François Pottier, Didier Rémy and François-Régis Sinot for their patient relectures and insightful comments.

## Bibliographie

[1] David Delahaye. *Conception de langages pour d½rire les preuves et les automatisations dans les outils d'aide ½la preuve.* Thèse de doctorat, Université Paris 6, 2001.

[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, 2000.

[3] Catherine Dubois and Olivier Boite. Proving Type Soundness of a Simply Typed ML-like Language with References. Technical Report EDI-INF-RR-0046, TPHOLs 2001: Supplemental Proceedings, University of Edinburgh, 2001.

[4] Jean-Christophe Filliâtre. Proof of Imperative Programs in Type Theory. In *International Workshop, TYPES '98, Kloster Irsee, Germany*, volume 1657 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1998.

[5] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proc. Marktobderdorf Summer School 2003*. IOS Press, 2003. To appear.

[6] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[7] François Pottier. Typage et Programmation. DEA PSPL Lecture notes, 2002. http://pauillac.inria.fr/~fpottier/mpri/dea-typage.ps.gz.

[8] Fran½is Pottier and Didier R½y. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[9] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, 1993. WIN g2 93:1 P-Ex.

[10] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.