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 first recall how to synthesize executable interpreters from small-step MSOS specifications by compiling MSOS rules to Prolog clauses. Implementing the transitive closure of compiled clauses gives an executable interpreter in Prolog. Such interpreters, however, traverse each intermediate program term, resulting in a significant overhead in each step.
We then show how to transform small-step MSOS specifications into corresponding big-step specifications via a two-step specialization by internalizing and extending the rules implementing the transitive closure in MSOS. 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) [19] provides a simple and direct method for specifying the semantics of programming language constructs and process algebras. The behaviour of constructs defined in SOS is modelled by a labelled transition system where inference rules and axioms of the SOS specification define valid transition steps in the underlying transition system. The computations of an SOS specification are obtained by taking the transitive closure of the transition relation that the SOS specification defines.

SOS rules for programming languages define valid transition steps wrt 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 currently visible identifiers). In conventional SOS, auxiliary entities are explicit in all rules. This gives rise to the modularity problem in SOS: language extensions involving new auxiliary entities requires reformulating existing rules.

Modular SOS [14] solves the modularity problem in SOS by implicitly propagating all unmentioned auxiliary entities. As with SOS, the computations of an MSOS specification are obtained by taking the transitive closure of the transition relation that an MSOS specification inductively defines.

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 [3,16] or structural induction on the underlying MSOS rules [14], and prototype interpreter generation. In this paper we focus on generating prototype interpreters in Prolog.

Contribution. We show how to minimize overhead in MSOS interpreters. Compilation of MSOS rules into Prolog clauses has been utilized and hinted at in earlier publications (e.g., [3,13,14,15]). This work presents the first systematic account of how to synthesize executable interpreters in Prolog from MSOS specifications. It is also the first to assess and improve the efficiency of these interpreters.

The efficiency of generated interpreters is significantly enhanced by a simple generalization of the internalized closure of the transition relation. This is achieved by introducing a *refocusing rule*, in a similar style to Danvy and Nielsen's refocusing transformation for reduction semantics [7]. Specializing the refocusing rule wrt an MSOS specification forces evaluation of sub-terms and effectively disposes of computational overhead that previous interpreters generated from MSOS specifications [1,3,14,15] have suffered from.

Through a subsequent specialization step, called *striding*, a small-step specification is transformed into its corresponding big-step counterpart. *Left-factoring* [18] the resulting big-step specification results in generated interpreters whose efficiency is significantly better.

We demonstrate and illustrate our techniques on MSOS specifications due to the pragmatic advantages of MSOS over SOS, but we expect that the techniques can be straightforwardly applied to obtain more efficient interpreters for other operational semantics frameworks.

Related work. The Maude MSOS Tool [1] executes MSOS specifications encoded as rewriting logic rules in Maude [4]. It allows for elegant representation of MSOS rules utilizing many of Maude's features, such as sorts and records. However, the approach to interpreting MSOS specifications is essentially similar to that of the Prolog MSOS tool: steps occur at the top-level, resulting in a significant overhead in each step.

The refocusing rule that we introduce is inspired by Danvy et al.'s work on refocusing in reduction semantics [7]. Similarly, the striding transformation, which effectively transforms small-step MSOS rules into big-step MSOS rules, is

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

[2] In fact, funcons are specified using Implicitly Modular SOS [17], a variant of MSOS with syntax closer to SOS.

inspired by Danvy et al.'s work on inter-derivable small-step and big-step abstract machines [6]. That work is based on program transformations applied to functional programs implementing a reduction semantics, and requires advanced functional programming transformations. In contrast, the specialization we present here applies directly to MSOS rules, and is based on simple unfolding corresponding to partial evaluation in logic programming [8,11].

Prolog has been widely used as a vehicle for implementing and executing semantics, dating back to Kahn et al.'s TYPOL [5]. *Horn logical semantics* [9] even suggests using Horn logic clauses directly for semantic specification. However, in the context of component-based specification, Horn logical semantics have several drawbacks: extending a specification with new semantic domains requires modification of existing predicates. Furthermore, Horn logical semantics is based on denotational semantics and specified in a big-step style (i.e., predicates map terms to their values or denotations). The big-step style makes specification of control instructions challenging, as witnessed by Wang et al.'s suggestion of using Horn logical continuation-based semantics [20] to handle abrupt termination: in the continuation-based approach each predicate is parameterized over terms, semantic domains, control stacks, and continuations.

Extending small-step MSOS does not require reformulation of existing rules. It also straightforwardly supports jumps in a similar style to exception handling as described in [14]. This paper suggests using the small-step style for specification, and describes how to systematically derive the corresponding (more runtime efficient) big-step specification by specialization. While partial evaluation in logic programming [11] has been extensively studied as a means of compiling programs and speeding up interpreters based on binding time analyses, to the authors' knowledge, this is the first work to use partial evaluation in logic programming for transforming small-step style inference rules.

Refocused rules bear a striking resemblance to Charguéraud's pretty-big-step rules [2]. A more in-depth investigation of the similarities between refocused rules in MSOS and pretty-big-step semantics is left to future work.

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 specializes the refocusing rule wrt an MSOS specification. The efficiency of generated naive, refocused, and striding interpreters is assessed in Sect. 6. Section 7 concludes and suggests further lines of research.

2 Modular Structural Operational Semantics

The main features of Modular SOS are outlined and compared with SOS.

2.1 An Example SOS

Formally, SOS rules define admissible transition steps in an underlying labelled transition system. In SOS, a transition step from γ to γ' is admissible if: (1) it matches the *conclusion* 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, α 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 SOS rules with empty premises or satisfied side-conditions[3]. See [14,19] for a more detailed introduction to (M)SOS.

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

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

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

Consider instead the rules defining sequential execution, $\mathsf{seq}(e_1, e_2)$, variable assignment, $\mathsf{assign}(ref, e)$, and variable dereferencing, $\mathsf{deref}(ref)$. σ ranges over stores, ref over references, and skip is a value. The formula $\langle e, \sigma\rangle \rightarrow \langle e', \sigma'\rangle$ asserts that the configuration given by term e and store σ can make a transition to the configuration given by term e' and store σ'.

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

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

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

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 σ; likewise, seq, assign, and deref must propagate an environment ρ. We refrain from this tedious reformulation here. Instead, we switch to MSOS.

2.2 Modular SOS

Like in SOS, MSOS rules define admissible steps; i.e., a step is admissible relative to a set of rules if we can construct a derivation tree using them. In contrast to SOS, all auxiliary entities in MSOS are encoded in the label of the transition relation, and only explicitly mentioned when required. For example, the [LET1]

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

rule makes no explicit mention of auxiliary entities, since they are inessential for that rule:

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

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

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

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

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

Crucially, computations in MSOS requires labels on consecutive transitions to be *composable*. The remainder of this section defines MSOS labels and label composition.

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

Label variables *refer to an arbitrary number of label components. The label variable '—' ranges over* unobservable labels*. Other label variables '. . .', X, Y, etc. refer to arbitrary label components.*

Informally, a label is *observable* if it exhibits side effects. The label instance with store label components $\mathbf{sto}\!=\!\sigma, \mathbf{sto}'\!=\!\sigma'$ such that $\sigma \neq \sigma'$ in the [asn2] rule is an example of an observable label.

Another example of an observable label is illustrated by the print construct:

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

The **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 $\mathbf{out}'\!=\![\,]$.

Environments, stores, and output channels each exemplify a distinct category of label components. These categories define the information flow between consecutive transition labels (i.e., how labels compose). The interested reader is referred to the literature [12,14] for a more in-depth treatment of label composition in MSOS. For the purpose of this paper, the following definition of label composition suffices[4], where '$\circ$' is the label composition operation:

[4] Labels in MSOS are actually modelled by arrows in a category. The category gives the semantics of label composition, here considered as a partial operation on arrows.

– 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 skip under the label where readable label component **sto** is σ and writable label component $\mathbf{sto}'$ is σ'. It also says that no observable side effects occur on any other label components. Label composition in MSOS propagates the written σ' 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 σ' in both **sto** and $\mathbf{sto}'$; i.e., no unobservable side-effects occur on the $\mathbf{sto}, \mathbf{sto}'$ 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.

	MSOS term	**Prolog predicate**
Rule	$\left[\!\!\left[\dfrac{C_1 \quad \cdots \quad C_n}{\gamma \xrightarrow{L} \gamma'}\right]\!\!\right]$	$\texttt{step}([\![\gamma]\!], \texttt{L}, [\![\gamma']\!])$ `:-` $\texttt{label\_instance}(\texttt{L}, [\![L]\!])$, $[\![C_1]\!]$, $\vdots$ $[\![C_n]\!]$.
Transition step	$\left[\!\!\left[\gamma \xrightarrow{L} \gamma'\right]\!\!\right]$	$\texttt{step}([\![\gamma]\!], [\![L]\!], [\![\gamma']\!])$
Label	$[\![\{\mathbf{ix}=E, X\}]\!]$, $[\![\{\mathbf{ix}'=E, X\}]\!]$	$\texttt{[}[\![\mathbf{ix}]\!]\texttt{=}[\![E]\!]\mid[\![X]\!]\texttt{]}$, $\texttt{[}[\![\mathbf{ix}]\!]\texttt{+=}[\![E]\!]\mid[\![X]\!]\texttt{]}$
Unobservable label	$[\![—]\!]$	$\texttt{unobs(}[\![L]\!]\texttt{)}$
Map (e.g., $\rho, \sigma, \ldots$)	$[\![\ [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\ ]\!]$	$\texttt{[}\ [\![x_1]\!]\texttt{+>}[\![v_1]\!], \ldots, [\![x_n]\!]\texttt{+>}[\![v_n]\!]\texttt{]}$
Terms, values, and label indices	$[\![t]\!]$	Prolog atoms, annotated with `v(_)` for values.
Variables	$[\![x]\!]$	`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)) :-` `label_instance(L,Dots),` `step(E1,Dots,E1_).`
$\mathsf{seq}(\mathsf{skip}, e_2) \xrightarrow{\{-\}} e_2$	`step(seq(v(skip),E2),L,E2) :-` `label_instance(L,unobs(L)).`

Table 2: Compiled Prolog clauses for the seq construct

3.1 From MSOS Rule to Prolog Clause

Terms in MSOS 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 admissible in MSOS. Using the clauses in Table 2, we can check that the term $\mathsf{seq}(\mathsf{seq}(\mathsf{skip}, \mathsf{skip}), \mathsf{skip})$ has an admissible step and what the result is:

```
?- 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))
```

`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(L))` 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[5] in the Prolog MSOS Tool generates the transitive closure of the transition relation:

```
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(L)).
```

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 Subsect. 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)
```

[5] 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.

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

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

Figure 1 summarizes the number of inferences required for interpreters generated by the Prolog MSOS tool to reduce terms of the structure[6]:

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

Since each step occurs on the outermost program term, each intermediate term is traversed in its full depth, i.e., $O(n)$. Since we require n steps to reduce a seq term of depth n, deeply nested seq terms require $O(n^2)$ inferences to reduce. We next show that *refocusing* reduces the number of required inferences to $O(n)$.

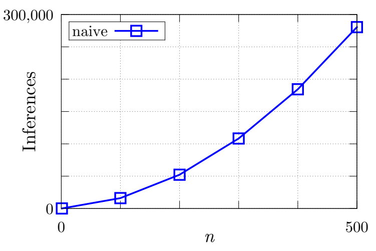

Fig. 1: Naive evaluation of deeply nested seq terms

4 Refocused MSOS Interpreters

The transitive closure implemented by the `steps` predicate in Prolog is straightforwardly internalized in MSOS by the following 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 a judgment $s \xrightarrow{L}{}^{*} t$, where L is an initial label. Using $\Gamma, \Delta, \ldots$ to refer to transitive steps ($\rightarrow^{*}$) and $A, B, \ldots$ to refer to ordinary transition steps ($\rightarrow$), derivation trees have the structure:

$$\cfrac{\cfrac{\vdots}{A} \qquad \cfrac{\cfrac{\vdots}{B} \qquad \cfrac{\cfrac{\vdots}{C} \qquad \cdot^{\cdot^{\cdot}}}{\Psi}}{\Delta}}{\Gamma}$$

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 reduce reducible expressions as we encounter them. Augmenting the set of evaluation rules by the following *refocusing rule* permits exactly this:

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

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

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. Figure 2 shows the transformation: the leftmost premise of [REFOCUS] is unfolded wrt all rules in an MSOS specification.

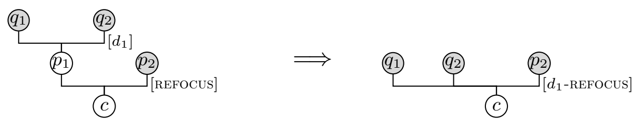

Fig. 2: The refocusing transformation

The refocusing transformation changes the structure of derivation trees:

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

Refocusing [SEQ1] gives:

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

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

Unfolding [SEQ2] and applying the [REFL-V] rule trivially gives an identical rule. The refocused rules for seq are:

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

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 naive evaluation, the number of inferences increases linearly, since each sub-term is reduced when it is first encountered. The number of inferences required to evaluate terms is reduced to $O(n)$ inferences.

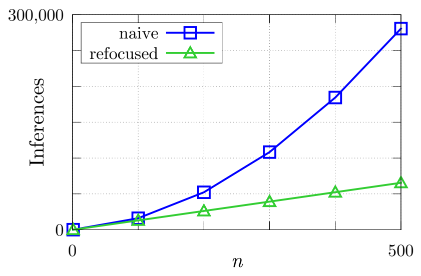

Fig. 3: Refocused and naive evaluation of deeply nested seq terms

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 add, $\mathsf{blocking}$, block, and loop constructs, whose semantics is given by the following rules, where $+_i$ is integer addition:

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

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

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

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

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

Under ordinary small-step evaluation of the term $\mathsf{blocking}(\mathsf{add}(\mathsf{block}, \mathsf{loop}))$ we have the two intended possible interpretations of the term: either evaluation terminates with the value skip; or it diverges. If the sub-term block is evaluated, the $\mathbf{block}' = 1$ label component is propagated to the outermost $\mathsf{blocking}$ 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 $\mathsf{add}(\mathsf{stuck}, \mathsf{loop})$. 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. Handling abrupt termination in big-step rules poses a similar challenge: in the presence of abrupt termination, one needs extra rules propagating the abruptly terminated term, requiring extra rules that clutter the semantics.

We show how to circumvent the problem of abrupt termination in refocused and big-step MSOS rules in a generic way: we introduce a special read-write label component, $\varepsilon, \varepsilon'$, which represents a flag indicating abrupt termination.

First, we add a single reflexive rule that propagates abruptly terminated terms[7] $(\varepsilon = 1)$, and update our existing evaluation rules to indicate that they apply only to terms that are not abruptly terminated $(\varepsilon = 0)$:

$$\frac{x \xrightarrow{\{\varepsilon=0,X_1\}} y \qquad y \xrightarrow{L_2}{}^{*}\ z}{x \xrightarrow{L_2\circ\{\varepsilon=0,X_1\}}{}^{*}\ z}\ [\textsc{trans-}\varepsilon] \qquad \frac{x \xrightarrow{\{\varepsilon=0,X_1\}} y \qquad y \xrightarrow{L_2}{}^{*}\ z}{x \xrightarrow{L_2\circ\{\varepsilon=0,X_1\}} z}\ [\textsc{refocus-}\varepsilon]$$

$$\frac{}{v \xrightarrow{\{\varepsilon=0,—\}}{}^{*}\ v}\ [\textsc{refl-v-}\varepsilon] \qquad \frac{}{x \xrightarrow{\{\varepsilon=1,—\}}{}^{*}\ x}\ [\textsc{refl-v-}\varepsilon]$$

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

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

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

Using this alternative set of rules, refocused evaluation has the same possible outcomes for the example term as ordinary evaluation.

In summary, the refocusing rule is a simple extension of the evaluation rules for MSOS, which significantly reduces overhead. However, it requires explicit annotation of abrupt termination and of rules for constructs whose behaviour varies depending on the behaviour of their sub-terms. It is ongoing work to identify syntactic constraints which uniquely distinguish abruptly terminating constructs and sub-term behaviour-sensitive contexts. Such constraints would enable automatic annotation with ε label components of abruptly terminating and step sensitive rules.

5 Big-Step Style MSOS Interpreters

Under naive evaluation, the transition relation maps terms to terms. Under refocused evaluation, the transition relation map terms to values or abruptly terminated terms. Strictly speaking, 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

For each construct in a refocused specification, the striding transformation works by specializing each rule wrt the set of rules for that construct, filtering semantically equivalent rules.

Figure 4 visualizes the striding transformation. A refocused rule, [d_1-REFOCUS], is specialized wrt a second refocused rule, [d_2-REFOCUS]. The result is a big-step style striding rule, [d_1-d_2-STRIDING].

The striding transformation in Fig. 4 generates the set of all possible combinations of rule applications. However, we are only interested in the set of all *unique* combinations. To ensure uniqueness, and to ensure termination of unfolding, rules are filtered by *formal hypothesis simulation* (*fh-simulation*) [16].

Specializing the [SEQ1-REFOCUS] rule wrt itself gives:

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

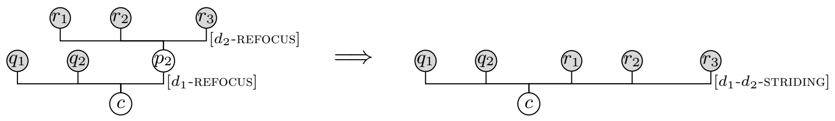

Fig. 4: The striding transformation

However, every possible step admissible by this rule can be matched by [SEQ1-REFOCUS]. It is possible to prove this by an fh-simulation proof. Hence, we omit this rule from the set of striding rules. Specializing [SEQ1-REFOCUS] wrt [SEQ2-REFOCUS] gives the rule:

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

By the MSOS rules for the seq construct, substituting the ordinary transition relation ($\rightarrow$) with the transitive relation ($\rightarrow^*$) is semantically equivalent:

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

This rule matches all steps that can be made using the [SEQ2] rule. There are no rules which can match all possible admissible steps by the [SEQ1-SEQ2-STRIDING*] rule. 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-SEQ2-STRIDING] rule maps terms directly to a value, which is characteristic of big-step style rules.While the big-step style derivations require fewer inferences, they also increase non-determinism. E.g., the conclusions of both [SEQ1-SEQ2-STRIDING*] and [SEQ1-REFOCUS] match arbitrary seq terms. Under Prolog's depth-first proof search strategy, this non-determinism leads to back-tracking, thereby *increasing* the number of inferences required to evaluate terms that do not yield values.

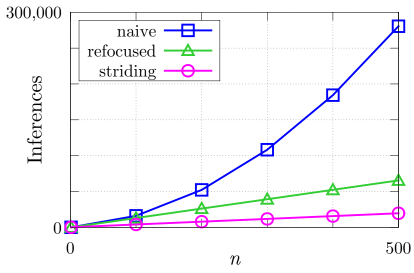

Fig. 5: Striding, refocused, and naive evaluation of deeply nested seq terms

Left-factoring, used by, e.g., Pettersson [18], 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 encoding big-step style rules directly in Prolog. Figure 5 summarizes the reduction in the number of inferences for deeply nested `seq` terms resulting from striding and left-factoring.

6 Benchmark Experiments

We assess the viability of the specializations proposed in previous sections by considering a variant of a larger previously published MSOS example semantics [3] with function closures and applicative state incorporating many of the constructs considered throughout this paper.

Figure 6 summarizes the number of logic inferences used in Prolog to calculate the factorial of n, the nth Fibonacci number, and the greatest common divisor of the nth and $n + 1$st Fibonacci numbers using Euclid's algorithm. Each program is implemented[8] in two ways: applicatively, based on recursive unfolding; and imperatively, based on assignment and a `while` loop construct.

For small n, almost all the programs in Figure 6 use more inferences than naive evaluation. Under naive evaluation, label composition occurs only at the top-level. In contrast, the refocusing rule introduces label composition at each inference step. For nested computational terms this saves having to re-traverse the term in the next step. However, it entails redundant computations for value terms. 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 short and shallowly nested programs, such as the imperative factorial and Fibonacci benchmarks.

We emphasize that our specialized interpreters have significantly reduced overhead in 4 out of 6 benchmarks, where the number of inferences is reduced by 4 times or more. The overhead of evaluating shallowly nested terms using striding rules compared to naive evaluation is relatively modest and requires around 1.3 times more inferences.

7 Concluding Remarks

We have described how to generate interpreters from MSOS specifications and how such interpreters can be encoded in Prolog. Assessing the overhead of naively constructed interpreters, we suggested introducing the refocusing rule which allows transitive steps to occur anywhere ordinary transition steps can occur. Specializing the refocusing rule wrt an MSOS specification results in a specialized interpreter whose rules are in big-step style.

Our benchmarks show that the refocusing rule significantly reduces overhead for deeply nested terms. For shallowly nested terms, refocusing introduces a modest overhead, due to redundant label composition. For both deeply and shallowly nested terms, striding reduces overhead.

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

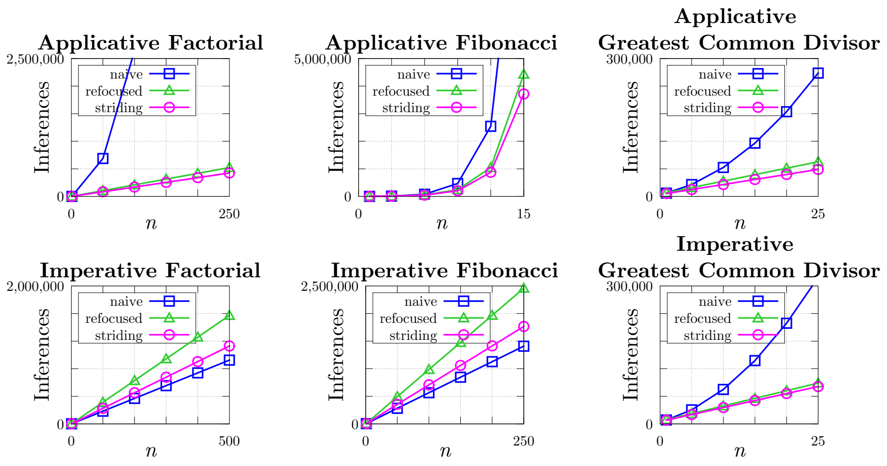

Fig. 6: Benchmark inference graphs

Further work. As the benchmark experiments summarized in Figure 6 show, label composition is a major culprit in terms of the number of inferences. The Prolog auxiliary label composition predicates enable us to generate components that can be composed without requiring any recompilation. However, they also require the Prolog list representing the label components to be traversed multiple times in each step. For complete languages, where we do not need the flexibility of being able to extend the language with further label components, it should be possible to use a partial evaluator, such as LOGEN [10], to unfold these auxiliary predicates. This would correspond to compiling an MSOS specification into an SOS specification, similarly to compiling generalized transition systems (underlying MSOS) to labelled transition systems (underlying SOS), as described in [14]. Unfolding the auxiliary label composition predicates in generated Prolog interpreters should decrease the number of inferences required to reduce 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 which (conservatively) identifies non-abruptly terminating constructs. Such a rule format would allow automatically annotating an MSOS semantics with the necessary $\varepsilon, \varepsilon'$ label components.

Striding requires filtering specialized rules that are equivalent to existing ones. We suggested using formal hypothesis bisimulation 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. Many thanks to the Swansea *PLanCompS* group 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. F. Chalub and C. Braga. Maude MSOS tool. *ENTCS*, 176(4):133 – 146, 2007.
2. A. Charguéraud. Pretty-big-step semantics. In *ESOP'13*, volume 7792 of *LNCS*, pages 41–60. Springer Berlin Heidelberg, 2013.
3. M. Churchill and P. D. Mosses. Modular bisimulation theory for computations and values. In *FOSSACS'13*, volume 7794 of *LNCS*, pages 97–112. Springer Berlin Heidelberg, 2013.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.6), 2008. `http://maude.cs.uiuc.edu/maude2-manual/`.
5. D. Clement, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Research Report RR-0416, INRIA, 1985.
6. O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Inf. Process. Lett.*, 106:100–109, 2008.
7. O. Danvy and L. R. Nielsen. Refocusing in Reduction Semantics. BRICS Research Series RS-04-26, Dept. of Computer Science, University of Aarhus, 2004.
8. J. P. Gallagher. Tutorial on specialisation of logic programs. In *PEPM'93*, pages 88–98. ACM, 1993.
9. G. Gupta. Horn logic denotations and their applications. In *The Logic Programming Paradigm*, Artificial Intelligence, pages 127–159. Springer Berlin Heidelberg, 1999.
10. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *TPLP*, 4(1):139–191, 2004.
11. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Log. Program.*, 11(34):217 – 242, 1991.
12. P. D. Mosses. Foundations of Modular SOS. BRICS Research Series RS-99-54, Dept. of Computer Science, University of Aarhus, 1999.
13. P. D. Mosses. Pragmatics of Modular SOS. In *AMAST'02*, volume 2422 of *LNCS*, pages 21–40. Springer Berlin Heidelberg, 2002.
14. P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.
15. P. D. Mosses. Teaching semantics of programming languages with Modular SOS. In *Teaching Formal Methods '06*, Electr. Workshops in Comput. BCS, 2006.
16. P. D. Mosses, M. R. Mousavi, and M. A. Reniers. Robustness of equations under operational extensions. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 106–120, 2010.
17. P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *ENTCS*, 229(4):49 – 66, 2009.
18. M. Pettersson. *Compiling Natural Semantics*, volume 1549 of *LNCS*. Springer Berlin Heidelberg, 1999.
19. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
20. Q. Wang, G. Gupta, and M. Leuschel. Towards provably correct code generation via Horn logical continuation semantics. In *PADL'05*, volume 3350 of *LNCS*, pages 98–112. Springer, 2005.