

# ASMs versus Natural Semantics: A Comparison with New Insights

Sabine Glesner

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe  
76128 Karlsruhe, Germany  
<http://www.info.uni-karlsruhe.de/~glesner>

**Abstract.** We compare three specification frameworks for the operational semantics of programming languages, abstract state machines (ASMs) and the two incarnations of natural semantics, big-step and small-step semantics, with respect to two criteria: the range of imperative programming languages to which they are applicable and the way the program is used in the specifications and treated during the thereby defined execution. To reveal the fundamental differences between these three mechanisms, we investigate if there are automatic transformations between them. As a side effect, this leads to new insights concerning the classification of big-step and small-step semantics.

## 1 Introduction

Abstract state machines (ASMs) and natural semantics with its two incarnations big-step and small-step semantics are competing specification frameworks for the operational semantics of programming languages. In the ASM as well as in the natural semantics community, there exists an extensive engineering knowledge of how to use these specification mechanisms appropriately. This raises the widely debated question if there are fundamental differences between them. We compare ASMs and natural semantics according to two main criteria: First, we characterize them regarding the structure of imperative programming languages whose semantics can be defined with them. Secondly, we evaluate them with respect to the way the programs are treated in the specifications and during their thus defined execution. While both criteria are certainly coupled, it turns out that the second criterion really shows where the fundamental differences are.

To accomplish the desired comparison between ASMs and natural semantics, we define, if possible, automatic semantics-preserving transformations from one mechanism into the other. This proceeding is particularly helpful as it separates non-relevant discrepancies in notation from essential differences. Since semantics is defined operationally in these frameworks, each program is regarded as a state transition system. Hence, semantic equivalence means in our context that for each program, the state transitions during its execution are the same. If all specification mechanisms, ASMs as well as big-step and small-step semantics, could be applied for arbitrary programming languages, then we could hope to

find transformations in any desired direction. This is not the case and leads us directly to our classification of imperative programming languages. We distinguish between strictly compositional and non-strictly compositional programming languages. In a strictly compositional programming language, the semantics of each part of the program, which we regard in form of its abstract syntax tree, can be defined solely in terms of the semantics of its direct parts, i.e. subtrees. Big-step semantics defines semantics of programs recursively in terms of the semantics of their direct subtrees. This implies that big-step semantics can only define the semantics of strictly compositional programming languages. This does not hold for small-step semantics and ASMs.

Concerning the second criterion, treatment of programs, there are more similarities between big-step semantics and ASMs while small-step semantics is the outlier. Both big-step semantics and ASMs use the abstract syntax trees of programs in their specifications but do not modify it during program execution. In contrast, small-step semantics explicitly rewrites the abstract syntax trees during execution. In general, a small-step semantics defines a term-rewriting system. Starting with the original program as initial continuation program, during each state transition, the current continuation program is rewritten until the empty program is reached. In each state, the continuation program represents the computation which still needs to be done. In contrast, ASMs represent the remaining computation in a given state as a pointer to the node in the abstract syntax tree which is executed next. In each state transition, this pointer is updated.

We show that each small-step semantics can be transformed automatically into an equivalent ASM semantics and vice versa. We also prove that each big-step semantics can be transformed automatically into an equivalent ASM.

## 2 Semantics of Programming Languages

In general, the semantics of programs is compositional. Given a program in form of an abstract syntax tree, the semantics of each node can be defined directly given its immediate successors. Nevertheless, certain constructs in programming languages exhibit a semantics which is inherently not compositional. E.g., `goto`-statements may leave a program part and go to some other place which cannot be described via the predecessor or successor relation in abstract syntax trees. To define non-compositional semantics, we need *continuations*. A continuation tells us where to proceed with the computation. In compositional program constructs, the continuation specifies simply a child or the parent node. In general, the continuation denotes an arbitrary program node. Continuations can be computed already at compile time. Therefore, attributes are specified defining where to continue the computation. If the control flow branches at a node, then several such continuations are defined, each describing the succeeding computation depending on the branch direction. In this paper, we regard a program as an attributed abstract syntax tree (AST) whose attributes specify the continuations.

### 3 Defining Semantics with ASMs

Abstract state machines (ASMs) [Gur95, AW] have been used extensively in the definition of the semantics of programming languages, e.g. in the definition of C [GH93], of Java [SSB01] and of SDL [EGGP00]. Moreover, ASMs have been used successfully in proving the correctness of compilations, e.g. the correctness of the compilation of Prolog to the WAM [BR94], the correctness of the translation of Occam to transputer code [BD96] and in the Verifix project which deals with the construction of provably correct compilers [ZG97, GZ99]. During these projects, a remarkable engineering knowledge has emerged concerning the way in which specifications should be written to be useful for the purpose of semantics specification and translation verification. In this section, we summarize this particular use of ASMs. Our presentation generalizes the description in [GZ99].

**Remark:** Some ASM semantics (e.g. [SSB01]) modify the AST during program execution. We do not consider this here as it is not the typical case.

Abstract state machines (ASMs) are used to describe the semantics of programming languages operationally as state transition systems based on the abstract syntax trees. Part of the current state is the current task, a pointer to the node in the abstract syntax tree which is currently executed. During program execution, states are transformed into new states, thereby also updating the pointer to the current task. States are regarded as algebras over a given signature. During a state transition, the interpretation  $\mathcal{I}$  of some of the function symbols may change. For example, if a function symbol  $S$  specifies the state of memory, then a variable assignment  $\mathbf{x} := \mathbf{v}$  changes the interpretation  $\mathcal{I}(S)$  of the function symbol  $S$  for argument  $\mathbf{x}$ :  $\mathcal{I}(S(\mathbf{x})) := \mathcal{I}(\mathbf{v})$  holds in the new state. Each  $n$ -ary function symbol is interpreted with an  $n$ -ary mapping. For each state transition, the interpretation of some function values might change. In general, an ASM consists of four components  $(\Sigma \cup \Delta, \mathcal{A}, \text{Init}, \text{Trans})$ : The signature is composed of two disjoint sorted signatures, the signature of the *static functions*  $\Sigma$  and the signature of the *dynamic functions*  $\Delta$ .  $\mathcal{A}$  is the *static algebra*, an order-sorted  $\Sigma$ -algebra interpreting the function symbols in  $\Sigma$ . *Init* is a set of equations over  $\mathcal{A}$  which defines the *initial states* of  $\mathcal{A}$ . Finally, *Trans* is a set of *transition rules* for specifying the state transitions by defining or updating, resp., the interpretations of certain function values of functions in  $\Delta$ . A  $(\Sigma \cup \Delta)$ -algebra is a state of the ASM iff its restriction to  $\Sigma$  is the static algebra  $\mathcal{A}$ . If  $q$  is a state,  $f \in \Delta$  is a function symbol, and  $t_i$  are terms over  $\Sigma \cup \Delta$  with interpretations  $x_i$  in  $q$ , then the update  $f(t_1, \dots, t_n) := t_0$  defines the new interpretation of  $f$  in the succeeding state  $q'$  as

$$q' \models f(x_1, \dots, x_n) = \begin{cases} x_0 & \text{if for all } i, 0 \leq i \leq n, q \models t_i = x_i \\ f_q(x_1, \dots, x_n) & \text{otherwise} \end{cases}$$

A transition rule defines a set of updates which are executed in parallel:

**if** *Cond* **then**  $\text{Update}_1 \dots \text{Update}_n$  **fi**

If  $q \models \text{Cond} = \text{true}$  in state  $q$ , then  $\text{Update}_1 \dots \text{Update}_n$  are executed in  $q$ .

When defining the semantics of programming languages, we use the abstract syntax tree as basis and attach meaning to it, cf. section 2. Thereby, we assume that the abstract syntax tree contains attributes defining all continuations, especially for the non-compositional changes of the control flow. The definition of the ASM models the program counter during program execution, thereby using the continuation attributes which might be split up according to the value of conditions (true case and false case). Here is the example of a transition rule defining the semantics of the while-loop, as stated in [GZ99].  $CT$  ( $CT$  = current task) is the abstract program counter,  $CT.TT$  (true task) is the true-continuation attribute of  $CT$  and  $CT.FT$  (false task) is the false-continuation attribute of  $CT$ .

```

if  $CT \in \textit{While}$  then
  if  $\textit{value}(CT.\textit{cond}) = \textit{true}$  then
     $CT := CT.TT$ 
  else  $CT := CT.FT$  fi fi

```

The semantics of each program node is described by a finite set of transition rules. Typically the condition of such a transition rule specifies the nodes in the abstract syntax tree (**While**-nodes in our example) for which the transition rule is applicable. The transition rules define updates, thereby employing child nodes (in our example  $CT.\textit{cond}$ ) as well as statically computed continuations ( $CT.TT$  and  $CT.FT$  in our above example). In the remainder of this paper, we assume that in an ASM definition which specifies the semantics of a programming language, each transition rule is of the following general form:

```

if  $CT \in X$  then
  if  $\textit{applicability\_conditions}$  then
     $CT := \textit{new\_CT}; \textit{further\_updates}$ 
  else  $CT := \textit{new\_CT}'; \textit{further\_updates}'$  fi fi

```

## 4 Defining Semantics with Natural Semantics

Natural semantics [Plo81, Kah87] is a deductive method to define static and dynamic properties of programs. Thereby, axioms and inference rules specify semantic properties with respect to the abstract syntax. Natural semantics has been used extensively in the definition of programming languages. A prominent example is the complete specification of Standard-ML [MTH90]. Its revision [MTHM97] demonstrates that natural semantics specifications are very stable. Most of the modifications changed the semantics of ML itself rather than correcting errors in the original specification. Further examples for language specifications are the dynamic semantics of Eiffel [Att96], Eiffel// (Eiffel Parallel) [ACEL96], Esterel [Ber90], and in general imperative and object-oriented programming languages [GZ98, Gle99a, Gle99b]. Natural semantics has also been used successfully to prove properties of programs: The investigations in [DE99, Sym99, vON99] prove the static type safety of subsets of the Java programming language whereby specifications based on natural semantics are used.

There are two main variants of natural semantics, big-step semantics and small-step semantics (also called structural operational semantics (SOS)). While big-step semantics can only describe strictly compositional programming languages, small-step semantics is also able to handle non-compositional program constructs. We investigate both variants separately as they differ significantly wrt. our two criteria, i.e. their treatment of the abstract syntax trees and the range of programming languages to which they are applicable. In the following two subsections, we describe both of them, for more details consult e.g. [NN99].

**Remark:** The terminology is not consistent throughout the literature. Sometimes natural semantics refers only to big-step semantics, sometimes it comprises big-step as well as small-step semantics.

#### 4.1 Big-Step Semantics

If the semantics of a programming language is strictly compositional, then we can define the semantics of an abstract syntax tree depending solely on the semantics of its direct subtrees. Consider e.g. the two inference rules for defining the semantics of the while-loop:

$$\frac{\text{Eval}(\text{cond}, \sigma) = \text{false}}{\langle \text{while } \text{cond} \text{ do } S \text{ end}, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\text{Eval}(\text{cond}, \sigma) = \text{true}, \langle S, \sigma \rangle \rightarrow \sigma', \langle \text{while } \text{cond} \text{ do } S \text{ end}, \sigma' \rangle \rightarrow \sigma''}{\langle \text{while } \text{cond} \text{ do } S \text{ end}, \sigma \rangle \rightarrow \sigma''}$$

These two rules express that the body  $S$  of the loop is executed depending on the value of the condition  $\text{cond}$ . If it is executed, then the entire loop is executed recursively again. Typically, this kind of semantic description is only used for terminating computations. In this case, the second rule says that there exists a state transition from  $\sigma$  to  $\sigma''$  if the condition  $\text{cond}$  evaluates to true, if the body  $S$  is executed by a state transition from  $\sigma$  to  $\sigma'$  and if there is a state transition from  $\sigma'$  to  $\sigma''$  describing the recursive execution of the loop.

We can regard a natural semantics as a recursive procedure defined by inference rules. Each inference rule belongs to a production  $X_0 ::= X_1 \cdots X_n$  of the abstract syntax. It has the following general form, whereby  $X_{l_r} \in \{X_1, \dots, X_n\}$ ,  $1 \leq r \leq m$  and  $X_{i_j} \in \{X_0, X_1, \dots, X_n\}$ ,  $1 \leq j \leq k$ ,  $m, k$  natural numbers:

$$\frac{\text{Eval}(X_{l_1}, \sigma_0) = \text{value}_1, \dots, \text{Eval}(X_{l_m}, \sigma_0) = \text{value}_m, \langle X_{i_1}, \sigma_0 \rangle \rightarrow \sigma_1, \langle X_{i_2}, \sigma_1 \rangle \rightarrow \sigma_2, \dots, \langle X_{i_k}, \sigma_{k-1} \rangle \rightarrow \sigma_k}{\langle X_0, \sigma_0 \rangle \rightarrow \sigma_k}$$

The assumptions of an inference rule consist of two main parts, the evaluation conditions  $\text{Eval}(X_{l_r}, \sigma_0)$  and the “procedure calls” on direct successors of  $X_0$ . The evaluation conditions decide about the applicability of the inference rule in a given state  $\sigma_0$ . In the example of the while-loop, they express the value

$\text{Eval}(\text{cond}, \sigma)$  of the condition  $\text{cond}$ . If the evaluation conditions are fulfilled, then the procedures, i.e. inference rules, for  $X_{i_1}, \dots, X_{i_k}$  are called in this order, each with the corresponding initial state  $\sigma_0, \dots, \sigma_{k-1}$  as input value and with the corresponding final state  $\sigma_1, \dots, \sigma_k$  as result. The  $X_{i_j}$ ,  $1 \leq j \leq k$ , are the roots of direct subtrees of  $X_0$ . A particular  $X_{i_j}$  might be called several times with possibly different initial values or might not be called at all. Since we assume a strictly compositional programming language to be described with such inference rules, the semantics of the abstract syntax tree can be concluded solely from the semantics of its direct subtrees (in recursive cases also from its own semantics).

In natural semantics specifications, data structures define the values of the evaluation conditions and the states reached during program execution. These data structures are typically defined inductively by a term algebra over a fixed set of constructor functions. Additional (defined) functions are specified by equations defining recursively the effect of these functions on all constructor terms.

To summarize, big-step semantics describe state transitions for entire programs, i.e. abstract syntax trees. They formalize the execution of a program  $p$  as a transformation from an initial state  $\sigma$  into a final state  $\sigma'$ ,  $\langle p, \sigma \rangle \rightarrow \sigma'$ , specified in the conclusion of the inference rule. This state transition is split up into a sequence  $\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots \rightarrow \sigma_k = \sigma'$  of state transitions such that each individual state transition  $\sigma_{i-1} \rightarrow \sigma_i$ ,  $1 \leq i \leq k$ , is described by exactly one of the assumptions. Since the subsequent program part to be executed next is specified implicitly by this linear ordering of the states,  $\sigma_0 \rightarrow \sigma_1 \dots \rightarrow \sigma_k$ , we do not need to use any continuation attributes explicitly. This suffices because the described programming languages are assumed to be strictly compositional.

## 4.2 Small-Step Semantics

Small-step semantics, also called structural operational semantics (SOS), concentrate on individual steps of program execution and how these single steps are integrated in the overall execution. Assumptions of inference rules formalize smaller steps while their embedding into the larger program context is defined in the conclusion. Individual steps are described in the axioms. Such an individual step is either termination of execution  $\langle p, \sigma \rangle \rightarrow \sigma'$  in the final state  $\sigma'$  or it is a state transition  $\langle p, \sigma \rangle \rightarrow \langle p', \sigma' \rangle$  denoting that the execution of  $p$  in state  $\sigma$  yields a new program  $p'$  to be executed in the succeeding state  $\sigma'$ .  $p'$  is often called *continuation*. In this paper, we call it *continuation program* to distinguish it from the statically computable continuation attributes described in section 2. In most cases,  $p'$  is a direct subtree of  $p$  or composed from direct subtrees of  $p$ . The conclusions of inference rules define the embedding of such program parts into their larger context. In the case of compositional semantics, this context is simply the parent node in the abstract syntax tree. In general, arbitrary continuations are possible, allowing for the description of non-compositional semantics. As typical examples for small-step definitions, consider these inference rules:

$$\begin{array}{c}
\frac{\langle S_1, \sigma \rangle \rightarrow \langle S'_1, \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S'_1; S_2, \sigma' \rangle} \qquad \frac{\text{Eval}(cond)=true}{\langle \text{if } cond \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle} \\
\frac{\langle S_1, \sigma \rangle \rightarrow \sigma'}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \sigma' \rangle} \qquad \frac{\text{Eval}(cond)=false}{\langle \text{if } cond \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle} \\
\langle \text{skip}, \sigma \rangle \rightarrow \sigma \\
\langle \text{while } cond \text{ do } S, \sigma \rangle \rightarrow \langle \text{if } cond \text{ then } (S; \text{while } cond \text{ do } S) \text{ else skip}, \sigma \rangle
\end{array}$$

The first two inference rules in the left column describe how the execution of a sequence of statements  $S_1$  is integrated into a larger context, namely the sequence of statements  $S_1; S_2$ . The first two rules on the right-hand side specify the execution of the if-statement. The first axiom defines the effect of the skip-statement. Finally the last axiom describes the while-loop by reducing its semantics to the semantics of the if-statement. In contrast to a big-step semantics, these rules describe execution in a bottom-up style: execution of smaller program parts is integrated into the execution of larger parts of the program, without paying attention to the overall state transition performed by the entire program. The structure of the inference rules does not need to reflect the structure of the program. For example, the semantics of the while-statement is not defined in terms of its sub-statements but by a new program which has been built from the sub-statements. (Note that the if-statement in the last axiom is not part of the original program but created upon application of this axiom.) In general, also statically computed continuation attributes (cf. section 2) can be used. As an example, consider the semantics of the goto-statement:

$$\langle \text{goto } L; \sigma \rangle \rightarrow \langle L.\text{continuation}, \sigma' \rangle$$

This axiom defines a non-compositional semantics since the control-flow of the program branches to another arbitrary part of the program, denoted by the continuation of  $L$ ,  $L.\text{continuation}$ .

The general form of a small-step inference rule is as follows: Let  $X_0 ::= X_1 \cdots X_n$  be a production of the abstract syntax,  $X_{l_r} \in \{X_1, \dots, X_n\}$ ,  $1 \leq r \leq m$ ,  $m$  a natural number,  $X'_i$  is an arbitrary program built from  $X_i$  or its direct subprograms, i.e. direct subtrees of its abstract syntax tree,  $1 \leq i \leq n$ ,  $X'_0$  is an arbitrary program built from  $X_0$ , from its subprograms  $X_1 \cdots X_n$ , from  $X'_i$  and from the continuations of  $X_0, \dots, X_n$ .

$$\frac{\text{Eval}(X_{l_1}, \sigma) = value_1, \dots, \text{Eval}(X_{l_m}, \sigma) = value_m, \quad \langle X_i, \sigma \rangle \rightarrow \langle X'_i, \sigma' \rangle}{\langle X_0, \sigma \rangle \rightarrow \langle X'_0, \sigma' \rangle}$$

In its evaluation part,  $\text{Eval}(X_{l_1}, \sigma) = value_1, \dots, \text{Eval}(X_{l_m}, \sigma) = value_m$ , the inference rule describes conditions for which the rule is applicable. The state transition  $\langle X_i, \sigma \rangle \rightarrow \langle X'_i, \sigma' \rangle$  defines the execution of  $X_i$  in state  $\sigma$  and gives us a new continuation program  $X'_i$  to be executed in state  $\sigma'$ . The conclusion of the inference rule specifies how this single transition  $\langle X_i, \sigma \rangle \rightarrow \langle X'_i, \sigma' \rangle$  can be integrated into the larger context  $X_0$  whose execution in state  $\sigma$  yields the new continuation program  $X'_0$  to be executed in the new state  $\sigma'$ .

The data structures necessary to define the values of the evaluation conditions and the states reached during program execution are defined in the same way as in big-step semantics definitions, cf. subsection 4.1.

In a small-step semantics, the program to be executed is an explicit part of the state. Each state  $\langle p, \sigma \rangle$  contains a continuation program  $p$ . In the initial state,  $p$  is the original program while in the final state,  $p$  is simply the empty program. The axioms and inference rules of a small-step semantics define how to rewrite this program during each state transition.

## 5 Transformations between ASMs and Natural Semantics

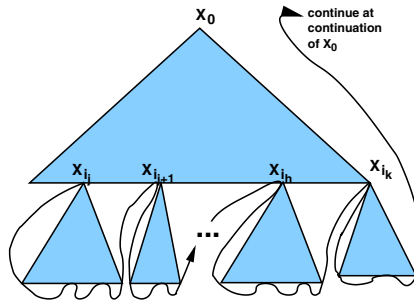
Since big-step semantics can only define strictly compositional semantics, we cannot hope for an automatic transformation from ASMs or small-step semantics to big-step semantics. The reverse direction is possible. We prove that we can transform each big-step semantics into an equivalent ASM. Furthermore, we show that each ASM can be transformed automatically into an equivalent small-step semantics and vice versa. This implies that each big-step semantics can also be transformed into an equivalent small-step semantics.

### 5.1 Data Structures in the Specifications

ASMs as well as big-step and small-step semantics define state transitions by exploiting (more or less strictly) the structure of abstract syntax trees. Thereby data structures are defined to represent the states and values which are computed during program execution. In the ASM case, these data structures are defined by the signatures  $\Sigma \cup \Delta$  of the static and dynamic functions, the set *Init* of equations defining the initial states, and implicitly by the transition rules which specify how to change their interpretation from one state to another. The signatures define a Herbrand universe. The set *Init* maps all terms into the same equivalence class which are equal under these equations. The transition rules define how to modify this Herbrand structure, i.e. the interpretation, from one state to the next. In natural semantics, the data structures are defined also as a term algebra based on *constructor functions*. Additional *defined functions* can be introduced by stating inductively how they operate on constructor terms. These data structures correspond directly to the states of an ASM and vice versa as they can be interpreted also by the same Herbrand structures.

### 5.2 From Big-Step Semantics to ASMs

A big-step semantics defines execution of programs top-down: the state transitions of an entire abstract syntax tree are composed from the state transitions of its direct subtrees and, in recursive definitions, also from its own state transitions. When transforming a big-step semantics into an ASM specification, we need to explicitly define the continuation attributes which are specified only implicitly by the top-down style of the big-step semantics. Therefore we define



**Fig. 1.** Dynamic Continuations

a continuation attribute *cont* for each node in the abstract syntax tree. Since a node  $X_0$  may be called recursively, these continuation attributes must also contain the continuations of all active calls of this node  $X_0$ . We organize the continuations in a stack (with the usual stack operations). We attach a dynamic stack attribute to each node in the abstract syntax tree. Its value during program execution is part of the current state.

It is important to observe that a big-step semantics defines individual state transitions only at the leaves of an abstract syntax tree. For all inner nodes, the inference rules specify how to compose the overall state transition sequence in the conclusion from the state transitions of the assumptions. When defining an equivalent ASM, the idea is to define rules modifying the state for the leaves of the abstract syntax tree. Thereby we use the function *update* taking two arguments  $\sigma$  and  $\sigma'$ . It maps the current ASM state  $\sigma$  to the new state  $\sigma'$  and can be defined easily (cf. remarks in subsection 5.1). Moreover, the rules for the inner nodes of the abstract syntax tree adjust the continuations. The idea is that the most right leaves (wrt. to the ordering of the nodes in the assumptions of the applied inference rules) of each subtree contain the continuation of the root of this subtree, cp. figure 1. We need to update the continuations sequentially from “right to left” wrt. the ordering of the assumptions. Since the ASM rules allow only for the specification of updates to be executed in parallel, we need to introduce several ASM rules per inference rule. The current task  $CT$  is a pair  $(X, n)$  where  $X$  is a pointer to the current node in the AST and  $n$  denotes the  $n$ -th update rule for  $X$  which needs to be executed next.

**Definition 1.** Let **Spec** be a big-step semantics as defined in subsection 4.1, i.e. a set of axioms and inference rules. Then the corresponding ASM  $\text{ASM}_{\text{Spec}}$  is defined by the following transition rules:

- For each axiom  $\langle X, \sigma \rangle \rightarrow \sigma'$ , the corresponding transition rule is defined:

**if**  $CT \in (X, 0)$  **then**  
 $\text{update}(\sigma, \sigma')$ ;  $CT := (\text{cont}(X).\text{top}, 0)$ ;  $\text{cont}(X) := \text{cont}(X).\text{pop}$  **fi**

– For each inference rule of the general form

$$\frac{\text{Eval}(X_{l_1}, \sigma_0) = \text{value}_1, \dots, \text{Eval}(X_{l_m}, \sigma_0) = \text{value}_m, \\ \langle X_{i_1}, \sigma_0 \rangle \rightarrow \sigma_1, \langle X_{i_2}, \sigma_1 \rangle \rightarrow \sigma_2, \dots, \langle X_{i_k}, \sigma_{k-1} \rangle \rightarrow \sigma_k}{\langle X_0, \sigma_0 \rangle \rightarrow \sigma_k}$$

the corresponding transition rules are defined as:

**if**  $CT \in (X_0, 0)$  **then**

**if**  $\text{Eval}(X_{l_1}) = \text{value}_1$  **and**  $\dots$  **and**  $\text{Eval}(X_{l_m}) = \text{value}_m$  **then**

$\text{cont}(X_{i_k}) := \text{cont}(X_{i_k}).\text{push}(\text{cont}(X_0).\text{top}); \text{cont}(X_0) := \text{cont}(X_0).\text{pop};$

$CT := (X_0, k - 1)$  **fi**

**if**  $CT \in (X_0, k - 1)$  **then**

$\text{cont}(X_{i_{k-1}}) := \text{cont}(X_{i_{k-1}}).\text{push}(X_{i_k}); CT := (X_0, k - 2)$  **fi**

...

**if**  $CT \in (X_0, 2)$  **then**

$\text{cont}(X_{i_1}) := \text{cont}(X_{i_1}).\text{push}(X_{i_2}); CT := (X_0.X_{i_1}, 0)$  **fi**

◇

To prove that the semantics of the ASM  $\mathbf{ASM}_{\mathbf{Spec}}$  defines the same semantics as the big-step semantics  $\mathbf{Spec}$ , we need to show that for each program, the state transitions are the same in both specifications  $\mathbf{ASM}_{\mathbf{Spec}}$  and  $\mathbf{Spec}$ .

**Theorem 1.** *Let  $\mathbf{Spec}$  be a big-step semantics and  $\mathbf{ASM}_{\mathbf{Spec}}$  the corresponding ASM. The state transitions are the same for each program in both specifications.*

*Proof.* State transitions happen only at the leaves of the AST. The continuation of a node  $X$  is a reference to the node where the computation is to be continued after the computation of  $X$  is finished. The computation of a node and subtree  $X$  is finished when all its leaves are computed. Therefore the leaf processed at last has a continuation pointing to the continuation of  $X$ . Since a node might be called recursively, the different calls and their continuations are superimposed recursively in the abstract syntax tree. Since the continuations are organized in a stack, they represent the nested recursive structure properly. To prove that the state transitions of  $\mathbf{Spec}$  and  $\mathbf{ASM}_{\mathbf{Spec}}$  are the same, we distinguish between terminating and non-terminating programs. For the terminating case, we do induction on the height of the abstract syntax tree  $X$  and its run-time expansion.

**Base Case:**  $X$  is a leaf described by axiom  $\langle X, \sigma \rangle \rightarrow \sigma'$ . Clearly the state transition  $\text{update}(\sigma, \sigma')$  gives us the same new state  $\sigma'$ . The deletion of the top continuation reference removes the current recursive frame.

**Induction Step:** For the computation of  $X_0, X_{i_1}, \dots, X_{i_k}$  need to be computed. We can assume that for  $X_{i_1}, \dots, X_{i_k}$ , the state transitions are the same in the big-step semantics  $\mathbf{Spec}$  and in  $\mathbf{ASM}_{\mathbf{Spec}}$  (induction hypotheses). It remains to show that the continuations are correct. Due to the updates  $\text{cont}(X_{i_j}) := \text{cont}(X_{i_j}).\text{push}(X_{i_{j+1}})$  for  $1 \leq j \leq k - 1$ ,  $X_{i_{j+1}}$  is computed directly after  $X_{i_j}$ ,  $1 \leq j \leq k - 1$ . The adjustment of the continuations from “right to left” makes sure that the stacking of the continuations is correct for the case that there exists  $j \leq h, j, h \in \{1, \dots, k\}$  such that  $X_{i_j} = X_{i_h}$ . When processing the subtree

marked with  $X_{i_j}$  (which equals  $X_{i_h}$ ), first the continuation of the  $i_j$ -th subtree needs to be taken, then the continuation of the  $i_h$ -th subtree. Finally, after  $X_{i_k}$ ,  $\text{cont}(X_0)$  is executed so that computation either stops if  $X_0$  is the root of the program or continues at the continuation of  $X_0$ . The deletion of the continuation of  $X_0$ ,  $\text{cont}(X_0).\text{pop}$ , removes the current recursive frame at  $X_0$ . Since the continuation of  $X_0$  is stored in the right most leaf of the subtree located at  $X_0$ , this continuation  $\text{cont}(X_0)$  is not needed any more. Whenever this right-most leaf is reached, the execution will continue directly at  $\text{cont}(X_0)$ .

**Induction Step for Non-terminating Programs:** The above proof is only valid if the programs terminate. Only then the expansions of  $X_{i_1}, \dots, X_{i_k}$  are truly smaller than the expansion of  $X_0$ . If the program does not terminate, then there is one  $l$ ,  $1 \leq l \leq k$ , such that  $X_{i_1}, \dots, X_{i_{l-1}}$  are truly smaller than  $X_0$  and such that  $X_{i_l}$  is the first subtree with infinite height. The computation will get stuck in  $X_{i_l}$ . To prove that both specifications **Spec** and **ASM<sub>Spec</sub>** show the same state transition behavior, we need to show that the execution at the root of  $X_{i_l}$  starts with the same state. This state is  $\sigma_{l-1}$ . The state at the root of  $X_{i_l}$  is the same for both specifications. Hence, we can conclude that both specifications enforce the same state transition behavior. If this were not the case, then there would be a smallest number of state transitions after which they are different. In the state before they would be the same. But then they must also be the same in the proceeding state.  $\square$

### 5.3 From Small-Step Semantics to ASMs

Small-step semantics define the execution of programs by recursively defining how to transform an initial state as well as a given program itself stepwise into a new state and a new program. This means that a small-step semantics does not define execution by a (eventually recursive) walk through the abstract syntax tree as it is the case in a big-step semantics. Rather the program is treated as a term which is rewritten during execution until it is reduced to the empty tree. This term to be reduced is also called *continuation*.

The axioms and inference rules in a specification define a recursive rewriting procedure. An axiom  $\langle X, \sigma \rangle \rightarrow \sigma'$  or  $\langle X, \sigma \rangle \rightarrow \langle X', \sigma' \rangle$ , resp., states that the current continuation  $X$  is to be replaced by the empty tree or the new program  $X'$ , resp. An inference rule of the general form, cf. subsection 4.2, calls the rewriting procedure recursively on one of the direct subtrees  $X_i$  of the current program. After its completion, the rewriting procedure modifies its continuation and state by possibly integrating the results of the recursive call. The detailed recursive algorithm is stated in figure 2 in a pseudo-Pascal notation.

In general, this is not a pure term rewriting procedure since nodes may have static continuation attributes, cf. the goto-definition in subsection 4.2, which might point to arbitrary nodes in the original program tree. So whenever we talk about a subtree of the original program, we do not only mean the subtree itself but the transitive closure of all subtrees to which static continuations point.

```

proc eval_AST(Cur_AST, state) : (New_AST, new_state);
if Cur_AST = nil then
  (New_AST, new_state) := (nil, state);
fi;
if Cur_AST ∈ X and  $\langle X, \sigma \rangle \rightarrow \sigma' \in \text{Spec}$  then
  new_state := update( $\sigma, \sigma'$ )[ $\sigma / \text{state}$ ];
  New_AST := nil;
fi;
if Cur_AST ∈ X and  $\langle X, \sigma \rangle \rightarrow \langle X', \sigma' \rangle \in \text{Spec}$  then
  new_state := update( $\sigma, \sigma'$ )[ $\sigma / \text{state}$ ];
  New_AST :=  $X'[X / \text{Cur\_AST}]$ ;
fi;
if Cur_AST ∈ X0
   $\text{Eval}(X_{i_1}, \sigma) = \text{value}_1, \dots, \text{Eval}(X_{i_m}, \sigma) = \text{value}_m,$ 
and  $\frac{\langle X_i, \sigma \rangle \rightarrow \langle X'_i, \sigma' \rangle}{\langle X_0, \sigma \rangle \rightarrow \langle X'_0, \sigma' \rangle} \in \text{Spec}$ 
and  $\exists$  direct subtree  $X_i(\text{Cur\_AST})$  of Cur_AST such that  $X_i(\text{Cur\_AST}) \in X_i$ 
and  $\text{Eval}(X_{i_1}(\text{Cur\_AST}), \text{state}) = \text{value}_{i_1}$  and  $\dots$ 
and  $\text{Eval}(X_{i_m}(\text{Cur\_AST}), \text{state}) = \text{value}_{i_m}$  then
  (Cur_AST', state') := eval_AST( $X_i(\text{Cur\_AST}), \text{state}$ );
  (New_AST, new_state) := eval_AST( $X'_0[X_0 / \text{Cur\_AST}, X'_i / \text{Cur\_AST}'], \text{state}'$ );
fi;
return (New_AST, new_state);
end_proc

```

(*AST* stands for *abstract syntax tree*.)

**Fig. 2.** Meaning of a Small-Step Semantics

The algorithm in figure 2 can easily be transformed into an ASM definition. Therefore the recursion is eliminated by transforming the recursive procedure into a while-loop which runs until the program is reduced to the empty tree. In the usual way, the nested recursive calls at run-time are modelled by a stack whose entries are tuples of the current continuation (i.e. program) and the current state, carrying the state of computation of the individual recursive calls. This is a trivial standard proceeding to eliminate recursion. The resulting while-loop can easily be restated as an ASM: The while-loop still contains the four if-statements as the original recursive procedure. It is straightforward to transform these if-statements into four corresponding ASM transition rules. The content of the stack of the while-loop during execution becomes the state of the ASM.

This resulting ASM is different from the ASMs typically defined when specifying the semantics of programming languages, cf. section 3. It does not define how to traverse the abstract syntax tree, e.g. by using a current task *CT*. Even though in many practical cases, given a small-step semantics, a human might easily be able to define walks through abstract syntax trees and a corresponding ASM, in general we cannot hope to find such an automatic transformation. This

is because a small-step semantics has many degrees of freedom in transforming a given program. It might duplicate subtrees, move subtrees from one part of the program to others by using the static continuation attributes, etc. This demonstrates the difference between small-step semantics on one hand and big-step semantics and ASM semantics on the other hand: While in big-step and (most) ASM semantics, the abstract syntax tree is a constant during program execution, it is modified during the execution defined by a small-step semantics.

#### 5.4 From ASMs to Small-Step Semantics

An ASM specification defines the semantics of a programming language operationally based on the abstract syntax trees of the programs. The state of the ASM contains a reference  $CT$  to some node in the abstract syntax tree, pointing to the current task to be executed. ASM semantics specifications are able to describe not strictly compositional semantics. Therefore we cannot expect to find a transformation from ASM semantics specifications to big-step semantics because big-step semantics can only define strictly compositional semantics. But we can define a transformation from ASM semantics to small-step semantics.

The idea is to take the abstract syntax tree as (constant) dynamic continuation. The current task  $CT$  becomes part of the state: If  $\sigma$  is a state of the ASM and  $CT$  the current task during some point of program execution, then  $(CT, \sigma)$  is the state at the same point of program execution wrt. to the corresponding small-step semantics. Formally, for each transition rule in an ASM semantics,

```

if  $CT \in X$  then
  if applicability_conditions then
     $CT := new\_CT; further\_updates$ 
  else  $CT := new\_CT'; further\_updates'$  fi
  fi
    
```

the corresponding inference rules are defined as follows:

$$\frac{\frac{\text{applicability\_conditions}(AST, (\sigma, CT))}{\langle AST, (\sigma, CT) \rangle \rightarrow \langle AST, (further\_updates(\sigma), new\_CT) \rangle}}{\frac{\neg \text{applicability\_conditions}(AST, (\sigma, CT))}{\langle AST, (\sigma, CT) \rangle \rightarrow \langle AST, (further\_updates'(\sigma), new\_CT') \rangle}}$$

Again, as in subsection 5.2 and 5.3, we assume that the data structures of the ASM specification can be transformed easily into corresponding data structures of a small-step semantics. To prove that the defined transformation is correct, we need to show that for each program, the state transitions are the same wrt. to both specifications. Since the inference rules are not recursive, i.e., there are no state transitions specified in their assumptions, each execution will directly undertake the state transition of the conclusion of some inference rule whose assumptions are valid. This is the same as saying that some transition rule whose conditions (which are equivalent to the assumptions of the matching inference rule) are fulfilled will be executed. Therefore it follows immediately that the original ASM specification and the defined small-step semantics are equivalent.

## 6 Related Work

The results of our investigations are in contrast to the common understanding (cf. [NN99] or any other textbook or lecture notes of your choice) that big-step semantics can only describe terminating programs while small-step semantics is also suited for the description of non-terminating computations. This view is not adequate as our investigations show. Rather, it is the common interpretation of big-step semantics which fits only for terminating computations. In the traditional interpretation of big-step semantics, the assumptions must be true (terminating) in order to infer the conclusion. In our view, we ask if a (terminating or non-terminating) state transition sequence is consistent with the rules of the big-step semantics, allowing us to deal with non-terminating computations as well. This seems to be the more appropriate view anyway. Otherwise one could not determine whether a program has a semantics at all because this would be equivalent to solve the halting problem.

There are no other works comparing ASMs and big-step and small-step semantics wrt. the criteria “applicability to imperative programming languages” and “treatment of the AST”. In particular, no transformations between the three mechanisms have been proposed. Only [ACK<sup>+</sup>00] proposes a mechanism to generate action notation environments from montages descriptions.

## 7 Conclusions

The three specification frameworks ASMs and small-step and big-step semantics vary significantly wrt. our two criteria “applicability to imperative programming languages” and “treatment of the abstract syntax tree”. While big-step semantics can only define strictly compositional programming languages, ASMs and small-step semantics can also specify non-strictly compositional program constructs. Furthermore, big-step and most ASM semantics do not modify the abstract syntax tree during computation, in contrast to small-step semantics which explicitly defines a term-rewriting system that rewrites the program during execution until the empty program is reached. These differences are reflected in the transformations between them. We have shown that each ASM semantics can be transformed into an equivalent small-step semantics and vice versa. Furthermore we have proved that each big-step semantics can be transformed into an equivalent ASM semantics while the reverse direction cannot be expected.

From a theoretical point of view, these transformations are interesting as they reveal the unexpressed interpretations of the specification frameworks. ASMs and small-step semantics follow the idea that a program defines a state transition system whose execution can be observed. In contrast, the usual interpretation of big-step semantics defines how to construct finite state transition sequences. Our transformations indicate that for each specification mechanism, both interpretations are possible. The traditional classification – big-step only for terminating programs, small-step also for non-terminating programs – is not a classification of the specification frameworks but rather of their usual interpretations.

These transformations are also interesting from a practical point of view. In the ASM as well as in the natural semantics community, a remarkable engineering knowledge has emerged concerning the way in which specifications should be written to be useful for the purpose of semantics specification and translation verification. Having the transformations between these mechanisms in mind, we can transfer this engineering knowledge from one community to another.

In practice, most small-step semantics do not have the intention to define a term-rewriting system but rather incorporate the idea of defining a current task as in the ASM semantics. Therefore it would be interesting to define a simplified small-step semantics which allows for recursive semantic definitions on the tree structure of the program but does not permit to rewrite it. This seems to be sufficient for the usual applications. Moreover, it should be investigated if the three specification mechanisms ASMs and small-step and big-step semantics deal differently with multi-threaded and parallel programming languages. Finally, it should be investigated if they behave differently when used in automated theorem proving. This is an important criterion in the formal reasoning on the semantics of programming languages and the correctness of translations.

## Acknowledgment

The author thanks A. Dold, W. Goerigk, G. Goos, H. Langmaack, V. Vialard, W. Zimmermann for many discussions on this research topic. Thanks also to the anonymous referees and E. Börger for valuable comments.

## References

- [ACEL96] Isabelle Attali, Denis Caromel, Sidi O. Ehmety, and Sylvain Lippi. Semantic-Based Visualization for Parallel Object-Oriented Programming. In *Proc. OOPSLA '96 (Object-Oriented Programming: Systems, Languages, and Applications)*, 1996. ACM Press, Sigplan Notices, Vol. 31, No. 10. 296
- [ACK<sup>+</sup>00] M. Anlauff, S. Chakraborty, P. W. Kutter, A. Pierantonio, L. Thiele. Generating an Action Notation Environment from Montages Descriptions. *STTT*, 2000. 306
- [Att96] Isabelle Attali. A Natural Semantics for Eiffel Dynamic Binding. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 18(5), 1996. 296
- [AW] ASM-Website. <http://www.eecs.umich.edu/gasm/>. 295
- [BD96] Egon Börger and Igor Durdanovic. Correctness of compiling Occam to Transputer Code. *Computer Journal*, 39(1):52–92, 1996. 295
- [Ber90] Yves Bertot. Implementation of an Interpreter for a Parallel Language in Centaur. In *Proceedings of the 3rd European Symposium on Programming*, Copenhagen, Denmark, May 1990. Springer Verlag, LNCS 432. 296
- [BR94] Egon Börger and Dean Rosenzweig. The WAM - definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications*. North-Holland Series in Computer Science and Art. Int., 1994. 295

- [DE99] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, page 41 ff. Springer Verlag, LNCS 1523, 1999. 296
- [EGGP00] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of Design Languages: A compilation approach using Abstract State Machines. In *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines*, 2000. Local Proceedings, TIK-report 87, Swiss Federal Institute of Technology (ETH) Zurich. 295
- [GH93] Yuri Gurevich and James K. Huggins. The semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, pages 274–308. Springer Verlag, LNCS 702, 1993. 295
- [Gle99a] Sabine Glesner. Natural Semantics for Imperative and Object-Oriented Programming Languages. In *Informatik '99 - Informatik überwindet Grenzen, Proceedings der 29. Jahrestagung der Gesellschaft für Informatik*, 1999. GI-Gesellschaft für Informatik e.V., Springer Verlag. 296
- [Gle99b] Sabine Glesner. *Natürliche Semantik für imperative und objektorientierte Programmiersprachen*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, February 1999. Shaker Verlag, Aachen, ISBN: 3-8265-6388-3. 296
- [Gur95] Y. Gurevich. Evolving Algebras: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995. 295
- [GZ98] Sabine Glesner and Wolf Zimmermann. Using Many-Sorted Natural Semantics to Specify and Generate Semantic Analysis. In *Proc. Systems Implementation Conference (SI2000)*, 1998. IFIP WG 2.4, Chapman & Hall. 296
- [GZ99] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In *Correct System Design*. Springer-Verlag, LNCS 1710, 1999. 295, 296
- [Kah87] Gilles Kahn. Natural Semantics. In *Proc. 4th Annual Symp. on Theoretical Aspects of Computer Science (STACS'87)*, 1987. Springer, LNCS 247. 296
- [MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990. 296
- [MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 296
- [NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, A Formal Introduction*. Published in 1992 by John Wiley & Sons, revised edition available at <http://www.daimi.au.dk/~hrn>, 1999. 297, 306
- [Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Comp. Sc. Dep., Aarhus University, Denmark, 1981. 296
- [SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer Verlag, 2001. 295
- [Sym99] Don Syme. Proving Java Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Springer Verlag, LNCS 1523, 1999. 296
- [vON99] David von Oheimb and Tobias Nipkov. Machine-Checking the Java Specification: Proving Type-Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, page 119 ff. Springer Verlag, LNCS 1523, 1999. 296

- [ZG97] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Backends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997. [295](#)