# Generating Specialized Interpreters for Modular Structural Operational Semantics

Casper Bach Poulsen and Peter D. Mosses

Department of Computer Science, Swansea University, Swansea, SA2 8PP, UK
cscbp@swansea.ac.uk, p.d.mosses@swansea.ac.uk

**Abstract.** Modular Structural Operational Semantics (MSOS) is a variant of Structural Operational Semantics (SOS). It allows language constructs to be specified independently, such that no reformulation of existing rules in an MSOS specification is required when a language is extended with new constructs and features.
Introducing the Prolog MSOS Tool, we recall how to synthesize executable interpreters from small-step MSOS specifications by compiling MSOS rules into Prolog clauses. Implementing the transitive closure of compiled small-step rules gives an executable interpreter in Prolog. In the worst case, such interpreters traverse each intermediate program term in its full depth, resulting in a significant overhead in each step.
We show how to transform small-step MSOS specifications into corresponding big-step specifications via a two-step specialization by internalizing the rules implementing the transitive closure in MSOS and 'refocusing' the small-step rules. Specialized specifications result in generated interpreters with significantly reduced interpretive overhead.

**Keywords:** interpreter generation, structural operational semantics, Modular SOS, specialization, partial evaluation, program derivation, refocusing

## 1 Introduction

*Background.* Structural operational semantics (SOS) [21] provides a simple and direct method for specifying the semantics of programming language constructs and process algebras. The behaviour of constructs defined by an SOS is modelled by a labelled transition system whose transition relation is defined by a set of inference rules and axioms. For programming language semantics, the configurations of the transition system are typically given by terms and auxiliary entities, such as stores (recording the values of imperative variables before and after each transition step) and environments (determining the bindings of identifiers). In conventional SOS, auxiliary entities are explicit in all rules. This gives rise to the modularity problem with SOS: language extensions involving new auxiliary entities require reformulation of existing rules. Modular SOS (MSOS) [16] solves the modularity problem in SOS by implicitly propagating all unmentioned auxiliary entities.

Besides propagating auxiliary entities, small-step (M)SOS rules relate terms to partly evaluated terms. Evaluation in small-step (M)SOS specifications is given by a sequence of transition steps that eventually reaches a final state. In contrast, big-step (M)SOS rules relate terms directly to final states. As illustrated elsewhere [2,16], small-step rules are typically more concise than big-step rules for programming languages with abrupt termination and/or divergence.

The *PLanCompS*[1] project is developing an open-ended set of reusable *fundamental constructs* (or *funcons*), whose dynamic semantics is given by small-step MSOS rules[2]. Translating concrete constructs of a programming language into fundamental constructs gives a *component-based semantics*. MSOS rules provide a basis for verification, using, e.g., bisimulation [5,18] or structural induction on the underlying MSOS rules [16], and prototype interpreter generation. In this paper we focus on generating prototype interpreters in Prolog.

*Contribution.* It is well-known that big-step SOS rules can be compiled into Prolog clauses [7], and compilation of small-step MSOS rules into Prolog clauses has been utilized and hinted at in earlier publications [5,15–17]. The present paper presents the first systematic account of how to synthesize executable interpreters in Prolog from small-step MSOS specifications. We also assess and show how to reduce interpretive overhead in these interpreters.

The efficiency of generated interpreters is significantly improved by adapting *refocusing* [9] to MSOS. This is achieved by specializing a *refocusing rule* wrt an MSOS specification. The specialization forces evaluation of sub-terms, effectively transforming small-step rules into big-step rules. Compiling these big-step rules gives interpreters that avoid the computational overhead of decomposing the program term in each intermediate step, which previous interpreters generated from small-step MSOS specifications [3,5,16,17] have suffered from.

Through a subsequent specialization step, called *striding*, a small-step specification is transformed into its corresponding big-step counterpart by compressing corridor transitions, in a similar style to [8]. By *left-factoring* [1,20] the resulting big-step specification, back-tracking in generated interpreters is minimized.

We demonstrate and illustrate our techniques on MSOS specifications due to the pragmatic advantages of MSOS over SOS, but expect that the techniques are straightforward to extend to SOS.

*Related work.* The Maude MSOS Tool [3] executes MSOS specifications encoded as rewriting logic rules in Maude [6]. It allows for elegant representation of MSOS rules utilizing Maude features such as sorts and records. The approach to interpreting MSOS specifications is essentially similar to that of the Prolog MSOS Tool, where evaluation is implemented by sequences of small-step transitions, resulting in a significant overhead in each step.

The refocusing rule that we introduce is inspired by the work on refocusing by Danvy et al. [8,9]. That work is based on program transformations applied to

[1] Programming Language Components and Specifications: `www.plancomps.org`

[2] In fact, funcons are specified using Implicitly Modular SOS [19], a variant of MSOS with syntax closer to SOS.

functional programs implementing reduction semantics. In contrast, the specialization we present here applies directly to MSOS rules, and is based on simple rule unfolding.

Partial evaluation in logic programming [10,13] has been extensively studied as a means of compiling programs and speeding up interpreters based on binding time analyses. The specializations that we consider here correspond to partial evaluations of the refocusing rule wrt to small-step inference rules.

*Horn logical semantics* [11] uses Horn clauses to relate terms to values or denotations. The big-step style inherent to Horn logical semantics makes specification of control instructions challenging, as witnessed by Wang et al.'s suggestion of using Horn logical continuation-based semantics [22] to handle abrupt termination: in the continuation-based approach each predicate is parameterized over terms, semantic domains, control stacks, and continuations. In contrast, small-step MSOS can deal with abrupt termination without parameterizing and modifying existing rules. This paper uses small-step MSOS for specification, and describes how to systematically derive a corresponding big-step specification by specialization.

Refocused rules bear a striking resemblance to Charguéraud's pretty-big-step rules [4]. As demonstrated in [2], pretty-big-step rules can be derived from small-step rules by unfolding refocused rules.

*Outline.* Section 2 reviews MSOS. Section 3 recalls how the Prolog MSOS Tool compiles MSOS rules into Prolog clauses. Section 4 shows how to improve the efficiency of generated interpreters by refocusing. Section 5 introduces the striding transformation, which unfolds refocused rules into classic big-step rules. The efficiency of generated naïve, refocused, and striding interpreters is assessed in Sect. 6. Section 7 concludes and suggests further lines of research.

## 2 Modular Structural Operational Semantics

This section outlines the main features of MSOS by comparing it with SOS.

### 2.1 An Example SOS

SOS rules define possible transitions between configurations in an underlying labelled transition system. In SOS, a configuration $\gamma$ can make a transition to $\gamma'$ if: (1) $\gamma$ matches the *conclusion source* of an SOS rule

$$\frac{C_1 \qquad \cdots \qquad C_n}{\gamma \xrightarrow{\alpha} \gamma'}$$

where $\gamma \xrightarrow{\alpha} \gamma'$ is the rule conclusion, $\alpha$ is a (possibly empty) *transition label*, and $C_i$ are the *premises* (e.g., transition steps or side-conditions) of the rule; and (2) using only SOS rules, for each premise $C_i$ we can construct an upwardly branching derivation tree whose leaves are axioms, i.e., rules with empty premises

and satisfied side-conditions[3]. For a more detailed introduction to (M)SOS, the reader is referred to [16,21].

The following SOS rules define the applicative constructs $\mathsf{let}(id, e_1, e_2)$ and $\mathsf{bound}(id)$. We let $\rho$ range over environments, $id$ over identifiers, $e$ over expressions, and $v$ over values. The formula $\rho \vdash \gamma \to \gamma'$ asserts that $\gamma$ makes a transition to $\gamma'$ under environment $\rho$. $\rho[id \mapsto v]$ returns an environment $\rho'$ where $\rho'(id) = v$ and $\rho'(id') = \rho(id')$ for $id' \neq id$.

$$\frac{\rho \vdash e_1 \to e_1'}{\rho \vdash \mathsf{let}(id, e_1, e_2) \to \mathsf{let}(id, e_1', e_2)}\ [\textsc{let1-sos}] \qquad \frac{\rho[id \mapsto v] \vdash e_2 \to e_2'}{\rho \vdash \mathsf{let}(id, v, e_2) \to \mathsf{let}(id, v, e_2')}\ [\textsc{let2-sos}]$$

$$\frac{}{\rho \vdash \mathsf{let}(id, v_1, v_2) \to v_2}\ [\textsc{let3-sos}] \qquad \frac{\rho(id) = v}{\rho \vdash \mathsf{bound}(id) \to v}\ [\textsc{bound-sos}]$$

We now turn our attention to a semantics for sequential composition, $\mathsf{seq}(e_1, e_2)$, variable assignment, $\mathsf{assign}(\mathit{ref}, e)$, and variable dereferencing, $\mathsf{deref}(\mathit{ref})$. We let $\sigma$ range over stores, *ref* over references, and skip is a value. The formula $\langle e, \sigma\rangle \to \langle e', \sigma'\rangle$ asserts that the configuration given by term $e$ and store $\sigma$ can make a transition to the configuration given by term $e'$ and store $\sigma'$.

$$\frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma'\rangle}{\langle \mathsf{seq}(e_1, e_2), \sigma\rangle \to \langle \mathsf{seq}(e_1', e_2), \sigma'\rangle}\ [\textsc{seq1-sos}] \qquad \frac{}{\langle \mathsf{seq}(\mathsf{skip}, e_2), \sigma\rangle \to \langle e_2, \sigma\rangle}\ [\textsc{seq2-sos}]$$

$$\frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma'\rangle}{\langle \mathsf{assign}(\mathit{ref}, e_1), \sigma\rangle \to \langle \mathsf{assign}(\mathit{ref}, e_1'), \sigma'\rangle}\ [\textsc{asn1-sos}]$$

$$\frac{\sigma' = \sigma[\mathit{ref} \mapsto v]}{\langle \mathsf{assign}(\mathit{ref}, v), \sigma\rangle \to \langle \mathsf{skip}, \sigma'\rangle}\ [\textsc{asn2-sos}] \qquad \frac{\sigma(\mathit{ref}) = v}{\langle \mathsf{deref}(\mathit{ref}), \sigma\rangle \to \langle v, \sigma\rangle}\ [\textsc{deref-sos}]$$

Combining the constructs let, bound, seq, assign, and deref in SOS requires that we reformulate all rules: the rules for let and bound must propagate a store $\sigma$; similarly, seq, assign, and deref must propagate an environment $\rho$. We refrain from this tedious reformulation, and use MSOS instead.

### 2.2 Modular SOS

Like in SOS, MSOS rules define transition steps; i.e., a configuration makes a transition if we can construct a derivation tree using the rules defining the transition relation. Auxiliary entities in MSOS are encoded in the label of the transition relation, and are only explicitly mentioned when required. For example, in Fig. 1 the [LET1] rule makes no explicit mention of auxiliary entities, since they are not explicitly used by that rule. Crucially, computations in MSOS require labels on consecutive transitions to be *composable*. The remainder of this section defines MSOS labels and label composition.

[3] This notion of transition is based on positive SOS specifications. Rules with negative premises are not considered here.

$$\frac{e_1 \xrightarrow{\{\ldots\}} e_1'}{\mathsf{let}(id, e_1, e_2) \xrightarrow{\{\ldots\}} \mathsf{let}(id, e_1', e_2)}\ [\textsc{let1}] \qquad \frac{e_2 \xrightarrow{\{\mathbf{env}=\rho[id\mapsto v_1],\ldots\}} e_2'}{\mathsf{let}(id, v_1, e_2) \xrightarrow{\{\mathbf{env}=\rho,\ldots\}} \mathsf{let}(id, v_1, e_2')}\ [\textsc{let2}]$$

$$\frac{}{\mathsf{let}(id, v_1, v_2) \xrightarrow{\{\_\}} v_2}\ [\textsc{let3}] \qquad \frac{\rho(id) = v}{\mathsf{bound}(id) \xrightarrow{\{\mathbf{env}=\rho,\_\}} v}\ [\textsc{bound}]$$

$$\frac{e_1 \xrightarrow{\{\ldots\}} e_1'}{\mathsf{seq}(e_1, e_2) \xrightarrow{\{\ldots\}} \mathsf{seq}(e_1', e_2)}\ [\textsc{seq1}] \qquad \frac{}{\mathsf{seq}(\mathsf{skip}, e_2) \xrightarrow{\{\_\}} e_2}\ [\textsc{seq2}]$$

$$\frac{e_1 \xrightarrow{\{\ldots\}} e_1'}{\mathsf{assign}(ref, e_1) \xrightarrow{\{\ldots\}} \mathsf{assign}(ref, e_1')}\ [\textsc{asn1}]$$

$$\frac{\sigma' = \sigma[ref \mapsto v]}{\mathsf{assign}(ref, v) \xrightarrow{\{\mathbf{sto}=\sigma,\mathbf{sto}'=\sigma',\_\}} \mathsf{skip}}\ [\textsc{asn2}] \qquad \frac{\sigma(ref) = v}{\mathsf{deref}(ref) \xrightarrow{\{\mathbf{sto}=\sigma,\_\}} v}\ [\textsc{deref}]$$

Fig. 1: MSOS rules for example constructs

**Definition 1 (MSOS Label).** *An* MSOS label $L$ *is an unordered set of* label components*, where each* label component $\mathbf{ix}=E$ *consists of a distinct label index* $\mathbf{ix}$ *and an auxiliary entity* $E$ *such that each index is either unprimed (e.g.,* $\mathbf{env}$*) meaning the label is* readable*, or primed (e.g.,* $\mathbf{sto}'$*) meaning the label is* writable*.*

Label variables *refer to sets of label components. The label variable '—' ranges over sets of* unobservable *label components, while label variables '...',* $X, Y$*, etc. refer to sets of arbitrary label components.*

Informally, a label component is *observable* if it exhibits side effects. An example of an observable label component is the component pair $\mathbf{sto}=\sigma, \mathbf{sto}'=\sigma'$ such that $\sigma \neq \sigma'$. The change from $\mathbf{sto}$ to $\mathbf{sto}'$ is an observable side effect. Another example of an observable component is illustrated by the print construct:

$$\frac{}{\mathsf{print}(v) \xrightarrow{\{\mathbf{out}'=[v],\_\}} \mathsf{skip}}\ [\textsc{print}]$$

The $\mathbf{out}'$ component represents an output channel. An output channel may emit observable output several times during program execution. The observable output of evaluating the print construct above is the single element list $[v]$. The label component is unobservable when it contains the empty list, i.e., $\mathbf{out}'=[\,]$.

Environments, stores, and output channels each exemplify a distinct category of label components. These categories define how information is propagated between consecutive transition labels (i.e., how labels *compose*). Labels are defined by arrows in a category. The category gives the semantics of label composition [14,16]. For the purpose of this paper, the following definition of a label composition operator '$\circ$' suffices:

– Read-only label components (e.g., environments) remain unchanged between consecutive transition steps; e.g., $\{\mathbf{env}=\rho\} \circ \{\mathbf{env}=\rho\} = \{\mathbf{env}=\rho\}$.

- Read-write label components (e.g., stores) compose like binary relations; e.g., $\{\mathbf{sto}=\sigma', \mathbf{sto'}=\sigma''\} \circ \{\mathbf{sto}=\sigma, \mathbf{sto'}=\sigma'\} = \{\mathbf{sto}=\sigma, \mathbf{sto'}=\sigma''\}$.
- Write-only label components (e.g., output channels) are monoidal, generating lists of observable outputs; e.g., $\{\mathbf{out'}=l_2\} \circ \{\mathbf{out'}=l_1\} = \{\mathbf{out'}=l_1 \cdot l_2\}$, where '$\cdot$' is list concatenation, and $l_1, l_2$ are lists.

The formula $\mathsf{assign}(\mathit{ref}, v) \xrightarrow{\{\mathbf{sto}=\sigma, \mathbf{sto'}=\sigma', —\}} \mathsf{skip}$ says that $\mathsf{assign}(\mathit{ref}, v)$ makes a transition to $\mathsf{skip}$ under the label where the readable label component **sto** is $\sigma$ and the writable label component **sto′** is $\sigma'$. It also says that no observable side effects occur in the remaining label components. Label composition in MSOS propagates the written $\sigma'$ entity to the **sto** label component in the next transition. The following consecutive steps illustrate this propagation:

$$\mathsf{seq}(\mathsf{assign}(\mathit{ref}, v), \mathsf{skip}) \xrightarrow{\{\mathbf{sto}=\sigma, \mathbf{sto'}=\sigma', —\}} \mathsf{seq}(\mathsf{skip}, \mathsf{skip}) \xrightarrow{\{\mathbf{sto}=\sigma', \mathbf{sto'}=\sigma', —\}} \mathsf{skip}$$

The second transition has $\sigma'$ in both **sto** and **sto′**; i.e., no unobservable side effects occur on the **sto**, **sto′** label components. Since no observable side effects occur in the second label, it could alternatively be written as $\{—\}$.

## 3 Generating MSOS Interpreters

This section describes how the Prolog MSOS Tool synthesizes interpreters in Prolog from MSOS specifications.

### 3.1 From MSOS Rule to Prolog Clause

MSOS terms are compiled as summarized in Table 1. Table 2 shows the compiled Prolog clauses for the seq construct. Solving a goal `step(_,_,_)` in Prolog using the compiled clauses corresponds to checking that the step is valid relative to the MSOS rules. Using the clauses in Table 2, we can check that the term $\mathsf{seq}(\mathsf{seq}(\mathsf{skip}, \mathsf{skip}), \mathsf{skip})$ can make a transition step:

```
?- init_label(L), step(seq(seq(v(skip),v(skip)),v(skip)), L, X).
L = [env=map_empty, sto=map_empty, sto+=map_empty, out+=[]],
X = seq(v(skip), v(skip))
```

Here, `init_label` initializes MSOS labels with initial label components; in this case, `env=map_empty`, `sto=map_empty`, `sto+=Sigma_`, and `out+=Out`. Solving this goal executes the second Prolog clause in Table 2 which by `label_instance(L,unobs)` unifies `sto=map_empty` with `sto+=Sigma_`, and `out+=Out` with the unobservable output `out+=[]`.

### 3.2 Implementing the Transitive Closure in Prolog

The `steps` predicate[4] generates the transitive closure of the transition relation:

[4] This predicate is not tail-recursive. It is, however, possible to construct a tail-recursive version: `post_comp` accumulates sequences of emitted write-only data. If this data were to be emitted as it is generated, the call to `post_comp` could be removed.

| | **MSOS term** | **Prolog predicate** |
|---|---|---|
| Rule | $\left[\!\!\left[ \dfrac{C_1 \quad \cdots \quad C_n}{\gamma \xrightarrow{L} \gamma'} \right]\!\!\right]$ | `step(`$[\![\gamma]\!]$`,L,`$[\![\gamma']\!]$`) :-`<br>`label_instance(L,`$[\![L]\!]$`),`<br>$[\![C_1]\!], \cdots, [\![C_n]\!]$. |
| Transition step | $\left[\!\!\left[ \gamma \xrightarrow{L} \gamma' \right]\!\!\right]$ | `step(`$[\![\gamma]\!], [\![L]\!], [\![\gamma']\!]$`)` |
| Readable label | $[\![\{\mathbf{ix} = E, X\}]\!]$ | `[`$[\![\mathbf{ix}]\!]$`=`$[\![E]\!]$`|`$[\![X]\!]$`]` |
| Writable label | $[\![\{\mathbf{ix}' = E, X\}]\!]$ | `[`$[\![\mathbf{ix}]\!]$`+=`$[\![E]\!]$`|`$[\![X]\!]$`]` |
| Unobservable label | $[\![\text{—}]\!]$ | `unobs` |
| Map (e.g., $\rho, \sigma, \ldots$) | $[\![\ [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\ ]\!]$ | `[`$[\![x_1]\!]$`+>`$[\![v_1]\!], \ldots, [\![x_n]\!]$`+>`$[\![v_n]\!]$`]` |
| Terms, values, and label indices | $[\![t]\!]$ | Prolog atoms, annotated with `v(_)` for values. |
| Variables | $[\![x]\!]$; $[\![x_1]\!]$; $[\![x']\!]$ | `X`; `X1`; `X_` |

Table 1: Compilation of MSOS terms into Prolog predicates

| **MSOS rule** | **Prolog clause** |
|---|---|
| $\dfrac{e_1 \xrightarrow{\{\ldots\}} e_1'}{\mathsf{seq}(e_1, e_2) \xrightarrow{\{\ldots\}} \mathsf{seq}(e_1', e_2)}$ | `step(seq(E1,E2),L,seq(E1_,E2)) :-`<br>`label_instance(L,Dots),`<br>`step(E1,Dots,E1_).` |
| $\mathsf{seq}(\mathsf{skip}, e_2) \xrightarrow{\{\text{—}\}} e_2$ | `step(seq(v(skip),E2),L,E2) :-`<br>`label_instance(L,unobs).` |

Table 2: Compiled Prolog clauses for the seq construct

```
steps(T1,L,T3) :-
        pre_comp(L,L1), step(T1,L1,T2), mid_comp(L1,L2),
        steps(T2,L2,T3), post_comp(L1,L2,L).
steps(v(V),L,v(V)) :-
        label_instance(L,unobs).
```

These clauses are mutually exclusive; i.e., values are final terms for which no further transition is possible. `pre_comp`, `mid_comp`, and `post_comp` propagate readable and writable label components as described in Sect. 2.2.

```
?- init_label(L), steps(seq(seq(v(skip),v(skip)),v(skip)), L, X).
L = [env=map_empty, sto=map_empty, sto+=map_empty, out+=[]],
X = v(skip)
```

Prolog fails if no sequence of steps exists that yields a value:

```
?- init_label(L), steps(seq(seq(v(0),v(skip)),v(skip)), L, X).
false.
```

Figure 2 summarizes the number of inferences required for interpreters generated by the Prolog MSOS Tool to reduce terms of the structure[5]:

$$\underbrace{\mathsf{seq}(\mathsf{seq}(\cdots \mathsf{seq}}_{n}(\mathsf{skip},\mathsf{skip})\cdots,\mathsf{skip}),\mathsf{skip})$$

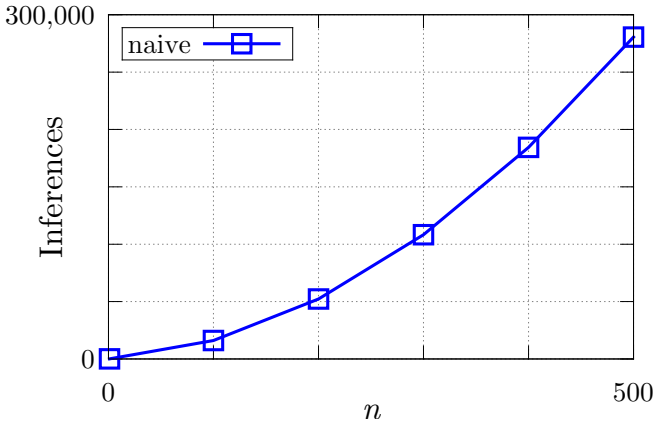


Fig. 2: Naïve evaluation of deeply nested seq terms

Since each step occurs on the outermost program term, the Prolog interpreter traverses the term in its full depth in each step, i.e., each step uses $O(n)$ inferences. It takes $n$ steps to evaluate a seq term of depth $n$, hence evaluation of deeply nested seq terms uses $O(n^2)$ inferences. We next demonstrate how *refocusing* reduces the number of required inferences to $O(n)$.

## 4 Refocused MSOS Interpreters

The transitive closure implemented by the `steps` predicate in Prolog is straightforwardly internalized in MSOS by the $\rightarrow^*$ relation defined by the rules:

$$\frac{x \xrightarrow{L_1} y \qquad y \xrightarrow{L_2}{}^* z}{x \xrightarrow{L_2 \circ L_1}{}^* z}\ [\textsc{trans}] \qquad \frac{}{v \xrightarrow{\{-\}}{}^* v}\ [\textsc{refl-v}]$$

Evaluating a term $s$ using these rules proceeds by constructing an upwardly branching derivation tree, if one exists, from the root formula $s \xrightarrow{L}{}^* t$. Using

[5] Right-nested seq terms do not suffer from runtime overhead. This is not the case, however, for deeply right-nested arithmetic expressions or $\lambda$-applications. We use left-nested seq terms here for simplicity of exposition.

$\Gamma, \Delta, \ldots$ to refer to instances of $\rightarrow^*$ and $A, B, \ldots$ to refer to instances of $\rightarrow$, derivation trees have the structure:

$$\dfrac{\dfrac{\vdots}{A} \quad \dfrac{\dfrac{\vdots}{B} \quad \dfrac{\dfrac{\vdots}{C} \quad \ddots}{\Psi}}{\Delta}}{\Gamma}$$

In generated Prolog interpreters this corresponds to traversing the entire program term in each intermediate step. Ideally, we want to construct as few derivation trees, and have as few traversals of intermediate program terms, as possible; i.e., we want to evaluate sub-terms as they are encountered. Augmenting our rules by the following *refocusing rule* permits exactly this:

$$\dfrac{x \xrightarrow{L_1} y \qquad y \xrightarrow{L_2}{}^* z}{x \xrightarrow{L_2 \circ L_1} z}\ [\textsc{refocus}]$$

The *refocusing transformation* forces evaluation of sub-terms by specializing the refocusing rule wrt an MSOS specification. The resulting set of *refocused rules* replace the original set of rules. The refocusing transformation unfolds the leftmost premise of [REFOCUS] wrt all rules in an MSOS specification:

$$\dfrac{\dfrac{\dfrac{C \quad D}{B}\ [d_1] \quad \Gamma}{A}\ [\textsc{refocus}]}{} \quad \Longrightarrow \quad \dfrac{C \quad D \quad \Gamma}{A}\ [d_1\text{-}\textsc{refocus}]$$

Using refocused rules changes the structure of derivation trees:

$$\dfrac{\dfrac{\dfrac{\vdots \quad \vdots}{B} \quad \dfrac{C \quad \vdots}{\Gamma}}{A} \quad \Psi}{\Delta}$$

For example, refocusing [SEQ1] (from Fig. 1, page 5) gives:

$$\dfrac{\dfrac{e_1 \xrightarrow{L_1} e_1'}{\mathsf{seq}(e_1, e_2) \xrightarrow{L_1} \mathsf{seq}(e_1', e_2)}\ [\textsc{seq1}] \qquad \mathsf{seq}(e_1', e_2) \xrightarrow{L_2}{}^* z}{\mathsf{seq}(e_1, e_2) \xrightarrow{L_2 \circ L_1} z}\ [\textsc{refocus}]$$

$$\Longrightarrow \quad \dfrac{e_1 \xrightarrow{L_1} e_1' \qquad \mathsf{seq}(e_1', e_2) \xrightarrow{L_2}{}^* z}{\mathsf{seq}(e_1, e_2) \xrightarrow{L_2 \circ L_1} z}\ [\textsc{seq1-refocus}]$$

Unfolding [SEQ2] and applying the [REFL-V] rule trivially gives an identical rule. Thus the refocused rules for $\mathsf{seq}$ are:

$$\dfrac{e_1 \xrightarrow{L_1} e_1' \qquad \mathsf{seq}(e_1', e_2) \xrightarrow{L_2}{}^* z}{\mathsf{seq}(e_1, e_2) \xrightarrow{L_2 \circ L_1} z}\ [\textsc{seq1-refocus}] \qquad \dfrac{}{\mathsf{seq}(\mathsf{skip}, e_2) \xrightarrow{\{-\}} e_2}\ [\textsc{seq2-refocus}]$$

Figure 3 summarizes the number of inferences the interpreter generated from the refocused MSOS specification uses to evaluate deeply nested seq terms. In contrast to naïve evaluation, the number of inferences increases linearly, since each sub-term is reduced when it is first encountered: evaluation uses $O(n)$ inferences.

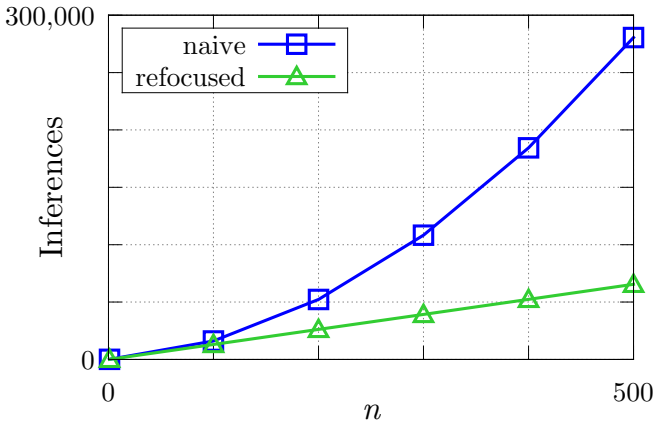


Fig. 3: Refocused and naïve evaluation of deeply nested seq terms

Introducing the refocusing rule permits sub-terms to be evaluated locally in derivations. Specializing the refocusing rule wrt an MSOS specification produces a specialized interpreter which forces evaluation of all sub-terms. However, forcing evaluation of sub-terms is not semantically sound in the presence of abrupt termination.

### 4.1 Refocusing and Abrupt Termination

Consider the language given by the following add, blocking, block, and loop constructs, where $+_i$ is integer addition, and $\mathbf{block}'$ is a write-only label component:

$$\frac{}{\mathsf{block} \xrightarrow{\{\mathbf{block}'=1,\_\}} \mathsf{stuck}}\ [\textsc{block}] \qquad \frac{e \xrightarrow{\{\mathbf{block}'=1,\ldots\}} e'}{\mathsf{blocking}(e) \xrightarrow{\{\mathbf{block}'=0,\ldots\}} \mathsf{skip}}\ [\textsc{blocking1}]$$

$$\frac{e \xrightarrow{\{\mathbf{block}'=0,\ldots\}} e'}{\mathsf{blocking}(e) \xrightarrow{\{\mathbf{block}'=0,\ldots\}} \mathsf{blocking}(e')}\ [\textsc{blocking2}] \qquad \frac{}{\mathsf{loop} \xrightarrow{\{\_\}} \mathsf{loop}}\ [\textsc{loop}]$$

$$\frac{}{\mathsf{blocking}(v) \xrightarrow{\{\_\}} v}\ [\textsc{blocking3}] \qquad \frac{v = v_1 +_i v_2}{\mathsf{add}(v_1, v_2) \xrightarrow{\{\_\}} v}\ [\textsc{add1}]$$

$$\frac{e_1 \xrightarrow{\{\ldots\}} e_1'}{\mathsf{add}(e_1, e_2) \xrightarrow{\{\ldots\}} \mathsf{add}(e_1', e_2)}\ [\textsc{add2}] \qquad \frac{e_2 \xrightarrow{\{\ldots\}} e_2'}{\mathsf{add}(e_1, e_2) \xrightarrow{\{\ldots\}} \mathsf{add}(e_1, e_2')}\ [\textsc{add3}]$$

If a block term is evaluated inside a blocking term, evaluation terminates and produces the value skip. Evaluating the loop-construct results in divergence.

Under ordinary small-step evaluation of the term blocking(add(block, loop)) we have the two possible outcomes of evaluation: either evaluation terminates with the value skip, or it diverges. If block is evaluated, the $\mathbf{block}' = 1$ label component is matched by the [BLOCKING1] rule for the blocking term, which terminates the program with value skip. Otherwise, the sub-term loop is evaluated, which results in a program term identical to the initial program.

Refocused evaluation, on the other hand, always diverges: evaluating block gives the term add(stuck, loop). This term has an evaluable sub-term, namely loop. Refocused evaluation forces evaluation of this term, resulting in divergence. In other words, adding the refocusing rule to a semantics with abrupt termination is not correct by default.

The issue of dealing with abrupt termination is symptomatic for big-step rules. In the presence of abrupt termination, one typically needs extra rules propagating the abruptly terminated term [4]. We show how to circumvent the problem with abrupt termination in refocused and big-step MSOS rules in a generic way: we introduce a special read-write label component, labeled by $\varepsilon$ and $\varepsilon'$, representing a flag indicating abrupt termination.

First, we add a single reflexive rule that propagates abruptly terminated configurations[6] ($\varepsilon = 1$), and update our existing evaluation rules to indicate that they apply only to configurations that are not abruptly terminated ($\varepsilon = 0$):

$$\frac{x \xrightarrow{\{\varepsilon=0, X_1\}} y \qquad y \xrightarrow{L_2}{}^{*} z}{x \xrightarrow{L_2 \circ \{\varepsilon=0, X_1\}}{}^{*} z}\ [\text{TRANS-}\varepsilon] \qquad \frac{x \xrightarrow{\{\varepsilon=0, X_1\}} y \qquad y \xrightarrow{L_2}{}^{*} z}{x \xrightarrow{L_2 \circ \{\varepsilon=0, X_1\}} z}\ [\text{REFOCUS-}\varepsilon]$$

$$\frac{}{v \xrightarrow{\{\varepsilon=0, -\}}{}^{*} v}\ [\text{REFL-V-}\varepsilon] \qquad \frac{}{x \xrightarrow{\{\varepsilon=1, -\}}{}^{*} x}\ [\text{REFL-}\varepsilon]$$

Second, MSOS specifications must explicitly indicate abrupt termination in rules. For example, rules that are sensitive to the behaviour of their sub-terms, such as [BLOCKING1] which inspects the writable $\mathbf{block}'$ component during evaluation of its sub-term, must explicitly indicate abruptly terminating steps via $\varepsilon, \varepsilon'$:

$$\frac{}{\mathsf{block} \xrightarrow{\{\mathbf{block}'=1, \varepsilon'=1, -\}} \mathsf{stuck}}\ [\text{BLOCK-}\varepsilon] \qquad \frac{e \xrightarrow{\{\mathbf{block}'=1, \varepsilon=0, \varepsilon'=1, \ldots\}} e'}{\mathsf{blocking}(e) \xrightarrow{\{\mathbf{block}'=0, \varepsilon=0, \varepsilon'=0, \ldots\}} \mathsf{skip}}\ [\text{BLOCKING1-}\varepsilon]$$

Using this alternative set of rules, refocused evaluation has the same possible outcomes for the example term blocking(add(block, loop)) as small-step evaluation.

Refocusing is a simple specialization which significantly reduces overhead compared to traditional small-step MSOS rules. However, it requires explicit specification of abrupt termination and of rules for constructs whose behaviour

[6] We refer to configurations as being *abruptly terminated* rather than *stuck*, since the terms in the configuration may have computational behaviour. E.g., the add(stuck, loop) term is not stuck in a strict sense, since it has evaluable sub-terms.

is sensitive to the behaviour of their sub-terms. It is ongoing work to identify syntactic constraints which uniquely distinguish abruptly terminating constructs and constructs that are sensitive to the number of steps their sub-terms can make. Such constraints would enable automatic insertion of $\varepsilon, \varepsilon'$ label components.

## 5 Big-Step MSOS Interpreters

A small-step transition relation relates terms to other partly evaluated terms. Under refocused evaluation, the transition relation relates terms directly to values or abruptly terminated terms. Refocused rules are therefore in big-step style. However, refocused rules may use several intermediate inferences to map a term to a value. The *striding transformation* specializes refocused rules to remove the extra overhead. The resulting rules are similar to classic big-step rules.

### 5.1 The Striding Transformation

The striding transformation has the effect of compressing 'corridor' transitions [8], i.e., transitions for which a unique further transition exists. The striding transformation specializes a refocused rule, [$d_1$-REFOCUS], wrt another refocused rule, [$d_2$-REFOCUS]. The result is a big-step style rule, [$d_1$-$d_2$-STRIDING]:

$$\dfrac{C \qquad D \qquad \dfrac{E \qquad \Delta}{\Gamma}\,[d_2\text{-REFOCUS}]}{A}\,[d_1\text{-REFOCUS}] \implies \dfrac{C \qquad D \qquad E \qquad \Delta}{A}\,[d_1\text{-}d_2\text{-STRIDING}]$$

The striding transformation generates the set of all possible combinations of rule unfoldings. To filter semantically equivalent rules resulting from the transformation, we use *formal hypothesis simulation* (*fh-simulation*) [18]. For example, specializing the [SEQ1-REFOCUS] rule wrt itself gives:

$$\dfrac{e_1 \xrightarrow{L_1} e_1' \qquad e_1' \xrightarrow{L_2} e_1'' \qquad \mathsf{seq}(e_1'', e_2) \xrightarrow{L_3}{}^{*}\ z}{\mathsf{seq}(e_1, e_2) \xrightarrow{L_3 \circ L_2 \circ L_1} z}\,[\text{SEQ1-SEQ1-STRIDING}]$$

However, every possible step this rule can make can be matched by [SEQ1-REFOCUS]. Hence, we omit this rule from the set of striding rules. Specializing [SEQ1-REFOCUS] wrt [SEQ2-REFOCUS] gives the rule:

$$\dfrac{e_1 \xrightarrow{\{\ldots\}} \mathsf{skip}}{\mathsf{seq}(e_1, e_2) \xrightarrow{\{\ldots\}} e_2}\,[\text{SEQ1-SEQ2-STRIDING}]$$

By the MSOS rules for the seq construct, substituting the $\rightarrow$ with $\rightarrow^*$ is equivalent:

$$\dfrac{e_1 \xrightarrow{\{\ldots\}}{}^{*}\ \mathsf{skip}}{\mathsf{seq}(e_1, e_2) \xrightarrow{\{\ldots\}} e_2}\,[\text{SEQ1-SEQ2-STRIDING}^*]$$

This rule matches all steps that can be made using the [SEQ2] rule. There are no rules which can match all possible steps that the [SEQ1-SEQ2-STRIDING*] rule can make. Any subsequent rule unfoldings can be shown to be equivalent to the current set of rules by fh-similarity. Thus the set of rules resulting from applying the striding transformation to the seq construct are:

$$\frac{e_1 \xrightarrow{\{\ldots\}}{}^{*} \mathsf{skip}}{\mathsf{seq}(e_1, e_2) \xrightarrow{\{\ldots\}} e_2}\,[\text{SEQ1-SEQ2-STRIDING*}] \qquad \frac{e_1 \xrightarrow{L_1} e_1' \quad \mathsf{seq}(e_1', e_2) \xrightarrow{L_2}{}^{*} z}{\mathsf{seq}(e_1, e_2) \xrightarrow{L_2 \circ L_1} z}\,[\text{SEQ1-REFOCUS}]$$

### 5.2 Left-Factoring

The [SEQ1-REFOCUS] rule relates a seq term to a result, which is characteristic of big-step rules. While big-step derivation trees contain fewer inferences, compiled big-step Prolog clauses potentially give rise to non-determinism and back-tracking during proof search. For example, the conclusions of both [SEQ1-SEQ2-STRIDING*] and [SEQ1-REFOCUS] match arbitrary seq terms. In the worst case, this non-determinism leads to back-tracking, which would increase the number of inferences required to evaluate terms that do not yield values.

Left-factoring [1,20] is a simple clause transformation which improves the determinacy of Prolog clauses generated from big-step style rules:

$$\begin{array}{l} H \leftarrow A \wedge B \\ H \leftarrow A \wedge C \end{array} \quad \Longrightarrow \quad H \leftarrow A \wedge (B \vee C)$$

Using this simple idea, Prolog clauses are transformed to obtain specialized interpreters without the back-tracking penalty incurred by compiling big-step style rules into Prolog clauses. Figure 4 summarizes the reduction in the number of inferences resulting from striding and left-factoring when evaluating deeply nested seq terms.

## 6 Benchmark Experiments

We assess the viability of the specializations proposed in previous sections by considering a variant of a larger MSOS example semantics [5] with function closures and imperative state.

Figure 5 summarizes the number of Prolog inferences used to calculate the factorial of $n$, the $n$th Fibonacci number, and the greatest common divisor of the $n$th and $n+1$st Fibonacci numbers using Euclid's algorithm. Each program is implemented[7] in two ways: applicatively, based on recursive unfolding; and imperatively, based on assignment and a while loop construct.

The refocusing rule introduces extra label composition operations in generated Prolog clauses for refocused rules. For deeply nested program terms this saves

[7] Benchmark code, generated interpreters, and details about the Prolog system used are available online: `http://cs.swansea.ac.uk/~cscbp/lopstr13`

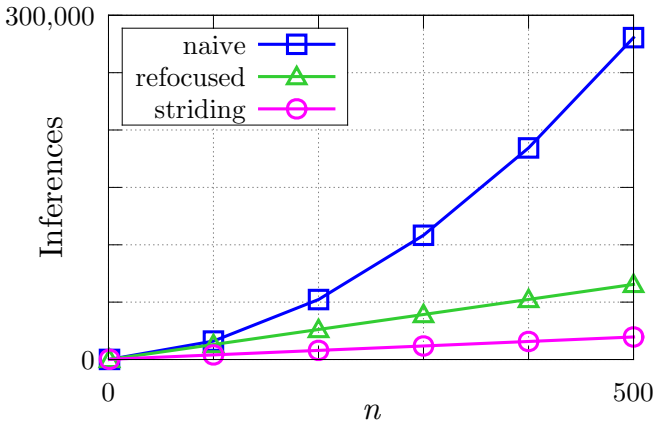


Fig. 4: Striding, refocused, and naïve evaluation of deeply nested seq terms

having to re-traverse the term in the next step. However, it entails redundant computations for values. This explains both the encouraging speed-ups in the applicative benchmarks (which unfold function closures to form deeply nested terms), and the slight overhead that refocusing and striding introduces for shallowly nested terms, such as the imperative factorial and Fibonacci benchmarks.

We emphasize that our specialization significantly reduces overhead in 4 out of 6 benchmarks, where the number of inferences is reduced by 4 times or more. Evaluating shallowly nested terms using big-step rules compared to small-step entails a relatively modest overhead of around 1.3 times more inferences.

## 7 Conclusion and Further Work

We have described how to generate interpreters from MSOS specifications and how such interpreters can be encoded in Prolog. After assessing the overhead of interpreters generated from small-step rules, we applied refocusing and striding to derive their big-step counterparts. The resulting generated interpreters significantly reduced the number of inferences used to evaluate deeply nested program terms.

Label composition is computationally expensive in generated interpreters as Fig. 5 illustrates. Our label composition strategy alleviates the need to re-compile rules as new constructs are added to languages, but requires us to traverse the Prolog list representation of label components multiple times (in the worst case) in each step. One could use a partial evaluator, such as Logen [12], to unfold label composition predicates. This would correspond to compiling an MSOS specification into an SOS specification, similar to compiling generalized transition systems (underlying MSOS) to labelled transition systems (underlying SOS), as

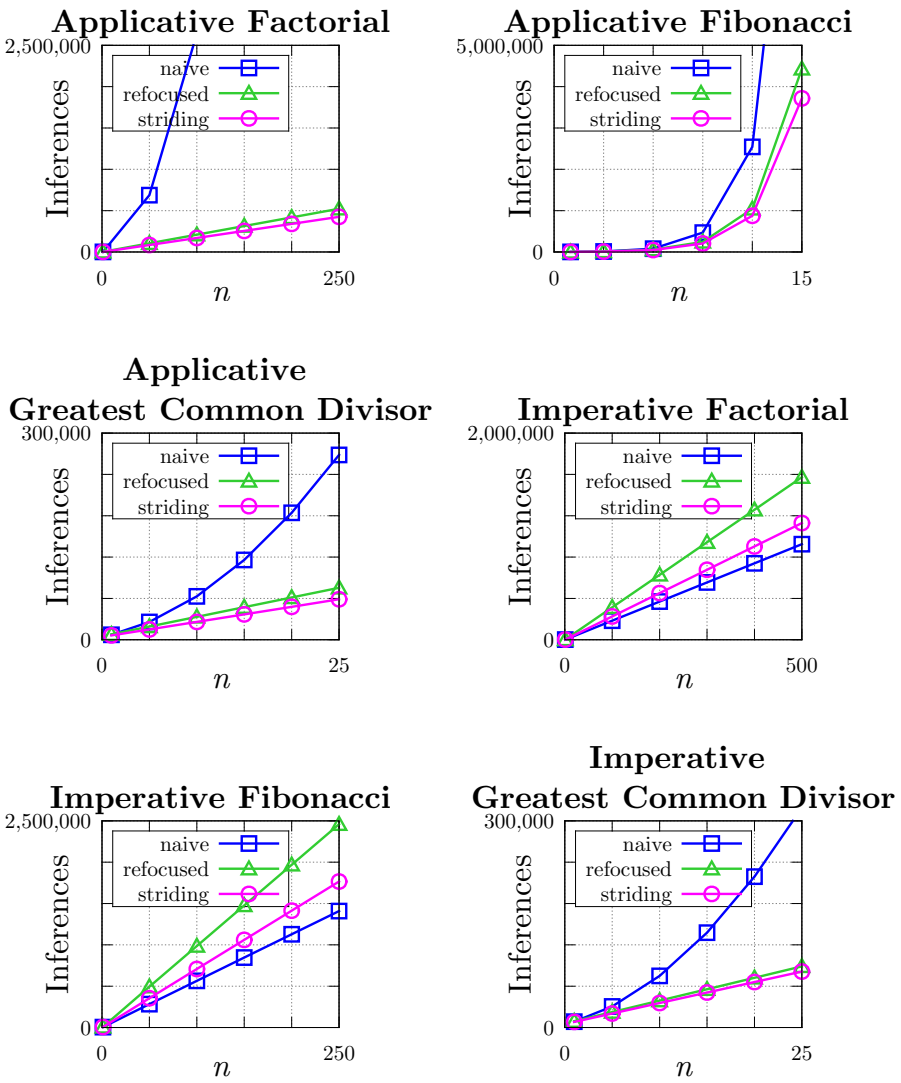


Fig. 5: Benchmark inference graphs

described in [16]. Unfolding label composition predicates in generated Prolog interpreters should decrease the number of inferences required to evaluate terms.

The refocusing rule requires MSOS rules to be explicit about abruptly terminating constructs and constructs that are sensitive to the number of steps their sub-terms make. It should be possible to specify a rule format for conserva-

tively identifying abruptly terminating constructs. This would enable automatic annotation of MSOS rules with $\varepsilon, \varepsilon'$ label components.

Striding requires filtering specialized rules that are equivalent to existing ones. We suggested using fh-simulation [18] for this. For the purposes of this paper, these proofs were constructed manually. While bisimulation is undecidable in general, it should be possible to automate proofs for at least some constructs.

*Acknowledgements.* Thanks to Paolo Torrini, Martin Churchill, Ferdinand Vesely, and the anonymous referees for their useful comments. This work was supported by an EPSRC grant (EP/I032495/1) to Swansea University in connection with the *PLanCompS* project (`www.plancomps.org`).

## References

1. Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling. Prentice-Hall, Inc. (1972)
2. Bach Poulsen, C., Mosses, P.D.: Deriving pretty-big-step semantics from small-step semantics. In: Shao, Z. (ed.) ESOP'14. LNCS, vol. 8410, pp. 270–289. Springer Berlin Heidelberg (2014)
3. Chalub, F., Braga, C.: Maude MSOS tool. ENTCS 176(4), 133–146 (2007)
4. Charguéraud, A.: Pretty-big-step semantics. In: Felleisen, M., Gardner, P. (eds.) ESOP'13. LNCS, vol. 7792, pp. 41–60. Springer Berlin Heidelberg (2013)
5. Churchill, M., Mosses, P.D.: Modular bisimulation theory for computations and values. In: Pfenning, F. (ed.) FOSSACS'13. LNCS, vol. 7794, pp. 97–112. Springer Berlin Heidelberg (2013)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude manual (version 2.6) (2008), `http://maude.cs.uiuc.edu/maude2-manual/`
7. Clement, D., Despeyroux, J., Despeyroux, T., Hascoet, L., Kahn, G.: Natural semantics on the computer. Research Report RR-0416, INRIA (1985)
8. Danvy, O.: From reduction-based to reduction-free normalization. In: Koopman, P.W.M., Plasmeijer, R., Swierstra, S.D. (eds.) AFP'08. LNCS, vol. 5832, pp. 66–164. Springer Berlin Heidelberg (2008)
9. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. BRICS Research Series RS-04-26, Dept. of Computer Science, Aarhus University (2004)
10. Gallagher, J.P.: Tutorial on specialisation of logic programs. In: PEPM'93. pp. 88–98. ACM (1993)
11. Gupta, G.: Horn logic denotations and their applications. In: Apt, K.R., Marek, V.W., Truszczynski, M., Warren, D.S. (eds.) The Logic Programming Paradigm, pp. 127–159. Artificial Intelligence, Springer Berlin Heidelberg (1999)
12. Leuschel, M., Jørgensen, J., Vanhoof, W., Bruynooghe, M.: Offline specialisation in Prolog using a hand-written compiler generator. TPLP 4(1), 139–191 (2004)
13. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Log. Program. 11(34), 217–242 (1991)
14. Mosses, P.D.: Foundations of Modular SOS. BRICS Research Series RS-99-54, Dept. of Computer Science, Aarhus University (1999)
15. Mosses, P.D.: Pragmatics of Modular SOS. In: Kirchner, H., Ringeissen, C. (eds.) AMAST'02, LNCS, vol. 2422, pp. 21–40. Springer Berlin Heidelberg (2002)
16. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. 60-61, 195–228 (2004)

17. Mosses, P.D.: Teaching semantics of programming languages with Modular SOS. In: Boca, P., Bowen, J.P., Duce, D.A. (eds.) TFM'06. Electr. Workshops in Comput., BCS (2006)
18. Mosses, P.D., Mousavi, M.R., Reniers, M.A.: Robustness of equations under operational extensions. In: Fröschle, S.B., Valencia, F.D. (eds.) EXPRESS'10. EPTCS, vol. 41, pp. 106–120 (2010)
19. Mosses, P.D., New, M.J.: Implicit propagation in structural operational semantics. ENTCS 229(4), 49–66 (2009)
20. Pettersson, M.: Compiling Natural Semantics, LNCS, vol. 1549. Springer Berlin Heidelberg (1999)
21. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
22. Wang, Q., Gupta, G., Leuschel, M.: Towards provably correct code generation via Horn logical continuation semantics. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL'05, LNCS, vol. 3350, pp. 98–112. Springer Berlin Heidelberg (2005)