Simulation of SOS Definitions with Term Rewriting Systems

Karl-Heinz Buth*

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preusserstr. 1–9, D–24105 Kiel, Germany
e-mail: khb@informatik.uni-kiel.d400.de

Abstract. Reasoning about programming language semantics with an automated proof tool requires that the semantics definition be stated in a formalism that is suitable for the tool. This paper presents a method to transform a structured operational semantics (SOS) definition $\mathcal{S}$, given by a special form of deduction system, into a term rewriting system $\mathcal{R}$. This system $\mathcal{R}$ simulates $\mathcal{S}$ very closely in that sense that the sets of possible configuration sequences are essentially the same. Since only standard unconditional rewrite rules are used, every theorem prover based on rewriting can be employed to implement this kind of semantics definitions, and so to reason about them.

1 Introduction

A common way to describe the semantics of a programming language is to give an operational definition. This means that an abstract machine is introduced, and that the elements of the language are explained in terms of the machine instructions. If we want to reason about such a definition with the help of a proof support system, we have to express the explanations within the tool's formalism.

In this paper, we demonstrate a way how this can be done for structured operational semantics (SOS) definitions in the sense of Plotkin (cf. [21]), and proof tools based on term rewriting. An SOS definition is given by means of a transition system, where the transition relation is presented in form of a set of deduction rules. These rules cannot be used directly as rewrite rules since the conditions on the variables that occur in rewrite rules are more restrictive than those required for deduction rules. Our way to solve this problem is to use a limited subset of λ-calculus to model the rules. This subset can be expressed completely by rewrite rules; thus no new formalism is needed. We can prove that our method leads to a very close simulation of the original transition system. Since only simple rewrite rules are used, the method can be implemented with every proof tool that supports term rewriting.

* Partially supported by Esprit BRA projects 3104 "ProCoS" and 7071 "ProCoS II" and Deutsche Forschungsgemeinschaft, grants La 426/12-1 and La 426/12-2.

An application of this approach is presented in [7], where the Larch Prover [11] is used as a proof assistant in an equivalence proof for different semantics definitions. The motivation for the work emerged from verification work in the ESPRIT BRA project ProCoS (*Provably Correct Systems*, cf. [4]). Several of the ProCoS project languages have been defined operationally, and therefore there is a need to reason about SOS definitions.

We start in section 2 with a short account of term rewriting, followed in section 3 by an introduction to SOS definitions. Section 4 presents the transformation of deduction into rewrite rules. A summary of the simulation properties of the derived rewriting system is given in section 5. The proofs for the results can be found in the full version of this paper ([6]). Section 6 describes in which way the simulating rewrite system can be applied.

2 Term rewriting controlled by contexts

The approach to term rewriting we use in our simulation is a typed one, based on a many-sorted logic (for details, cf. [9]). We only employ unconditional rewrite rules; so any standard tool can be used for the implementation of the ideas.

A **term rewriting system** (**TRS** for short) is a finite set of rules $\lambda \rightarrow \rho$, where λ and ρ are terms of the same type, and ρ does not contain *extra variables* w. r. t. λ, i. e. all variables in ρ are also in λ. An f**-term** is a term whose outermost operator is f. An f**-rule** is a rule $\lambda \rightarrow \rho$ where the left-hand side λ is an f-term. For a signature Σ and a set of variables V, $T(\Sigma, V)$ denotes the set of all terms built over Σ and V. For $n \in \mathbb{N}_0$, we define $[n] =_{df} \{1, \ldots, n\}$.

Our simulation relies on terms of the form $T \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, since this is a convenient way to give names to intermediate results: x is a name for the result of "evaluating" e_1. Now T is just another notation for $(\lambda x.e_2)e_1$, and so we have started using concepts of λ-calculus. But we need not merge rewriting and β-reduction as it is done in e. g. [10, 18]; since we only have λ-abstractions that are directly applied to non-functional arguments, we can use standard rewrite systems to evaluate β-reductions. This method, presented e. g. in [1], is based on the replacement of bound variables by de Bruijn indices ([5]). For details, cf. [6]; in the following, we will only use the **let** form to write such terms.

In order to obtain a correct simulation, we need to control the rewriting process in two ways (for the examples, assume that f is a unary operator):

- We want to be able to perform just one top-level rewriting of an f-term t without restricting the rewriting of subterms. This cannot be done by ordinary methods like restricting the length of rewriting sequences to 1.
- We want to allow a rule of the form
  $f(t_1) \rightarrow \mathsf{if}\ b\ \mathsf{then}\ f(t_2)\ \mathsf{else}\ f(t_1)$
  This rule is obviously non-terminating (in the usual sense), but we want to be able to apply it without successive selections of the else part. This means that when an instance of b is known to reduce to **false**, the corresponding instance of this rule shall be "de-activated".

For the first wish, we need to have some kind of counter, and for the second a way of "switching a rule off" (and on again, of course). This is the motivation for the following definition.

Definition 1. Let $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n$ be all the f-rules in our rewrite system. Then a **context for f** is an element of $\{0, 1, *\} \times \{\mathsf{on}, \mathsf{off}\}^n$. For $a \in \{0, 1, *\}$ and $s_1, \ldots, s_n \in \{\mathsf{on}, \mathsf{off}\}$, the context $\langle a, s_1, \ldots, s_n \rangle$ is called an **a-context**.

Contexts can be used for our purposes since they contain a counter component ($0, 1$ or $*$) and a switch for each f-rule. Instead of rewriting terms $f(t)$, we now rewrite terms $f(t) @ \langle a, s_1, \ldots, s_n \rangle$ where @ is the special context application operator, $a \in \{0, 1, *\}$ and $s_1, \ldots, s_n \in \{\mathsf{on}, \mathsf{off}\}$. The intended interpretation for a is:

- no more top-level rewriting steps, if $a = 0$,
- at most one top-level rewriting step, if $a = 1$,
- no limit on the number of top-level rewriting steps, if $a = *$.

The interpretation for the s_i is that application of the i-th rule is allowed if $s_i = \mathsf{on}$ and disallowed otherwise.

But of course it does not suffice to modify the term that shall be rewritten. We also have to build the control mechanism into the f-rules $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n$. This is done by supplying all f-terms in all l_i and r_j with appropriate contexts: the i-th rule

$$f(t_1) \rightarrow \mathsf{if}\ b\ \mathsf{then}\ f(t_2)\ \mathsf{else}\ f(t_1)$$

becomes (assume that b, t_1 and t_2 do not contain f-terms)

$$f(t_1) @ \langle 1, s_1, \ldots, s_{i-1}, \mathsf{on}, s_{i+1}, \ldots, s_n \rangle \rightarrow$$
$$\mathsf{if}\ b\ \mathsf{then}\ f(t_2) @ \langle 0, \mathsf{on}, \ldots, \mathsf{on} \rangle$$
$$\mathsf{else}\ f(t_1) @ \langle 1, s_1, \ldots, s_{i-1}, \mathsf{off}, s_{i+1}, \ldots, s_n \rangle$$

In the then case, all rules are switched on, since the rule has been successfully applied. Switching rule i off in the else case is the "de-activation" that was mentioned above.

Usually, the introduction of contexts restricts the rewriting relation of a given system $\mathcal{R}$. Certain unwanted rewriting sequences are thrown out, but no additional sequences become possible. This is because every rewriting step in the system with contexts corresponds to a step in the original system; the terms themselves are not changed.

In order not to complicate the control of rewriting, we demand that f-rules must not have nested f-terms on their left-hand sides. This means that the connection between a rule and the corresponding switch position in contexts is easy to maintain.

3 Structured operational semantics

The operational definition of the semantics of a programming language L is accomplished by first defining an abstract machine M and then interpreting the constructs of L by means of the machine instructions of M. In Plotkin's approach, M is given in the form of a transition system:

Definition 2. A **transition system** is a triple $(\Gamma, T, \rightarrow)$, where Γ is the set of **configurations**, $T \subseteq \Gamma$ is the set of **terminal configurations**, and $\rightarrow \subseteq \Gamma \times \Gamma$ is the **transition relation** satisfying $(T \times \Gamma) \cap \rightarrow = \emptyset$.

In order to ease the modelling of interactions with the environment, transitions are usually *labelled* with *actions*: $\gamma_1 \xrightarrow{a} \gamma_2$ then means that the step from configuration γ_1 to γ_2 is taken while performing some action a together with the program's environment. Typical actions of that kind are communication events. There is, however, no greater expressive power in labelled systems. They can be simulated by unlabelled systems whose configurations have an extra component that contains the sequences of labels; therefore, there is no need to consider labelled systems.

The starting point in defining a transition system is the definition of the configurations. So let us assume the two sets Γ and T given. Furthermore, let us assume a signature Σ and a set of variables V such that $T(\Sigma, V)$ contains all the terms that we need to express configurations, contexts, and other mathematical objects we need. Special subsets of $T(\Sigma, V)$ are Γ' and T', representing schemata for configurations and terminal configurations, respectively. If no confusion can arise, we will identify configurations and their term representations.

The transition relation $\rightarrow$ is now defined by means of a special kind of deduction system:

Definition 3. An **SOS deduction system** for Γ and T consists of the term sets $T(\Sigma, V)$, Γ' and T', and inference rule schemata of the following kind:

$$\frac{\vdash \bigwedge_{i=1}^{p} b_i \ \wedge \ \bigwedge_{j+1}^{n} \gamma_j \xrightarrow{L_j} \gamma_j' \ \wedge \ \bigwedge_{k=1}^{q} B_k}{\vdash \overline{\gamma} \rightarrow \overline{\gamma}'}$$

The b_i are basic predicate terms restricting the input variables in $\overline{\gamma}$ ("preconditions"), and the B_j are predicate terms restricting the output variables not in $\overline{\gamma}$ ("postconditions"). In transitions, extra variables w. r. t. $\overline{\gamma}$ must only occur on the right-hand side; all variables in $\overline{\gamma}'$ must also occur in some other configuration term. L_j may be 1, denoting one-step transition, or $*$, denoting an arbitrary number of steps.[2] In the latter case, γ_j' must be terminal, i. e. a normal form.

The semantics of this kind of inference rules is as usual: An instance of the conclusion is established if the corresponding instance of the hypothesis can be established using the rules of the system. All variables of the rules are implicitly universally quantified.

As a consequence of this definition, the restriction to conjunctions in the hypothesis does not limit the expressive power of the formalism. Any quantifier-free hypothesis can be implemented by first transforming it into disjunctive normal form and then splitting the rule into several rules with the same conclusion, each component of the disjunction forming the hypothesis of a separate rule.

[2] L_j is *not* an action as they occur in labelled transition systems (see remarks above).

As an additional requirement for SOS deduction systems we demand that the rules do not permit non-terminating proof attempts. The simplest example for a rule that is forbidden is

$$\frac{\vdash \gamma \rightarrow \gamma'}{\vdash \gamma \rightarrow \gamma'}$$

It has the form of definition 3, but it cannot be used for proving any transition. A way to exclude such unpleasant behaviour is to demand that all transitions in the premise of a rule be smaller than the transition in the conclusion w. r. t. some well-founded ordering. The existence of such an ordering is sufficient to prevent non-terminating proof attempts.[3]

4 An example for transformation

In this chapter, we will informally describe how we can transform SOS deduction rules into term rewriting rules and why some other seemingly "obvious" ways do not work. The exact definition of this transformation can be found in [6].

4.1 The example language definition

As an example (artificial, but not overly simple) let us consider an extract from an imperative language L. In L, there is a syntactic class of statements, denoting state transformations, and the usual operator ";" for sequential composition:

$$Stmt \ni stmt \ ::= \ stmt_1; stmt_2 \mid \ldots$$

On the semantic side, we have a set of states Σ that statements can transform. The internal structure of states $\sigma \in \Sigma$ is not important to us. A configuration can either consist of a statement to be executed together with an initial state for this execution, or it can be the final state of an execution:

$$\Gamma_{Stmt} = Stmt \times \Sigma \cup T_{Stmt}$$
$$T_{Stmt} = \Sigma$$

The execution of a statement list proceeds from left to right. After one computation step, the first statement in a list may have terminated, resulting in a final state, or there may still be a rest of this statement waiting for execution. For these two possibilities, we have the following two inference rule schemata:

$$\frac{\vdash \langle stmt_1, \sigma \rangle \rightarrow_{Stmt} \sigma_1}{\vdash \langle stmt_1; stmt_2, \sigma \rangle \rightarrow_{Stmt} \langle stmt_2, \sigma_1 \rangle} \quad (1)$$

$$\frac{\vdash \langle stmt_1, \sigma \rangle \rightarrow_{Stmt} \langle stmt_1', \sigma_1 \rangle}{\vdash \langle stmt_1; stmt_2, \sigma \rangle \rightarrow_{Stmt} \langle stmt_1'; stmt_2, \sigma_1 \rangle} \quad (2)$$

Rule (1) deals with the case of termination of $stmt_1$ and (2) with the other case. $stmt_1'$ is what remains of $stmt_1$ after one computation step.

[3] This is the usual method to prove termination of rewrite systems. There the right-hand side of a rule must be smaller than its left-hand side.

4.2 Transformation into rewrite rules

The simplest possible approach to the problem of transforming a rule

$$\frac{\vdash \mathit{hypo}}{\vdash \gamma \rightarrow \gamma'}$$

into a rewrite rule is to simulate the rule's semantics ("if *hypo* holds, then the step from γ to γ' is possible") with the conditional operator if _ then _ else _:

$$\gamma \rightarrow \text{if } \mathit{hypo} \text{ then } \gamma' \text{ else } \gamma''$$

where γ'' has to be defined appropriately. But of course this is only possible when there are no extra variables in *hypo*, which is an exceptional case (e. g. both (1) and (2) contain extra variables, viz. σ_1 and stmt'_1). Furthermore, the problem of defining a suitable γ'' is not trivial (we will return to this problem).

So we have to be a little bit more inventive and have to find a way of disposing of the extra variables. Consider rule (1). The extra variable σ_1 stands for a terminal configuration that is related to $\langle\, \mathit{stmt}_1, \sigma \,\rangle$ by the transition relation. Viewing this relation more operationally, we can rephrase this as σ_1 standing for a possible (one-step) *result* of evaluating $\langle\, \mathit{stmt}_1, \sigma \,\rangle$.[4] The name σ_1 itself is irrelevant; we only need the property that it denotes a terminal configuration.

$\langle\, \mathit{stmt}_1, \sigma \,\rangle$ does not contain extra variables; so it may safely occur on the right-hand side of the rewrite rule that we are aiming at. Since we are interested in its result, we enclose it by an additional operator *eval* that is intended to yield the result of evaluating its argument. By using let abstraction, i. e. λ-terms, we can name this result σ_1, and we arrive at the rewrite rule

$$\langle\, \mathit{stmt}_1; \mathit{stmt}_2, \sigma \,\rangle \rightarrow \text{let } \sigma_1 = \mathit{eval}(\langle\, \mathit{stmt}_1, \sigma \,\rangle) \text{ in } \langle\, \mathit{stmt}_2, \sigma_1 \,\rangle \qquad (3)$$

Note that σ_1, although not appearing on the left-hand side, is not an extra variable. It is a bound variable of λ-calculus and, as much as term rewriting is concerned, it is just a constant of type T'. We assume that let terms are evaluated in applicative order (in *call by value* fashion).

So far, this looks like the kind of rule we wanted. But there still remains a problem. We have the other rule (2), and when we apply our procedure to this rule, we end up with a rewrite rule like

$$\langle\, \mathit{stmt}_1; \mathit{stmt}_2, \sigma \,\rangle \rightarrow \text{let } \mathit{cf} = \mathit{eval}(\langle\, \mathit{stmt}_1, \sigma \,\rangle) \text{ in } \langle\, \mathit{cf} \downarrow 1; \mathit{stmt}_2, \mathit{cf} \downarrow 2 \,\rangle \quad (4)$$

where cf is a variable of type $\mathit{Stmt} \times \Sigma$ and $\downarrow 1$ and $\downarrow 2$ are the projections to the first and second component of a tuple, respectively. Now the left-hand sides of (3) and (4) are identical, and each of the two rules can be applied in any case where the other could be applied, too. In (1) and (2), the decision which rule to apply is taken in the hypothesis by means of a type check. In order to get correct rewrite rules, we must add this kind of check as well: We must test whether the result of evaluating $\langle\, \mathit{stmt}_1, \sigma \,\rangle$ is terminal or not. And for the case that the result is non-terminal even though we chose the rule derived from (1),

[4] There may be more than one possible result if the language is non-deterministic.

we must provide a "way back" giving a result that still allows application of the other rule: choosing the wrong rule must not lead into a "dead end".

Implementing the type check is simple: it amounts to having rules of the form:

$$\langle\, stmt_1; stmt_2, \sigma \,\rangle \rightarrow \\ \quad \text{let } \sigma_1 = eval(\langle\, stmt_1, \sigma \,\rangle) \text{ in if } type(\sigma_1) = T \text{ then } \langle\, stmt_2, \sigma_1 \,\rangle \text{ else } \ldots \tag{5}$$

$$\langle\, stmt_1; stmt_2, \sigma \,\rangle \rightarrow \\ \quad \text{let } cf = eval(\langle\, stmt_1, \sigma \,\rangle) \text{ in} \\ \quad\quad \text{if } type(cf) = Stmt \times T \text{ then } \langle\, cf \downarrow 1; stmt_2, cf \downarrow 2 \,\rangle \text{ else } \ldots \tag{6}$$

More problematic is the "way back" that must be placed in the else parts of (5) and (6). Intuitively, we would demand that in these cases, the original configuration $\langle\, stmt_1; stmt_2, \sigma \,\rangle$ should remain unchanged. But we cannot simply put this into the else parts since it would render the rewrite system non-terminating: If the type check failed, the same rule could be applied over and over again.

So we must find a way to indicate that a rewrite rule has been tried in vain (i. e. its type check has been rewritten to false). For each of the rules generated from the SOS rules there must be a flag that can be raised when the else part is selected. This, however, is exactly the kind of situation that the concept of contexts has been defined for. Configuration terms never occur in nested form on the left-hand side of rewrite rules, so we can supply each of these with an appropriate context.

In our example, there are only two rules. Hence it suffices to introduce contexts as elements of $\{0, 1, *\} \times \{\mathsf{on}, \mathsf{off}\}^2$ and the desired rewrite rules become (for the one-step case)

$$\langle\, stmt_1; stmt_2, \sigma \,\rangle @ \langle\, 1, \mathsf{on}, s \,\rangle \rightarrow \\ \quad \text{let } \sigma_1 = eval(\langle\, stmt_1, \sigma \,\rangle @ \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle) \text{ in} \\ \quad \text{if } type(\sigma_1) = T \\ \quad\quad \text{then } \langle\, stmt_2, \sigma_1 \,\rangle @ \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle \\ \quad\quad \text{else } \langle\, stmt_1; stmt_2, \sigma \,\rangle @ \langle\, 1, \mathsf{off}, s \,\rangle \tag{7}$$

$$\langle\, stmt_1; stmt_2, \sigma \,\rangle @ \langle\, 1, s, \mathsf{on} \,\rangle \rightarrow \\ \quad \text{let } cf = eval(\langle\, stmt_1, \sigma \,\rangle @ \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle) \text{ in} \\ \quad \text{if } type(cf) = Stmt \times \Sigma \\ \quad\quad \text{then } \langle\, cf \downarrow 1; stmt_2, cf \downarrow 2 \,\rangle @ \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle \\ \quad\quad \text{else } \langle\, stmt_1; stmt_2, \sigma \,\rangle @ \langle\, 1, s, \mathsf{off} \,\rangle \tag{8}$$

The rules that define the operator *eval* guarantee that its argument is evaluated appropriately (see section 5.1); so $\langle\, stmt_1, \sigma \,\rangle$ is evaluated in one step only. Furthermore, we can easily prove that the else parts are smaller than the left-hand sides (under the well-founded ordering off < on); there is no termination problem when the type check fails. So we see that (7) and (8) are rules of the kind we have been looking for. In the following, we will call them **SOS-derived rules**.

What remains is to mention what happens if the hypothesis contains simple Boolean conditions. The preconditions b_i can safely be put into the type check since they do not contain extra variables. The conditions B_k restricting the intermediate and final configurations must also become part of the type check with the extra variables being replaced by suitable selection expressions in the style that has been used in rule (8). And finally, multiple transitions in the hypothesis are translated into iterated let expressions.

5 Properties of the transformed system

5.1 The basic rewrite system $\mathcal{B}$

The purpose of our transformation process is to provide a way to simulate program executions as specified by the operational semantics with the help of rewriting sequences. Since we want to employ "pure" rewriting and not rewriting modulo some equational theory, we also have to supply rewrite rules for modelling properties of the underlying data types. These rules form the basic rewrite system $\mathcal{B}$. We require $\mathcal{B}$ to allow all rewritings that are not directly connected to application of SOS rules. Especially, all conditions should be decidable by rewriting. This amounts to demanding that $\mathcal{B}$ be complete and correct in the logical sense (w. r. t. the standard interpretation of logical symbols), and also complete, i. e. confluent and terminating, in the sense of term rewriting.[5] So we have the **general assumption** that $\mathcal{B}$ provides (in the logical sense) a correct and complete decision procedure for all conditions that do not depend on the semantics definition. This means that each term expressing such a condition has exactly one normal form w. r. t. $\mathcal{B}$, viz. either true or false.

The operator *eval* plays a special rôle. It only occurs in terms of the form $eval(\gamma @ k)$, and its purpose is to ensure that its argument configuration γ is evaluated according to the context k. This is achieved by retaining the *eval* and the context as long as further evaluation is needed; only configurations in a 0-context or a terminal configuration in a *-context are completely evaluated. So we have the rules

$$eval(\gamma @ \langle 0, \ldots \rangle) \rightarrow \gamma$$
$$eval(t @ \langle *, \ldots \rangle) \rightarrow t$$

where γ is a variable for configurations and t for terminal configurations.

5.2 Simulation

The transformation procedure has been devised in order to produce a rewrite system $\mathcal{R}$ that models a semantics definition $\mathcal{S}$ as closely as possible, the characteristic feature being the set of possible transition sequences. Therefore the most interesting questions to ask about $\mathcal{R}$ are what rewriting sequences are possible and how they are related to the transition sequences of $\mathcal{S}$. We will see that the

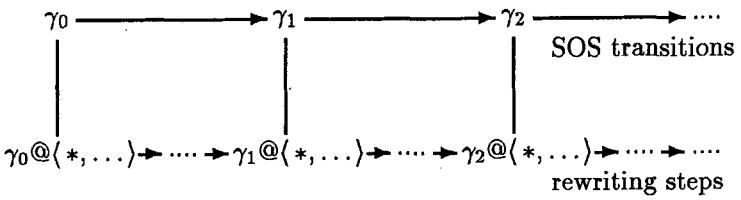
[5] The practical consequences of a system not fulfilling these demands are discussed in [7].

relation between $\mathcal{R}$ and $\mathcal{S}$ is indeed very close; if rewriting is considered modulo the equational theory of the basic system $\mathcal{B}$, we have a 1-1 correspondence between rewriting sequences and "flattened" transition sequences (where steps needed to establish a premise are also visible).

Let $\mathcal{S}$ be defined by the transition system $(\Gamma, T, \rightarrow_{\mathcal{S}})$, and let $\rightarrow_{\mathcal{S}}$ be given by a set of $N \in \mathbf{N}$ deduction rules. We will use the following additional abbreviations for contexts, where $k \in [N]$ and $r_j \in \{\mathsf{on}, \mathsf{off}\}$ for $j \in [N]$:

$$
\begin{aligned}
K_{0f} &=_{df} \langle 0, \mathsf{on}, \ldots, \mathsf{on} \rangle \\
K_1^{(k)} &=_{df} \langle 1, r_1, \ldots, r_{k-1}, \mathsf{on}, r_{k+1}, \ldots, r_N \rangle \\
K_f &=_{df} \langle *, \mathsf{on}, \ldots, \mathsf{on} \rangle \\
K^{(k)} &=_{df} \langle *, r_1, \ldots, r_{k-1}, \mathsf{on}, r_{k+1}, \ldots, r_N \rangle
\end{aligned}
$$

Overview: The simulation of $\mathcal{S}$ by $\mathcal{R}$ can be described as below. The intermediate terms in the rewriting sequence result from applying rules from $\mathcal{R}$ to configuration terms. They need not themselves be configuration terms, but they are equal to such a term modulo $=_R$.



Each transition step is modelled by a rewriting sequence in $\mathcal{R}$ which generally has more than just one step. In order to be allowed to perform one step $\gamma_1 \rightarrow_{\mathcal{S}} \gamma_2$ using the transition system, we usually have to perform other transitions before that correspond to premises of transition rules. These "hidden" transitions only contribute indirectly to $\gamma_1 \rightarrow_{\mathcal{S}} \gamma_2$ by determining parts of γ_2. So the transition process is not organized in linear form; each transition is equipped with a tree of other transitions (a proof tree) that justifies it. The corresponding rewriting process, however, can only construct flat sequences of terms. Therefore all the hidden transitions become part of the simulating sequence $\gamma_1 \mathbin{@} K_f \xrightarrow{*}_{\mathcal{R}} \gamma_2 \mathbin{@} K_f$ as well. Furthermore, rewriting makes use of the rules in $\mathcal{B}$, while transition takes place modulo $=_{\mathcal{B}}$.

Simulation works in the other direction as well. If we have a rewriting sequence that uses one SOS-derived rule, then this sequence corresponds to a transition sequence that is obtained via this particular SOS rule.

We have the following simulation results:

One-step completeness

$$
\begin{aligned}
&\forall \gamma, \gamma' \in \Gamma\ \forall k \in [N]\ \forall r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_N \in \{\mathsf{on}, \mathsf{off}\} : \\
&\quad \text{if } \gamma \rightarrow_{\mathcal{S}} \gamma' \text{ using rule } k \text{ then } \gamma \mathbin{@} K_1^{(k)} \xrightarrow{+}_{\mathcal{R}} \gamma' \mathbin{@} K_{0f}
\end{aligned}
\tag{9}
$$

This is the basic building stone for the simulation results, viz. the simulation of one transition step by a rewriting sequence.

One-step correctness

$$\begin{aligned}&\forall \gamma, \gamma' \in \Gamma\ \forall k \in [N]\ \forall r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_N \in \{\mathsf{on}, \mathsf{off}\} : \\ &\quad \text{if } \gamma \mathbin{@} K_1^{(k)} \xrightarrow{+}_{\mathcal{R}} \gamma' \mathbin{@} K_{0f} \text{ then } \gamma \to_{\mathcal{S}} \gamma'\end{aligned} \tag{10}$$

The names "correctness" and "completeness" are used in the logical sense: Any transition corresponds to a rewriting sequence (completeness), and any rewriting sequence that contains one outermost application of an SOS-derived rule corresponds to a transition step (correctness). Note how the use of 1-contexts $K_1^{(k)}$ restricts rewriting to exactly one transition-related step.

Normal form completeness and correctness

$$\forall \gamma \in \Gamma\ \forall t \in T : \gamma \xrightarrow{*}_{\mathcal{S}} t \ \Leftrightarrow\ \gamma \mathbin{@} K_f \xrightarrow{*}_{\mathcal{R}} t \mathbin{@} K_f \tag{11}$$

Building up inductively from the one-step results, we can obtain simulation properties for longer transition sequences. One special case is of particular interest: sequences that end with a terminal configuration describe the complete evaluation of their initial configuration. Furthermore, expressions like $\gamma \xrightarrow{*}_{\mathcal{S}} t$ ($\gamma \in \Gamma, t \in T$) may occur in the premises of SOS rules.

The proofs for these results can be found in [6]. Because one-step and normal form transitions are intertwined via transitions in the premises of rules, all results must be proved by one simultaneous induction (on the number of applications of SOS-derived rules).

Divergence completeness

From one-step completess, we immediately obtain that each infinite transition sequence corresponds to an infinite rewriting sequence. So non-termination is preserved by the rewrite system:

$$\begin{aligned}&\forall \{\gamma^{(i)}\} \in \Gamma^{\mathbf{N}} : \\ &\quad (\forall i \in \mathbf{N} : \gamma^{(i)} \to_{\mathcal{S}} \gamma^{(i+1)}) \ \Rightarrow\ (\forall i \in \mathbf{N} : \gamma^{(i)} \mathbin{@} K_f \xrightarrow{*}_{\mathcal{R}} \gamma^{(i+1)} \mathbin{@} K_f)\end{aligned} \tag{12}$$

Divergence correctness

On the other hand, all infinite rewriting sequences correpond to "infinite behaviour" of the transition system. Because of the additional requirement in section 3 about terminating transition systems, the rewriting sequence keeps "coming back" to configurations, i. e. each tail of the sequence contains a configuration-context pair; therefore one-step correctness yields the existence of a corresponding infinite transition sequence:

$$\begin{aligned}&\forall \gamma \in \Gamma\ \forall \{t^{(i)}\} \in T(\Sigma, V)^{\mathbf{N}} : t^{(1)} = \gamma \mathbin{@} K_f \ \wedge\ (\forall i \in \mathbf{N} : t^{(i)} \to_{\mathcal{R}} t^{(i+1)}) \\ &\quad \Rightarrow\ \exists\, \{\gamma^{(i)}\} \in \Gamma^{\mathbf{N}}\ \exists j : \mathbf{N} \to \mathbf{N} \text{ strictly monotonic} : \\ &\qquad (\forall i \in \mathbf{N} : \gamma^{(i)} = (t^{(j(i))} \downarrow 1) \ \wedge\ \gamma^{(i)} \to_{\mathcal{S}} \gamma^{(i+1)})\end{aligned} \tag{13}$$

5.3 Confluence and termination

The system $\mathcal{R}$ consists of two parts: the basic system $\mathcal{B}$ and the system $\mathcal{R}'$ containing the SOS-derived rules. As already mentioned in section 5.1, we assume

$\mathcal{B}$ to be complete, i. e. confluent and terminating, so there are no problems with this part. But for $\mathcal{R}'$, the situation is totally different because these properties are completely determined by the semantics of the language L.

As we have seen in the previous section, every rewriting sequence in $\mathcal{R}$ has a direct counterpart in $\mathcal{S}$ and vice versa. This has immediate consequences for confluence and termination. Assume $\mathcal{R}$ is terminating. This means that there is no configuration-context pair that is the initial point for an infinite rewriting sequence. Therefore there is also no configuration that starts an infinite transition sequence in $\mathcal{S}$. Obviously, this property is equivalent to L being a language that only contains terminating programs.

For confluence, the situation is very similar. Consider rewriting modulo the equational theory E generated by the basic system $\mathcal{B}$. Then the only rewrite rules that we need are those derived from the SOS system. Confluence of this rewrite system means that every configuration has at most one normal form (modulo E). As a consequence, for each initial state and each program starting in this state, there is at most one final state, and hence the programming language must be deterministic.[6]

So typically $\mathcal{R}$ is not complete. In most cases, it will be non-terminating, and therefore normalization of configuration terms must be handled with care. Languages in the tradition of CSP ([16]) and Occam ([17]) do not even lead to confluent systems since they contain a non-deterministic choice operator. This might seem a serious drawback of the method, but it only reflects the desire to have a rewrite system that models the semantics as closely as possible. And the problem is very well known: Interpreters for functional languages, say, usually do not terminate when interpreting programs that are (semantically) "non-terminating", disregarding restrictions like finite stack size.

There is also no point in completing the system $\mathcal{R}$, e. g. by applying the Knuth-Bendix procedure (cf. [19]). Completion would add new rules to the system, and for these rules there would be no counterpart in the original SOS system. So the simulation property would disappear; essentially, the result of completion corresponds to a language where all non-determinism has been artificially removed by declaring different results for one program as equal.

6 Application

The method presented has been successfully applied in solving a problem originating from the ProCoS project. For a language named $\mathrm{PL}_0^{\mathrm{R}}$, two different semantics definitions have been given (SOS and denotational), and the aim is to prove their equivalence. In [20], a standard mathematical hand proof is presented, its single steps mainly being based on induction on the structure of programs. A typical feature of such proofs is that subproofs are repeated in several places identically or only slightly modified, due to the similarity of semantics definitions for different language constructs. Therefore the use of an automated tool to check hand proofs or to assist in them is desirable.

[6] This requirement can be slightly weakened; e. g. the evaluation order of parameters for function calls is unimportant as long as this evaluation has no side effects.

First results of applying the Larch Prover ([11]) to this problem are reported in [7]. By now, all the essential steps of the whole equivalence proof have been completed. For the basic idea behind application of the transformed rewrite system $\mathcal{R}$ let us consider the subproblem of proving the equivalence of the semantics definitions of expressions (in a slightly simplified form).

Expressions $exp \in Expr$ are evaluated w. r. t. to an environment $\rho \in \mathsf{OpEnv}$ and a state $\sigma \in \Sigma$, whose internal structure are not important here. The result is a value $v \in \mathsf{Val}$. So configurations for the operational semantics are either tuples $\langle exp, \rho, \sigma \rangle \in Expr \times \mathsf{OpEnv} \times \Sigma$ (non-terminal) or values from Val (terminal). The denotational semantics for expressions is given by a function $\mathcal{E} : Expr \times \mathsf{OpEnv} \times \mathsf{State} \to \mathsf{Val}$, and we have to prove:

$$\forall exp \in Expr, \rho \in \mathsf{OpEnv}, \sigma \in \Sigma : \langle exp, \rho, \sigma \rangle \longrightarrow_{Expr} \mathcal{E}[\![exp]\!]\,\rho\sigma \tag{14}$$

where $\longrightarrow_{Expr}$ is the SOS transition relation for expressions.

Now we transform $\longrightarrow_{Expr}$ into rewrite rules as described above, generating the system $\mathcal{R} = \mathcal{R}' \cup \mathcal{B}$. The denotational semantics is defined in form of equations that can directly be turned into rewrite rules; these rules belong to the basic system $\mathcal{B}$.

The proof that we have performed with the help of LP proceeds by induction on the structure of expressions. This form of induction is supported by LP; the necessary induction subgoals and hypotheses are generated automatically. For given exp, ρ and σ, the proof is structured as follows:

- First $\mathcal{E}[\![exp]\!]\,\rho\sigma$ is evaluated using rules from $\mathcal{B}$; this results in some term γ_1.
- Next, the configuration/context term $t \equiv eval(\langle exp, \rho, \sigma \rangle @ \langle 1, \mathsf{on}, \ldots, \mathsf{on} \rangle)$ is evaluated using all the rules in $\mathcal{R}$; this results in some term γ_2. In this phase, the induction hypotheses for the subexpressions of exp will also be used as rewrite rules.
  Note that the rules for *eval* (cf. section 5.1) together with the 1-context in t guarantee a one-step evaluation of $\langle exp, \rho, \sigma \rangle$.
- Finally, equality of γ_1 and γ_2 is proved using the rules of $\mathcal{B}$.

After the last step, we have proved

$$t \equiv eval(\langle exp, \rho, \sigma \rangle @ \langle 1, \mathsf{on}, \ldots, \mathsf{on} \rangle) \xrightarrow{*}_{\mathcal{R}} \gamma_2 =_{\mathcal{B}} \gamma_1 \;{}_{\mathcal{B}}\!\xleftarrow{*} \mathcal{E}[\![exp]\!]\,\rho\sigma$$

eval operators are only introduced by the rules simulating the SOS transition rules. Thus γ_1 cannot contain such an operator, and there must be a point in the above rewriting sequence where it is removed from t. From the special form of the *eval* elimination rules (cf. section 5.1), it follows that there must be an intermediate configuration term with a 0-context in the rewriting sequence (this term is the result of a successful application of a rule from $\mathcal{R}'$):

$$eval(\langle exp, \rho, \sigma \rangle @ \langle 1, \mathsf{on}, \ldots, \mathsf{on} \rangle) \xrightarrow{+}_{\mathcal{R}} eval(\gamma_2' @ \langle 0, \ldots \rangle) \xrightarrow{*}_{\mathcal{B}} \gamma_2$$

The *eval* elimination could not be applied in the first subsequence; so we have

$$\langle exp, \rho, \sigma \rangle @ \langle 1, \mathsf{on}, \ldots, \mathsf{on} \rangle \xrightarrow{+}_{\mathcal{R}} \gamma_2' @ \langle 0, \ldots \rangle$$

and by one-step correctness (9) the rewriting proof turns out to be sufficient to prove the original goal (14) since the $\rightarrow_{\mathcal{B}}$ steps can be neglected (transitions with the SOS system take take place modulo $=_{\mathcal{B}}$).

Proofs about non-deterministic languages normally cannot be performed in a single step. For in such proofs, each possible result for a non-deterministic construct has to be considered, and this is most easily done by subsequently selecting all branches (by deleting those SOS rules that lead to other branches).

The Larch Prover turned out to be a suitable tool for implementing the transformed SOS rules since it is largely oriented towards easy formulation and application of rewrite rules. A major advantage of the system concerning our simulation is that normally intermediate results occurring in rewriting sequences are not displayed. This means that appliciation of the rules in $\mathcal{R}'$ remains completely hidden, and therefore the user is not confused by terms appearing during β-reduction or evaluation of type check conditions in SOS rules. Since the transformation of SOS rules into rewrite rules is very systematic, it was easy to implement a tool that generates LP input from SOS system descriptions. This tool has proved very helpful in the example proofs.

Another useful feature of LP's is that the user can control the rewriting process to a very large extent. E. g. the evaluation order can be changed (*inside-out* or *outside-in*), and rewrite rules can be prevented from being considered. This can be used to increase the efficiency of the system, since typically large groups of rules are known not to be applicable, and so the time for testing whether they match a given term can be saved.

Among the disadvantages of LP are the lack of powerful proof control mechanisms like strategies or tactics (as they are provided e. g. in HOL [13] or in KIV [15]) and its weak type concept. LP only supports a subset of many-sorted first-order logic. During our experiments, this never prevented any proofs, but it made their formulation a lot more complicated. In an order-sorted system like OBJ3 [12] or PVS [22], terms could be kept smaller [7], and in a higher-order logic (present e. g. in HOL and PVS), many properties could be formulated much easier, since the rules about function application and extensionality would be supplied automatically. These rules can be simulated in LP's first-order logic, but a large number of explicit rules are needed for this.

7 Conclusion

In this paper, we have presented a way to simulate a special form of SOS definitions by standard term rewriting systems. Another approach to relating SOS definitions and equational logic is presented in [2]. Here an algorithm is described that transforms SOS rules into set of equations such that every true formula about the language can be proved in this theory. In [3] it is proved that for SOS definitions that only describe finite systems, these equations can be transformed into a complete term rewriting system. This technique, however, is restricted to SOS rules in a special format (GSOS) that is incomparable to the

[7] If we have sorts $T_1 \subseteq T_2 \subseteq T_3$, we would like to be allowed to write $t \in T_3$ for all $t \in T_1$. In LP, however, we have to write the injections, e. g. $in\text{-}T_3(in\text{-}T_2(t))$.

format of definition 3. There exist other special rule formats, e. g. the *tyft/tyxt* format of [14], that have received special attention because of certain pleasant properties that such systems possess. These formats are less restrictive than the GSOS format, but such definitions have not yet been axiomatized by equations or rewrite rules. [8] shows how to model SOS definitions with other methods than term rewriting, using the facilities of the HOL system.

Other ways to combine λ-calculus and term rewriting are described in [18] and [10]. But both approaches extend the λ-calculus, and do not try to include some of its concepts in pure term rewriting.

Our method provides a very close simulation of transition rules by standard term rewriting rules. Since no additional formalism is needed, ordinary rewriting-based proof assistants can be used to implement it in reasoning about SOS definitions. In [7], we have described a successful example application where the Larch Prover [11] is used to assist in a proof of equivalence between an operational and a denotational semantics definition. During this experiment, it turned out that the rather complicated structure of the SOS-derived rules does not lead to difficulties. On the one hand, these rules can be generated automatically from the original SOS form, due to their systematic definition. And on the other hand, their application usually remains hidden in normalization processes. This means that the user of the proof tool need not worry about involved intermediate terms, but can concentrate on the logical structure of the proof.

Acknowledgements: I would like to thank the ProCoS group in Kiel, especially Bettina Buth, Yassine Lakhneche and Markus Müller-Olm, for their help in clarifying my ideas, Ursula Martin for drawing my attention to [1], and the anonymous referees for many helpful comments.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.
2. Luca Aceto, Bard Bloom, and Frits Vaandrager. Turning SOS rules into equations. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science, Santa Cruz, CA*, pages 113–124, 1992. Full version available as CWI Report CS-R 9218, Centrum voor Wiskunde en Informatica, Amsterdam, June 1992.
3. Doeko Bosscher. Term rewriting properties of SOS axiomatisations. In *Proceedings of the Conference on Theoretical Aspects of Computer Science*, LNCS. Springer-Verlag, 1994. To appear.
4. Jonathan Bowen et al.. A ProCoS II project description: ESPRIT Basic Research project 7071. *Bulletin of the EATCS* **50**, 128–137, 1993.
5. N. De Bruijn. Lambda-calculus notation with nameless dummies. *Indagationes Mathematicae* **34**, 381–392, 1972.
6. Karl-Heinz Buth. Simulation of transition systems with term rewriting systems. Bericht 9212, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, 1992.

7. KARL-HEINZ BUTH. Using SOS definitions in term rewriting proofs. In URSULA H. MARTIN AND JEANNETTE M. WING, editors, *Proceedings of the First International Workshop on Larch, Dedham, MA, 1992*, Workshops in Computing Series, pages 36–54. Springer-Verlag, 1993.
8. JUANITO CAMILLERI AND TOM MELHAM. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.
9. NACHUM DERSHOWITZ AND JEAN-PIERRE JOUANNAUD. Rewrite systems. In JAN VAN LEEUWEN, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, chapter 6, pages 243–320. Elsevier/MIT Press, 1990.
10. DANIEL J. DOUGHERTY. Adding algebraic rewriting to the untyped lambda calculus. In RONALD V. BOOK, editor, *Proceedings of the 4th International Conference on Rewriting Techniques and Applications, Como, Italy*, LNCS 488, pages 37–48. Springer-Verlag, April 1991.
11. STEPHEN J. GARLAND AND JOHN V. GUTTAG. An overview of LP, the Larch Prover. In NACHUM DERSHOWITZ, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, LNCS 355, pages 137–155. Springer-Verlag, 1989.
12. JOSEPH A. GOGUEN. OBJ as a theorem prover with applications to hardware verification. Technical Report SRI-CSL-88-4R2, SRI International, August 1988.
13. MICHAEL J. C. GORDON. HOL: A proof generating system for higher-order logic. In G. BIRTWISTLE AND P.A. SUBRAMANYAM, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.
14. JAN FRISO GROOTE AND FRITS VAANDRAGER. Structured operational semantics and bisimulation as a congruence. *Information and Computation* **100**(2), 202–260, October 1992.
15. MARITTA HEISEL, WOLFGANG REIF, AND WERNER STEPHAN. Tactical theorem proving in program verification. In MARK E. STICKEL, editor, *Proceedings of the 10th International Conference on Automated Deduction*, LNCS 449, pages 117–131. Springer-Verlag, 1990.
16. C. A. R. HOARE. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.
17. INMOS LTD. *occam 2 Reference Manual*. Series in Computer Science. Prentice-Hall International, 1988.
18. STEFAN KAHRS. *λ-rewriting*. PhD thesis, Fachbereich Mathematik und Informatik, Universität Bremen, January 1991.
19. DONALD E. KNUTH AND PETER B. BENDIX. Simple word problems in universal algebras. In J. LEECH, editor, *Proceedings of the Conference on Computational Problems in Abstract Algebra, Oxford, 1967*, pages 263–298. Pergamon Press, 1970.
20. YASSINE LAKHNECHE. Equivalence of denotational and structural operational semantics of PL_0^R. ProCoS Technical Report Kiel YL1, Christian-Albrechts-Universität Kiel, 1991.
21. GORDON D. PLOTKIN. An operational semantics for CSP. In DINES BJØRNER, editor, *Formal Description of Programming Concepts - II*, pages 199–225. North-Holland, 1983.
22. JOHN RUSHBY. A tutorial on specification and verification using PVS. In JAMES C. P. WOODCOCK AND PETER GORM LARSEN, editors, *Tutorial Material for FME '93: Industrial-Strength Formal Methods. Proceedings of the First International Symposium of Formal Methods Europe, Odense, Denmark*, pages 357–406, April 1993.