Abstract

Compositionality of denotational semantics is an important concern in programming semantics. Mathematical operational semantics in the sense of Turi and Plotkin guarantees compositionality, but seen from the point of view of stateful computation it applies only to very fine-grained equivalences that essentially assume unrestricted interference by the environment between any two statements. We introduce the more restrictive stateful SOS rule format for stateful languages. We show that compositionality of two more coarse-grained semantics, respectively given by assuming read-only interference or no interference between steps, remains an undecidable property even for stateful SOS. However, further restricting the rule format in a manner inspired by the cool GSOS formats of Bloom and van Glabbeek, we obtain the streamlined and cool stateful SOS formats, which respectively guarantee compositionality of the two more abstract equivalences.

1 Introduction

A key prerequisite for modular reasoning about process calculi and programming languages is compositionality: A denotational semantics is compositional if the associated semantic equivalence forms a congruence, that is, subterms of a given process or program term may be replaced with equivalent subterms without affecting the overall denotational meaning of the term. For instance, the classical GSOS format of Bloom et al. [8] provides a unified formal...
representation of process languages interpreted over non-deterministic labelled transition systems, and guarantees that bisimilarity is compositional. Similarly, syntactic restrictions of the GSOS format due to Bloom [7] and van Glabbeek [39] guarantee compositionality for coarser equivalences.

More abstractly, GSOS is captured in Turi and Plotkin’s bialgebraic framework of mathematical operational semantics [38], in which sets of operational semantic rules are represented as distributive laws of a monad over a comonad, a principle that has come to be used in widely varying semantic settings [5, 21, 12, 23]. In particular, Turi and Plotkin demonstrated that GSOS rules correspond precisely to natural transformations of type

\[ \varrho_X : \Sigma(X \times (P_\omega X)^L) \rightarrow (P_\omega \Sigma^* X)^L, \]

where \( \Sigma \) is a polynomial functor on the category of sets (representing the signature of the process language at hand), \( L \) is a set of (transition) labels, \( P_\omega \) is the finite power set functor, corresponding to finitary non-determinism, and \( \Sigma^* \) denotes the free (term) monad on \( \Sigma \). This is an instance of an abstract GSOS law, a natural transformation of type \( \Sigma(I \times T) \Rightarrow T \Sigma^* \), with \( T \), the behaviour functor, instantiated to the functor \( P_\omega^L \), which is associated with image-finite \( L \)-labelled transition systems.

There is long-standing interest in SOS style specifications of stateful programming languages [32]. The natural instantiation of mathematical operational semantics to this setting would use \( TX = (S \times (X + 1))^S \) as the behaviour functor (for a given set \( S \) of states). This gives rise to an extremely expressive rule format: In abstract GSOS laws of type \( \Sigma(I \times T) \Rightarrow T \Sigma^* \), program constructs receive their arguments as full-blown state transformers, which in particular they can execute or probe on any number of input states. The semantic domain provided by mathematical operational semantics in this case is the final coalgebra for \( T \), which consists of possibly infinite \( S \)-branching, \( S \)-labelled trees, and thus is an instance of (coalgebraic) resumption semantics [29], originally developed for concurrent settings [13, 10]. The induced notion of semantic equivalence, for which the format guarantees compositionality, is very fine-grained: Being a resumption semantics, it assumes that programs cede complete control to the environment between any two consecutive steps, and thus makes rather few programs equivalent. Capturing less sceptical semantics, such as standard sequential end-to-end net execution, in a compositional manner has proved rather more challenging; generally speaking, compositionality is harder for coarser equivalences because less information is available about the behaviour of subterms [39].

In the present work, we approach this problem by restricting the rule format to various degrees. We first note that the operational rules typically associated to imperative languages resemble GSOS rules with an additional input parameter, the present state. We correspondingly introduce the stateful SOS format for the specification of stateful languages, and show that stateful SOS specifications are in an one-to-one correspondence with natural transformations of type

\[ \delta_X : S \times \Sigma(X \times S \times (X + 1)) \rightarrow S \times (\Sigma^* X + 1). \]

In a small-step operational semantics given in terms of transitions on pairs consisting of states in \( S \) and program terms (or a termination marker \( \checkmark \in 1 \)), \( \delta_X \) assigns to a given state (in \( S \)) and a program construct applied to argument variables with given next-step operational behaviour (i.e. an element of \( \Sigma(X \times S \times (X + 1)) \)) its small-step operational behaviour. Effectively, this means that, in small-step operational semantics, program constructs can execute and probe their arguments only on the current state. We give a resumption semantics (over the final coalgebra for \( T \) as above) for stateful SOS, and show that this semantics agrees with the one obtained by converting \( \delta \) into a GSOS law, in particular is compositional.
Table 1 Separating denotational domains by program equivalences.

<table>
<thead>
<tr>
<th>( \nu \gamma \cdot (S \times (\gamma + 1))^S )</th>
<th>( (x := 1; y := x) = (x := 1; y := 1) )</th>
<th>( (x := 1; x := 2) = (x := 2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S^* + S^\omega )^S</td>
<td>( \times )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( (S + 1)^S )</td>
<td>( \checkmark )</td>
<td>( \times )</td>
</tr>
<tr>
<td></td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>

We go on to define two successive coarsenings of resumption semantics: Trace semantics assumes that the environment can observe but not manipulate states reached in between successive computation steps, and correspondingly uses the semantic domain \( (S^* + S^\omega)^S \), the set of functions expecting an initial state and returning a possibly terminating \( S \)-stream.

The, yet coarser, termination semantics additionally abstracts from the intermediate states of a computation, and thus is defined over the semantic domain \( (S + 1)^S \), the set of functions expecting an initial state and returning either a final state or divergence. Trace semantics has been used, e.g., in the type-theoretic semantics of program logics [25] and in formalizing concurrent systems that feature memory isolation mechanisms [27, 28]. Termination semantics is the semantic domain typically associated with big-step [22, 26] or natural semantics [18], and is a popular choice in settings where fine architectural details are less relevant [33, 31, 30]. Table 1 presents the three domains in decreasing order of granularity and illustrates their differences in terms of the programs they distinguish. Here, \( S \) is the set of variable stores assigning to every program variable its current value. First, consider the programs \( x := 1; x := 2 \) and \( x := 2 \). These are clearly equivalent in termination semantics but not in trace semantics, as the additional initial step of the first program is visible in trace semantics. Similarly, the programs \( x := 1; y := x \) and \( x := 1; y := 1 \) are clearly equivalent under trace semantics but not under resumption semantics, as the latter assumes that the value of \( x \) may be changed by the environment between the two steps. In fact, we show as our first main result that despite the restricted expressiveness, it is undecidable whether the coarser program equivalences are compositional for a given stateful SOS specification. In a subsequent step, we thus introduce two sets of syntactic restrictions in the spirit of Bloom [7] and van Glabbeek [39], and show that these guarantee that stateful SOS specifications have compositional trace semantics or termination semantics, respectively.

Related Work. The above-mentioned cool GSOS rules of Bloom [7] and van Glabbeek [39] guarantee compositionality w.r.t. various flavours of weak bisimilarity; they motivate the cool stateful SOS format we introduce here. In a similar vein, Tsampas et al. [36] present abstract compositionality criteria for weak bisimilarity in the context of mathematical operational semantics [37]. Weak bisimilarity is still rather finer than the main semantics of interest for the present work (trace semantics and termination semantics), as it only abstracts away from steps that do not modify the state, such as \texttt{skip}.

Abou-Saleh and Pattinson [1, 2] consider abstract GSOS specifications for while-languages and construct semantics in Kleisli categories, working at a somewhat higher level of generality than we do here, in particular parametrizing over notions of side-effect. Roughly speaking, the coarsest of their semantics amounts to a steps-until-termination semantics that counts but does not enumerate intermediate states, and thus is coarser than trace semantics but finer than termination semantics. They propose an abstract condition on cones [1, Sec. 4.4] that guarantees compositionality for steps-until-termination semantics. This condition is hard to verify in concrete instances but ensured by evaluation-in-context rule formats [2] that correspond roughly to our cool stateful SOS format, for which we show compositionality even
w.r.t. termination semantics (a goal explicitly mentioned by Abou-Saleh and Pattinson [2, Section 6]). Our streamlined stateful SOS format, which guarantees compositionality of trace semantics, appears to be more permissive than evaluation-in-context.

Bloom and Vandraager [9] and Mousavi et al. [24] propose further SOS-style formats for computations with data and prove compositionality results for semantic equivalences resembling our resumption semantics. We note that these results require fairly tedious proofs; this again highlights the advantage of the categorical approach where they come entirely for free (see Theorem 4.6). The Sfisl format [24] is shown to make trace semantics compositional, but in contrast to our streamlined format it is not expressive enough to cover a fully fledged while-language. Termination semantics is not considered in either of these works.

2 Preliminaries

We assume that readers are familiar with basic notions from category theory such as functors, natural transformations, and monads. In the following we briefly recall some terminology concerning algebras and coalgebras. Throughout, \( \text{Set} \) denotes the category of sets and functions. We write \( 1 = \{\ast\} \) for the terminal object. For a pair \( X_1, X_2 \) of objects we write \( X_1 \times X_2 \) for the product with the projections \( \text{fst}: X_1 \times X_2 \to X_1 \) and \( \text{snd}: X_1 \times X_2 \to X_2 \). For a pair of morphisms \( f_1: Y \to X_i, \ i = 1, 2 \), we let \( (f_1, f_2): Y \to X_1 \times X_2 \) denote the unique induced morphism. The canonical strength of an endofunctor \( F: \text{Set} \to \text{Set} \) is the natural transformation with components \( st_{X,Y}: X \times FY \to F(X \times Y) \) defined by \( st_{X,Y}(x,p) = F(\lambda y. (x, y))(p) \). We usually drop the subscripts \( X \) and \( Y \).

Algebras. Given an endofunctor \( F \) on a category \( C \), an \( F \)-algebra is a pair \( (A, \alpha) \) of an object \( A \) (the carrier of the algebra) and a morphism \( \alpha: FA \to A \) (its structure). A homomorphism from an \( F \)-algebra \( (A, \alpha) \) to an \( F \)-algebra \( (B, \beta) \) is a morphism \( h: A \to B \) of \( C \) such that \( h \cdot \alpha = \beta \cdot Fh \). Algebras for \( F \) and their homomorphisms form a category \( \text{Alg} F \), and an initial \( F \)-algebra is simply an initial object in that category. If it exists, we denote the initial \( F \)-algebra by \( \mu F \) and its structure by \( \nu : F(\mu F) \to \mu F \).

A common example of functor algebras are algebras over a signature. An algebraic signature consists of a set \( \Sigma \) of operation symbols together with a map \( \text{ar}: \Sigma \to \mathbb{N} \) associating to every operation symbol \( f \) its arity \( \text{ar}(f) \). Symbols of arity \( 0 \) are called constants. Every signature \( \Sigma \) induces the polynomial functor \( \Pi_{f \in \Sigma} (-)^{\text{ar}(f)} \) on \( \text{Set} \), which we denote by the same letter \( \Sigma \). An algebra for the functor \( \Sigma \) then is precisely an algebra for the signature \( \Sigma \), i.e. a set \( A \) equipped with an operation \( f_A: A^n \to A \) for every \( n \)-ary operation symbol \( f \in \Sigma \). Homomorphisms between \( \Sigma \)-algebras are maps respecting the algebraic structure.

Given a set \( X \) of variables, we write \( \Sigma^* X \) for the \( \Sigma \)-algebra of terms generated by \( \Sigma \) with variables from \( X \). It is the free \( \Sigma \)-algebra on \( X \), that is, every map \( f: X \to A \) into the carrier of a \( \Sigma \)-algebra \( (A, \alpha) \) uniquely extends to a homomorphism \( \bar{f}: \Sigma^* X \to A \). In particular, the free algebra on the empty set is the initial algebra \( \mu \Sigma \); it is formed by all closed terms of the signature. As shown by Barr [4], the formation of free algebras extends to a monad \( \Sigma^* \): \( \text{Set} \to \text{Set} \), the free monad on \( \Sigma \). For every \( \Sigma \)-algebra \( (A, \alpha) \) we obtain an Eilenberg-Moore algebra \( \bar{\alpha}: \Sigma^* A \to A \) as the free extension of \( \text{id}_A \). This is the map evaluating terms over \( A \) in the algebra.

Coalgebras. A coalgebra for an endofunctor \( F \) on \( C \) is a pair \( (C, \gamma) \) of an object \( C \) (the carrier) and a morphism \( \gamma: C \to FC \) (its structure). A homomorphism from an \( F \)-coalgebra \( (C, \gamma) \) to an \( F \)-coalgebra \( (D, \delta) \) is a morphism \( h: C \to D \) such that \( Fh \cdot \gamma = \delta \cdot h \). Coalgebras
for \( F \) and their homomorphisms form a category \( \text{Coalg} F \), and a final coalgebra is a final object in that category. If it exists, we denote the final \( F \)-coalgebra by \( \nu F \) and its structure by \( \tau : \nu F \to F(\nu F) \), and we write \( \gamma : (C, \gamma) \to (\nu F, \tau) \) for the unique homomorphism.

\begin{itemize}
  \item \textbf{Example 2.1.}
  \begin{enumerate}
    \item Fix a set \( S \). The set functor \( BX = S \times (X + 1) \) has a final coalgebra carried by \( \nu B = S^+ + S^\omega \), the set of all non-empty possibly terminating \( S \)-streams. Its coalgebra structure \( S^+ + S^\omega \to S \times (S^+ + S^\omega + 1) \) sends a stream \( sw \) (where \( s \in S \) and \( w \in S^+ + S^\omega \)) to \( (s, w) \) if \( w \) is empty, and to \( (s, *) \) if \( w \) is non-empty.
    \item Similarly, for the set functor \( TX = (BX)^S = (S \times (X + 1))^S \), the terminal coalgebra is carried by the set of possibly infinite \( S \)-ary trees (i.e. every node is either a leaf or has an \( S \)-indexed set of children) that have more than one node and where every edge is labelled by an element of \( S \). The coalgebra structure \( vT \to (S \times (vT + 1))^S \) sends a tree \( t \) to the map \( s \mapsto (s', t') \) where \( s' \) is the label of the edge from the root to its \( s \)-th child, and \( t' \) is the subtree rooted at that child if it has more than one node, or \( * \) otherwise.
  \end{enumerate}
\end{itemize}

3 \textbf{Stateful SOS Specifications}

We start off with an observation on the standard operational semantics for sequential composition in imperative languages (see e.g. Plotkin [32]), given by the following rules:

\[
\begin{align*}
\text{seq1} & : s, p \downarrow s' \quad \text{seq2} & : s, p \rightarrow s', p' \\
& s, (p; q) \rightarrow s', q & & s, (p; q) \rightarrow s', (p'; q)
\end{align*}
\]

Rule seq1 asserts that if a program \( p \), on input (state) \( s \), terminates and produces a new state \( s' \), then the program \( p; q \), on input state \( s \), evolves to program \( q \) and produces the new state \( s' \). The other case is captured by rule seq2, which asserts that if \( p \), on input \( s \), transitions to \( p' \) and produces \( s' \), then \( p; q \), on input \( s \), transitions to \( p' \); \( q \) and produces \( s' \).

Note that for both rules, the input \( s \) is the same in the premises and in the conclusion. Consequently, to decide how \( p; q \) transitions from \( s \) in the next step, we need to know only how \( p \) behaves on \( s \), which we can regard as the input of the entire rule. This allows us to give a concise categorical formulation of the rules seq1 and seq2 in terms of a natural transformation \( S \times (X \times S \times (X + 1))^2 \to (S \times (\Sigma^* X + 1)) \) where \( \Sigma \) is a signature containing the binary operation symbol ‘\( \cdot \)’. The transformation is defined by

\[
(s, (x, s', *), (y, \_, \_)) \mapsto (s', y) \quad \text{and} \quad (s, (x, s', x'), (y, \_, \_)) \mapsto (s', (x'; y)).
\]

Compare the above with the interpretation obtained by instantiating the GSOS principle [38] to stateful computations in the standard manner [37]. The interpretation of ‘\( \cdot \)’ is then given as a natural transformation \( (X \times (S \times (X + 1))^S)^2 \to (S \times (\Sigma^* X + 1))^S \) whose uncurried form \( S \times (X \times (S \times (X + 1))^S)^2 \to S \times (\Sigma^* X + 1) \) is defined by

\[
(s, (x, f), (y, \_)) \mapsto \begin{cases} 
(s', y) & \text{if } f(s) = (s', *), \\
(s', (x'; y)) & \text{if } f(s) = (s', x').
\end{cases}
\]

In this setting, the semantics of \( p; q \) receives the next-step behaviours of \( p, q \) as state transformers, and can in principle probe these state transformers on arbitrary states (of course, for ‘\( \cdot \)’, this does not actually happen). By contrast, our rule format, the stateful SOS format formally introduced next, embodies the restriction that the behaviour of a complex term on an input state \( s \) is predicated only on the behaviour of its subterms on \( s \). It is this trade-off in expressiveness that buys our compositionality results for stateful SOS specifications.
The Stateful SOS Rule Format. We proceed to underpin the intuition given above with formal definitions. We fix a countably infinite set $\mathcal{V} = \{x_1, x_2, \ldots \} \cup \{y_1, y_2, \ldots \}$ of (meta-)variables and a countable set $S$ of states; in typical applications the elements of $S$ are variable stores. Moreover, we fix an algebraic signature $\Sigma$, equivalently a polynomial functor also denoted $\Sigma$ (cf. Section 2). We think of the operations in $\Sigma$ as program constructs, and correspondingly, programs are closed $\Sigma$-terms, i.e. terms formed using only the operations in $\Sigma$, with constants in $\Sigma$ forming the base case.

Definition 3.1 (Literals). A progressing $\Sigma$-literal is an expression $s, p \rightarrow s', q$ with $p, q \in \Sigma^* \mathcal{V}$ and $s, s' \in S$. We say that $s$ is the input, $p$ is the source, $s'$ is the output and $q$ is the target of the literal. A terminating $\Sigma$-literal is an expression $s, p \downarrow s'$ with $s, s' \in S$ and $p \in \Sigma^* \mathcal{V}$. In this case, $s$ is the input, $p$ is the source and $s'$ is the output of the literal. A $\Sigma$-literal (without further qualification) is either a progressing or a terminating $\Sigma$-literal.

Our rule format shares some similarities with stream GSOS [19, Def. 37].

Definition 3.2 (Rules). A stateful SOS rule for an $n$-ary operator $f \in \Sigma$ is an expression

\[
\frac{l_1 \ldots l_n}{L}
\]

(or, in inline notation, $l_1 \ldots l_n/L$) where $l_1, \ldots, l_n$ (the premisses of the rule) and $L$ (the conclusion of the rule) are $\Sigma$-literals that have the same input $s \in S$, the input of the rule, and satisfy the following conditions:

1. The source of the premiss $l_j$ is the variable $x_j$, and the target is $y_j$ if $l_j$ is progressing.
2. The source of the conclusion $L$ is the term $f(x_1, \ldots, x_n)$. Moreover, if $L$ is progressing, the variables of its target term appear either as the source or the target of some premiss.

The rule is progressing if $L$ is progressing, and otherwise the rule is terminating. The trigger of the rule is the tuple formed by its input $s$ together with the sequence of pairs $(s', c) = (s'_1, c_1), \ldots, (s'_n, c_n)$, where $s'_j$ is the output of $l_j$ and $c_j \in \{\text{pr, te}\}$ indicates whether $l_j$ is progressing ($c_j = \text{pr}$) or terminating ($c_j = \text{te}$).

Definition 3.3. A stateful SOS specification is a set of stateful SOS rules such that for each $n$-ary operator $f$, each $s \in S$ and each sequence $(s', c) = (s'_1, c_1), \ldots, (s'_n, c_n)$ where $s'_j \in S$ and $c_j \in \{\text{pr, te}\}$, there is exactly one rule for $f$ with trigger $(s, (s', c))$.

Notation 3.4. By writing

\[
\frac{l_1 \ldots l_{j-1} l_{j+1} \ldots l_n}{L}
\]

we mean the set of all stateful SOS rules of the form $l_1 \ldots l_n/L$ (with the missing premiss $l_j$ filled in in any way possible). This captures the situation where the behaviour of the source $f(x_1, \ldots, x_n)$ of $L$ does not depend on the behaviour of $x_j$, given $l_1, \ldots, l_{j-1}, l_{j+1}, \ldots, l_n$.

Remark 3.5. The use of fixed enumerated variables $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ simplifies abstract reasoning about stateful SOS (e.g. Theorem 3.9 below). In examples, we use arbitrary variable names such as $p, q, x, y$, and we typically write rules using rule schemes, using hopefully self-explanatory notation. For instance, rule $\text{seq1}$ in Figure 1 (discussed in detail in Example 3.6) is to be understood as the set $\{s, p \downarrow s' / s, (p; q) \rightarrow s', q | s, s' \in S\}$ of stateful SOS rules, with variables $p, q$, and rule $\text{while1}$ as the set $\{|/s, \text{while } c p \downarrow s | s \in S, [e_c] = 0\}$ (with premiss omitted as per Notation 3.4). Note the side condition $[e_c] = 0$ (expression $c$ evaluates to 0 in state $s$) of $\text{while1}$; the rule schemes and their side conditions need to be
Definition 3.7. ▶ $x$ does not affect the program state, we instead assume a denotational semantics for simplicity).

The operational semantics of $s, p$ is given by the grammar

$$
\begin{align*}
\text{skip} & \quad s, \text{skip} \xrightarrow{} s \\
\text{while1} & \quad s, \text{while } e \ p \xrightarrow{} [e]_s = 0 \\
\text{seq1} & \quad s, p \xrightarrow{} s', q \\
\text{while2} & \quad s, \text{while } e \ p \rightarrow s, (p; \text{while } e \ p) \\
\text{seq2} & \quad s, p \rightarrow s', p' \\
\end{align*}
$$

Figure 1 Operational semantics of While.

The operational semantics of While in the form of a stateful SOS specification is shown in Figure 1, using rule schemes as per Remark 3.5.

As indicated by the discussion at the beginning of this section, stateful SOS specifications can be represented as natural transformations:

Definition 3.7. A stateful SOS law is a natural transformation

$$
\delta_X: S \times \Sigma(X \times S \times (X + 1)) \rightarrow S \times (\Sigma^*X + 1) \quad (X \in \text{Set}).
$$

Remark 3.8.

1. Every stateful SOS specification $\mathcal{E}$ yields a stateful SOS law

$$
\delta_X = [\delta^X_f]_{f \in \Sigma}: S \times \Sigma(X \times S \times (X + 1)) \rightarrow S \times (\Sigma^*X + 1) \quad (X \in \text{Set})
$$

by distributing $S \times (-)$ over $\Sigma(X \times S \times (X + 1))$ and copairing the maps

$$
\delta^X_f: S \times (X \times S \times (X + 1))^\text{ar}(f) \rightarrow S \times (\Sigma^*X + 1) \quad (f \in \Sigma)
$$

(3.3)
defined as follows. Given \( (s, ((v_1, s'_1, w_1), \ldots, (v_n, s'_n, w_n))) \in S \times (X \times X \times (X + 1))^n \) with \( n = \text{ar}(f) \), let \( t_1 \ldots t_n/L \) be the unique rule in \( \mathcal{L} \) with source \( f \) and trigger \( (s, ((s'_1, c_1), \ldots, (s'_n, c_n))) \) where \( c_j = \text{pr} \) if \( w_j \in X \) and \( c_j = \text{te} \) if \( w_j = * \). Let \( s' \) be the output of \( L \). Then \( \delta_T^s(s, ((v_1, s'_1, w_1), \ldots, (v_n, s'_n, w_n))) \) is \( (s', *) \) if the rule is terminating, and otherwise \( (s', t') \) where \( t' \in \Sigma^* X \) is the term obtained from the target \( t \in \Sigma^* V \) of \( L \) by substituting \( x_j \) by \( v_j \) and \( y_j \) by \( w_j \) (the latter whenever \( c_j = \text{pr} \)).

2. Conversely, every stateful SOS law \( \delta \) yields a stateful SOS specification \( \mathcal{L} \) whose rules are defined as follows. For every \( n \)-ary operation symbol \( f \in \Sigma \), \( s, s'_1, \ldots, s'_n \in S \) and \( W \subseteq \{1, \ldots, n\} \), let \( (s', t) \) be the value of \( \delta_T^s \) on \( (s, ((x_1, s'_1, w_1), \ldots, (x_n, s'_n, w_n))) \) where \( w_j = y_j \) if \( j \in W \) and \( w_j = * \) otherwise. If \( t \in \Sigma^* V \), then \( \mathcal{L} \) contains the rule

\[
\frac{(s, x_j \to s'_j, y_j)_{j \in W}}{s, f(x_1, \ldots, x_n) \to s', t},
\]

and if \( t = * \), then \( \mathcal{L} \) contains the rule

\[
\frac{(s, x_j \to s'_j, y_j)_{j \in W}}{s, f(x_1, \ldots, x_n) \downarrow s'}.\]

**Theorem 3.9.** There is a bijective correspondence between (1) stateful SOS specifications, (2) stateful SOS laws, and (3) families of maps of the form

\[
(r_t : S \times S^{\text{ar}(f)} \to S \times \Sigma^*(\text{ar}(f) + W) + S)_{t \in \Sigma, \emptyset \subseteq \text{ar}(f)}.
\]

Here we identify the natural number \( \text{ar}(f) \) with the set \( \{1, \ldots, \text{ar}(f)\} \).

The correspondence between (1) and (2) is given by the translations of Remark 3.8, and the correspondence between (2) and (3) is shown using the Yoneda lemma.

4 Categorical Semantics and Compositionality

We proceed to develop a categorical treatment of stateful SOS along the lines of mathematical operational semantics in the style of Turi and Plotkin [38] and Bartels [5]. Furthermore, we shall define two semantic domains of interest, both coarser than the one initially obtained through Turi-Plotkin semantics, and show that the problem of whether a given stateful SOS specification is compositional is undecidable. We recall that if the denotational semantics of a programming language is given by a map \([\_]\) : \( \mu \Sigma \to D \) into a semantic domain \( D \), then it is called *compositional* if the corresponding behavioural equivalence forms a congruence, that is, for every \( n \)-ary operator \( f \in \Sigma \) and programs \( p_i, q_i \in \mu \Sigma \) \( (i = 1, \ldots, n) \),

\[
[\mathcal{B}p_i] = [\mathcal{B}q_i] \quad \text{for} \quad i = 1, \ldots, n \quad \text{implies} \quad [\mathcal{B}(f(p_1, \ldots, p_n))] = [\mathcal{B}(f(q_1, \ldots, q_n))].
\]

Compositionality asserts that subprograms of a program \( p \) may be replaced with equivalent subprograms without affecting the semantics of \( p \), and thus allows modular reasoning.

4.1 GSOS Laws

Turi and Plotkin’s *mathematical operational semantics* [38] identifies sets of rules in structural operational semantics (SOS) with distributive laws of various types on a cartesian base category. We will work more specifically with distributive laws of free monads over cofree copointed functors on the base category \( \text{Set} \), where the free monad is associated to a polynomial functor. Such distributive laws can equivalently be presented as follows.
Definition 4.1. Given a polynomial functor $\Sigma$ and an endofunctor $T$ on $\text{Set}$, a GSOS law of $\Sigma$ over $T$ is a natural transformation $\varrho: \Sigma(\text{Id} \times T) \Rightarrow T\Sigma^*$.

We shall see below that stateful SOS laws determine GSOS laws. The interested reader may find further examples of GSOS laws in the literature [37, 5, 19]. Roughly speaking, the input of $\varrho$ is a (program) operation applied to pairs each consisting of a meta-variable and its assumed next-step behaviour (encapsulated in $T$), and the output is a next-step behaviour reaching poststates given as programs with meta-variables.

Given a GSOS law $\varrho$, the initial $\Sigma$-algebra can be equipped with a unique $T$-coalgebra structure $\gamma: \mu\Sigma \rightarrow T(\mu\Sigma)$ such that the diagram (4.1)

\[
\begin{array}{ccc}
\Sigma(\mu\Sigma) & \xrightarrow{\iota} & \mu\Sigma \\
\Sigma(\mu\Sigma \times T(\mu\Sigma)) & \xrightarrow{\varrho_{\Sigma}} & T\Sigma^*(\mu\Sigma) \\
\end{array}
\]

commutes (see Section 2 for the notation). The coalgebra $(\mu\Sigma, \gamma)$ is called the operational model of $\varrho$. Dually, assuming the existence of a final coalgebra $\nu T$, there is a unique $\Sigma$-algebra structure $\alpha: \Sigma(\nu T) \rightarrow \nu T$ such that the following diagram commutes:

\[
\begin{array}{ccc}
\Sigma(\nu T) & \xrightarrow{\Sigma(\text{id}, \tau)} & \Sigma(\nu T \times T(\nu T)) \\
& & \xrightarrow{\varrho_{\tau}} T\Sigma^*(\nu T) \\
\nu T & \xrightarrow{\tau} & T(\nu T) \\
\end{array}
\]

(4.2)

The algebra $(\nu T, \alpha)$ is the denotational model of $\varrho$. A fundamental well-behavedness property of GSOS laws is that the unique $\Sigma$-algebra homomorphism $(\mu\Sigma, \iota) \rightarrow (\nu T, \alpha)$ and the unique $T$-coalgebra homomorphism $(\mu\Sigma, \gamma) \rightarrow (\nu T, \tau)$ coincide. We denote this morphism by

\[
\text{beh}_\varrho: \mu\Sigma \rightarrow \nu T,
\]

(4.3)

and we think of it as assigning to programs their denotational behaviour. Compositionality of this semantics is immediate from the fact that $\text{beh}_\varrho$ is a $\Sigma$-algebra homomorphism.

4.2 Semantic Domains for Stateful SOS

We proceed to introduce three denotational semantics of stateful SOS, in order of increasing abstraction: resumption semantics, in which the program essentially cedes control to the environment between any two program steps; trace semantics, where the environment may observe but not manipulate the state between program steps; and termination semantics, in which only the effect of executing the program end-to-end is observable.

Notation 4.2. From now on, we instantiate the functor $T$ of Definition 4.1 to

\[TX = (S \times (X + 1))^S,\]

for a fixed set $S$ of states. Thus $T$ represents state transformers with possible non-termination.

Resumption semantics. Every stateful SOS law $\delta$ (see Definition 3.7) canonically induces a GSOS law

\[\delta: \Sigma(\text{Id} \times T) \Rightarrow T\Sigma^*.\]
This will guarantee compositionality for the most fine-grained of our semantics, which we shall refer to as resumption semantics, via established methods of mathematical operational semantics as recalled above. Details are as follows. The component \( \hat{\delta} \) is obtained by currying the composite

\[
S \times \Sigma(X \times TX) \xrightarrow{(\text{fst, st})} S \times \Sigma(S \times (X \times TX)) \cong S \times \Sigma(X \times (S \times TX)) \xrightarrow{id \times \Sigma(id \times ev)} S \times \Sigma(X \times X \times (S \times TX)) \xrightarrow{\hat{\delta} \Sigma} S \times (\Sigma^* X + 1),
\]

(4.4)

where \( \text{st}: S \times \Sigma(X \times TX) \to \Sigma(S \times (X \times TX)) \) is the strength (cf. Section 2) and \( \text{ev}: S \times TX \to S \times (S \times (X + 1))^S \to S \times (X + 1) \) denotes the evaluation map. Recall from Example 2.1 that the final coalgebra for \( T \) is carried by the set of possibly infinite \( S \)-branching trees, with edges labelled in \( S \). Using (4.1) we obtain the operational model \( \gamma: \mu \Sigma \to T(\mu \Sigma) \) associated to \( \delta \). In terms of stateful SOS specifications, it can be described as follows.

\begin{definition}
Given a stateful SOS specification \( \mathcal{L} \), its transition function is the map \( \gamma_0: S \times \mu \Sigma \to S \times (\mu \Sigma + 1) \) inductively defined by

\[
\gamma_0(s, t_1, \ldots, t_n) = m(d_j^f(s, (d_1, \ldots, d_n)))
\]

where

\[
d_j = (t_j, \gamma_0(s, t_j)) \quad \text{and} \quad m = (S \times (\Sigma^* (\mu \Sigma) + 1) \xrightarrow{id \times (\nu^* + \id)} S \times (\mu \Sigma + 1)),
\]

using the term evaluation map \( \nu: \Sigma^* (\mu \Sigma) \to \mu \Sigma \), and \( d_j^f \) as in (3.3). Thus, \( \gamma_0(s, p) \) performs the first computation step of program \( p \) on input \( s \) according to the specification \( \mathcal{L} \). We write

\[
s, p \to s', p' \quad \text{and} \quad s, p \downarrow s'
\]

if \( \gamma_0(s, p) = (s', p') \) and \( \gamma_0(s, p) = (s', *) \), respectively.
\end{definition}

\begin{proposition}
Let \( \mathcal{L} \) be a stateful SOS specification with its associated transition function \( \gamma_0 \) and operational model \( \gamma \). Then

\[
\gamma = \text{curry}(\gamma_0): \mu \Sigma \to (S \times (\mu \Sigma + 1))^S.
\]

The proof makes use of an induction principle that combines primitive recursion (see e.g. [16, Prop. 2.4.7]) and induction with parameters (see e.g. [16, Exercise 2.5.5]).
\end{proposition}

\begin{definition}
The resumption semantics of a stateful SOS specification \( \mathcal{L} \) is given by

\[
[-]_\mathcal{L} = \text{beh}_\delta: \mu \Sigma \to \nu T,
\]

where \( \delta \) is the stateful SOS law associated to \( \mathcal{L} \), \( \delta \) is as per (4.4), and \( \text{beh} \) is defined in (4.3). Let \( \sim_\mathcal{L} \) denote the corresponding behavioural equivalence, that is, \( p \sim_\mathcal{L} q \) iff \( [p]_\mathcal{L} = [q]_\mathcal{L} \) for a given pair \( p, q \in \mu \Sigma \). We drop subscripts if \( \mathcal{L} \) is clear from the context.

Note that since \( T \) preserves weak pullbacks, \( \sim_\mathcal{L} \) coincides with \( T \)-bisimilarity in the operational model \( \gamma: \mu \Sigma \to T(\mu \Sigma) \) [34]. From the discussion in Section 4.1 we immediately get
\end{definition}

\begin{theorem}
The resumption semantics of stateful SOS specifications is compositional.
Resumption semantics is very fine-grained, essentially because it does not pass the output state of a computation step on as the input state of the next step; that is, resumption semantics assumes that the environment takes complete control in between steps. For instance, consider the While programs

\[
t_1 = (x := 1; x := x + 1) \quad \text{and} \quad t_2 = (x := 1; x := x \ast 2).
\]

The resumption semantics of these programs in each case consists in an \(S\)-branching tree of depth 2, in which the edge from the root to its \(s\)-th child is labelled \(s[x \leftarrow 1]\) and the edges at the next level are correspondingly labelled according to the effect of the assignments \(x := x + 1\) and \(x := x \ast 2\), respectively. In particular, the semantics of the two programs differ – as intuitively expected under a resumption semantics, since the environment may manipulate the value of \(x\) in between the two assignments. To obtain a more coarse-grained notion of process equivalence, we have to quotient the semantic domain \(\nu T\) further.

**Trace Semantics.** Consider the set functor \(B\) given by

\[
BX = S \times (X + 1);
\]

thus \(TX = (BX)^S\). Recall from Example 2.1 that the final coalgebra \(\nu B\) is carried by the set \(S^* + S^\infty\) of possibly terminating \(S\)-streams. The set \((\nu B)^S\) serves as the semantic domain for trace semantics for imperative programs [25, 27, 28], which associates to a program the possibly terminating sequence of states it computes from a given initial state. In order to formally introduce trace semantics in our setting, we proceed to construct a quotient map \(\nu T \twoheadrightarrow (\nu B)^S\) by coinduction. To this end, we define the functor \((\overline{\cdot})\) : \(\text{Coalg}\,T \rightarrow \text{Coalg}\,B\), which maps a \(T\)-coalgebra \((C, \zeta)\) to the \(B\)-coalgebra

\[
\overline{\zeta} = S \times C \xrightarrow{\text{id} \times \overline{\zeta}} S \times (BC)^S \xrightarrow{\text{ev}} BC = S \times (C + 1) \xrightarrow{(\text{fst}, \text{st})} S \times (S \times C + 1) = B(S \times C),
\]

where \(\text{st} : S \times (C + 1) \rightarrow S \times C^1\) is the strength of the functor \((-) + 1\), given by \((s, c) \mapsto (s, c)\) and \((s, *) \mapsto s\). Intuitively, while \(\overline{\zeta} : C \rightarrow \nu T\) (see Section 2 for the notation) maps a coalgebra state of \(C\) to its tree of state transformers, \(\overline{\zeta}((s, x)) \in \nu B\) executes all these state transformers without interruption, beginning at \(s\) and feeding the output state of each previous step to the next step, and outputs the intermediate states reached in each step. Applying \((\overline{\cdot})\) to the final coalgebra \((\nu T, \tau)\), we obtain a \(B\)-coalgebra \((S \times \nu T, \tau)\), and currying the unique coalgebra homomorphism \(\tau^2 : S \times \nu T \rightarrow \nu B\) yields the desired quotient map

\[
\text{trc} = \text{curry}(\tau^2) : \nu T \twoheadrightarrow (\nu B)^S.
\]

**(4.5)**

**Proposition 4.7.** The map \(\text{trc}\) is surjective.

**Definition 4.8.** The trace semantics of a stateful SOS specification \(L\) is given by

\[
[-]_L = (\mu \Sigma \xrightarrow{[-]} \nu T \xrightarrow{\text{trc}} (\nu B)^S).
\]

Let \(\simeq_L\) denote the corresponding behavioural equivalence, that is, \(p \simeq_L q\) iff \(\llbracket p \rrbracket_L = \llbracket q \rrbracket_L\), for \(p, q \in \mu \Sigma\). We drop subscripts if \(L\) is clear from the context.

**Remark 4.9.** Equivalently, \(\llbracket - \rrbracket_L\) is the curried form of the unique \(B\)-coalgebra homomorphism from \((S \times \mu \Sigma, \overline{\gamma})\) to \(\nu B\) (recall that \((\mu \Sigma, \gamma)\) is the operational model of \(L\)). Since

\[
\overline{\gamma} = (S \times \mu \Sigma \xrightarrow{\text{fst}} S \times (\mu \Sigma + 1) \xrightarrow{(\text{fst}, \text{st})} S \times (S \times \mu \Sigma + 1) = B(S \times \mu \Sigma))
\]
by definition of $\bar{\gamma}$ and Proposition 4.4, we see that for every $p \in \mu \Sigma$ and $s \in S$, the possibly infinite stream $[p]_{\mathcal{L}}(s) = s_1 s_2 s_3 \cdots$ is the sequence of states computed by the program $p$ on input state $s$, cf. Definition 4.3:

$$s, p \rightarrow s_1, p_1 \rightarrow s_2, p_2 \rightarrow s_3, p_3 \rightarrow \cdots.$$  

Hence trace equivalence $p \simeq q$ holds iff for each input state $s$, programs $p$ and $q$ produce the same sequence of states.

The following example demonstrates that trace semantics is generally not compositional:

$\triangleright$ Example 4.10. We extend While by adding a unary operator $[\_]$ with

$$
\begin{array}{c|c|c}
  s, [p] & s' & s, [p] \downarrow s'\\
  s & s_1 & s, [p] \downarrow s'
\end{array}
$$

where $\emptyset$ denotes the store with all variables set to 0. For $t_1 = (x := 1; x := x + 1)$ and $t_2 = (x := 1; x := x \times 2)$, we have that $t_1 \simeq t_2$ but $[t_1] \not\simeq [t_2]$ (since in $[t_1]$ and $[t_2]$, the store is erased after the first assignment).

Termination Semantics. As the coarsest of our semantic domains, we shall use the set $(S + \{\perp\})^S \cong (S + 1)^S$ of state transformers on $S$ with possible non-termination featuring pervasively in the denotational semantics of imperative programming (e.g. [33, 31, 30]). In comparison to $(\nu B)^S$, this domain abstracts from the intermediate steps of the computation. The essence of this abstraction is captured by the map

$$\text{fn}: \nu B \rightarrow S + 1 \quad \text{defined by} \quad \text{fn}(x) = \begin{cases} s & \text{if } x \text{ is finite, with last state } s, \\ \perp & \text{otherwise.} \end{cases}$$

$\triangleright$ Definition 4.11. The termination semantics of a stateful SOS specification $\mathcal{L}$ is given by

$$\begin{array}{c}
\llbracket - \rrbracket_{\mathcal{L}} = (\mu \Sigma \xrightarrow{\text{trc}} (\nu B)^S \xrightarrow{\text{fn}^S} (S + 1)^S).
\end{array}$$

Let $\approx_{\mathcal{L}}$ denote the corresponding behavioural equivalence, that is, $p \approx_{\mathcal{L}} q$ iff $\llbracket p \rrbracket_{\mathcal{L}} = \llbracket q \rrbracket_{\mathcal{L}}$ for $p, q \in \mu \Sigma$. We drop subscripts if $\mathcal{L}$ is clear from the context.

Thus $p \approx q$ iff for each initial state $s$, if $p$ eventually terminates with final state $s'$ then $q$ eventually terminates with final state $s'$ and vice-versa. Termination semantics is generally not compositional: the programs $t_1$ and $t_2$ of Example 4.10 satisfy $t_1 \approx t_2$ but $[t_1] \not\approx [t_2]$.

The maps introduced in this section are summarized in the following commutative diagram:

$$
\begin{array}{c}
\llbracket - \rrbracket_{\mathcal{L}} \xrightarrow{\text{trc}} (\nu B)^S \xrightarrow{\text{fn}^S} (S + 1)^S
\end{array}

4.3 Compositionality is Undecidable

We have seen that in contrast to resumption semantics, both trace and termination semantics generally fail to be compositional. As it turns out, reasoning about compositionality in these two cases is a very complex, viz. undecidable, task.
To make the ensuing decision problems precise, we fix suitable encodings of states and terms as finite strings and regard a stateful SOS specification $L$ as a total function that assigns to a given operation symbol, input state and list of premisses the target of the conclusion and output state of the respective rule. From a computational point of view, a minimum requirement on every reasonable specification $L$ is that it admits some finite representation. Hence, for simplicity, we assume in the following theorem that specifications are primitive recursive functions. For instance, this is clearly the case for the While language.

▶ **Theorem 4.12.** It is undecidable whether the trace semantics (or termination semantics, respectively) induced by a primitive recursive stateful SOS specification is compositional.

**Proof sketch.** The halting problem reduces to the compositionality problem. The idea is to take programs akin to $t_1$ and $t_2$ in Example 4.10 and precompose them with the simulation of a given Turing machine. This can be specified in stateful SOS. The failure of compositionality described in Example 4.10 then occurs if, and only if, the simulated machine halts. ◀

In view of the fact that there is no sound and complete decision procedure for compositionality w.r.t. $\simeq$ and $\approx$, we instead move on to identify easily checked sound syntactic criteria that, although necessarily incomplete, are sufficiently broad.

## 5 Cooling the Stateful SOS Format

We now introduce two sets of restrictions on the stateful SOS rule format, called streamlined stateful SOS and cool stateful SOS, that guarantee trace and termination semantics, respectively, to be compositional. Our approach is inspired by the work of Bloom [7] and van Glabbeek [39] on the cool congruence formats for weak bisimilarity for GSOS specifications. The following definition will help describe the restricted formats. We make pervasive use of the abbreviations from Notation 3.4, and we will additionally employ $s, p \rightarrow s', *$ as an alternative notation for a terminating literal $s, p \downarrow s'$.

▶ **Definition 5.1.** Let $L$ be a stateful SOS specification.

1. An $n$-ary operator $f$ is passive if all rules for $f$ are of the form

   $s, f(x_1, \ldots, x_n) \rightarrow s', t$

   where $t \in \Sigma^*(\{x_1, \ldots, x_n\})$ or $t = *$.

   In other words, the one-step behaviour of $f(x_1, \ldots, x_n)$ does not depend on the one-step behaviour of any of its subterms. In particular, every constant is passive. An active operator is one which is not passive.

2. A progressing rule for an $n$-ary operator $f$ is receiving at position $j \in \{1, \ldots, n\}$ if its $j$-th premiss $s, x_j \rightarrow s', y_j$ is progressing and the variable $y_j$ appears in the target of the conclusion. We say that the rule is receiving if it is receiving at some position $j$.

### 5.1 Streamlined Stateful SOS

As indicated above, the streamlined Stateful SOS format, introduced next, will guarantee compositionality of trace semantics.

▶ **Definition 5.2.** A stateful SOS specification is streamlined if for every active operator $f$ of arity $n$ there exists $j \in \{1, \ldots, n\}$ (the receiving position of $f$) such that the following holds:
1. All receiving rules for $f$ are of the form

$$s, x_j \rightarrow s', y_j$$

where $t = f(x_1, \ldots, x_n)[y_j/x_j]$ or $t = y_j$;

here, $[u/x]$ denotes substitution of the variable $x$ by the term $u$.

2. All non-receiving rules for $f$ are of the form

$$l_1 \rightarrow l_2 \rightarrow \cdots \rightarrow l_n$$

where $t \in \Sigma^* \cup \Lambda \cup \{ \{x_1, \ldots, x_n \} \setminus \{x_j \} \}$ or $t = \\ast$.

Note that in a stateful SOS specification, receiving rules for an active operator $f$ are receiving only in the receiving position of $f$. What Definition 5.2 boils down to is that an active operator can only progress its subterm at the receiving position $j$, leaving everything else unchanged and making sure that the output state in the $j$-th premise is correctly propagated, and discards the $j$-th subterm once it terminates.

**Example 5.3.** The While language (cf. Figure 1) is streamlined. The only active operator is sequential composition $p \cdot q$. Its progressing rules are receiving in the left position, and upon termination the left subterm is discarded.

Further examples are discussed after Corollary 5.5.

**Theorem 5.4.** Trace semantics is compositional for streamlined stateful SOS specifications.

**Proof sketch.** For $p, q \in \mu \Sigma$ and $k \in \mathbb{N}$ we put $p \simeq_k q$ if the programs $p$ and $q$ are $k$-step trace equivalent, that is, for every $s \in S$ the streams $[p](s)$ and $[q](s)$ have the same prefix of length at most $k$. By induction on $k$ one proves $\simeq_k$ to be a congruence, using a judicious strengthening of the inductive claim for receiving positions of active operators. This implies that $\simeq$ is a congruence, whence trace semantics is compositional.

From Theorem 5.4 we can deduce a slightly stronger statement. In what follows, the *kernel* of a map $e : X \rightarrow Y$ is the equivalence relation on $X$ relating $x, x'$ iff $e(x) = e(x')$.

**Corollary 5.5.** For every streamlined stateful SOS specification, the kernel of the map $\nu T \rightarrow (\nu B)^S$ is a congruence w.r.t. the canonical $\Sigma$-algebra structure on $\nu T$ as per (4.2).

We next look at examples of streamlined specifications but also at a few pathological cases where compositionality breaks.

**Example 5.6.** Streamlined specifications allow for complex control flow over programs, including signal or interrupt handling. For instance, we can extend While by a distinguished variable $i$ serving as an interrupt flag and modify the rules of sequential composition to

$$s, p \downarrow s' \rightarrow s', q$$

$$s, p \rightarrow s', p' \rightarrow [1], s = 0$$

$$s, p \rightarrow s', p' \rightarrow [1], s \neq 0 \land P(s')$$

where $P \subseteq S$. If flag $i$ is enabled and predicate $P$ is true for the output $s'$ of $p$, then $p$ is terminated prematurely. This type of rules can also be used to implement listeners or observers in high-level programming languages [17].
We now further restrict the streamlined format as follows:

A stateful SOS specification is cool if for every active operator \( f \) there exists \( j \in \{1, \ldots, n\} \) (again called the receiving position of \( f \)) such that the following holds:

1. All rules for \( f \) whose \( j \)-th premiss is progressing are of the form
   \[
   s, x_1, \ldots, x_n \rightarrow s', y_j
   \]
   \[
   s, f(x_1, \ldots, x_n) \rightarrow s', f(x_1, \ldots, x_n)[y_j/x_j]
   \]

2. All rules for \( f \) whose \( j \)-th premiss is terminating are of the form
   \[
   s, x_1, \ldots, x_n \rightarrow s''
   \]
   \[
   s, f(x_1, \ldots, x_n) \rightarrow s'', t
   \]
   where \( t \in \Sigma^*([x_1, \ldots, x_n] \setminus \{x_j\}) \) or \( t = * \),

   and moreover \( s'' \) and \( t \) depend only on \( s' \) but not on \( s \).

A stateful SOS specification is uncool if it is not cool.

The cool format asserts that an active operator \( f \) runs its \( j \)-th subterm until termination and then discards it, proceeding to a state derivable from the terminating state of the subterm. In GSOS, rules of type 1 (without states) are known as patience rules [39].
Example 5.9. The rules of the While language, which we have already observed to be streamlined (Example 5.3), are also cool.

Cool stateful SOS specifications are streamlined, and all of the negative examples from Section 5.1 apply here as well. Here is an example that separates the two concepts:

Example 5.10. The sequential composition semantics with interrupts from Example 5.6 is uncool, as the third rule has a progressing premiss but is not of the form in Definition 5.8.1. Indeed, \(\approx\) is not a congruence: For the predicate \(x = 42\) and the programs \(t_1 = (x := 42; x := 2)\) and \(t_2 = (x := 2)\), we have \(t_1 \approx t_2\) but \(t_1; \text{skip} \not\approx t_2; \text{skip}\).

As indicated above, coolness guarantees congruence for termination semantics:

Theorem 5.11. Termination semantics is compositional for cool stateful SOS specifications.

Proof sketch. Suppose that \(f \in \Sigma\) is an \(n\)-ary operator and \(p_m, q_m \in \mu\Sigma\) are programs with \(p_m \approx q_m\) for \(m = 1, \ldots, n\). By symmetry, it suffices to show the following for all \(s, \pi \in S\):

If \(s, f(p_1, \ldots, p_m)\) terminates in state \(\pi\), then \(s, f(q_1, \ldots, q_m)\) terminates in state \(\pi\).

The proof proceeds by an outer induction on the number of steps until termination of \(s, f(p_1, \ldots, p_m)\) and an inner induction on the structure of the programs. ▶

By Corollary 5.5 we know that for every cool (whence streamlined) specification the kernel of \(\text{trc}: \nu T \to (\nu B)^S\) forms a congruence. Since \(\text{trc}\) is surjective, this means precisely that there is a (unique) \(\Sigma\)-algebra structure on \((\nu B)^S\) for which \(\text{trc}\) is a \(\Sigma\)-algebra homomorphism.

Corollary 5.12. For every cool stateful SOS specification, the kernel of \(\text{fn}^S: (\nu B)^S \to (S + 1)^S\) is a congruence w.r.t. the induced \(\Sigma\)-algebra structure on \((\nu B)^S\).

6 Conclusions and Future Work

We have introduced the stateful SOS rule format for the operational semantics of stateful languages, and equipped it with three semantics: resumption semantics, trace semantics, and termination semantics, in decreasing order of granularity. Our main interest has been in compositionality of these semantics. While resumption semantics is always compositional, it is in general undecidable whether the coarser semantics are compositional. However, compositionality is ensured by restricting to streamlined stateful SOS specifications for trace semantics, and to cool stateful SOS specifications for termination semantics. The compositionality result for the cool format improves on previous results for the similar evaluation-in-context formats [2] by abstracting from steps until termination. The streamlined format is more permissive, as we illustrate on a signal handling construct.

Our results currently work with deterministic state transformers, captured by the functor \(TX = (BX)^S\) where \(BX = S \times (X + 1)\). We believe that our results generalize to functors \(B\) equipped with a natural transformation \(c_X: BX \to S\). As a first step, this generalization requires an abstract characterization of our streamlined and cool rule formats in terms of their corresponding natural transformations, along with categorical proofs of the respective congruence theorems. We leave this as an important point for future work.

A further direction of possible generalization is to cover effects, such as non-determinism, in a similar style as in work on evaluation-in-context [2]. Our work embeds the standard semantics of sequential imperative programming (in particular termination semantics) into the paradigm of operational semantics via distributive laws, and we expect to relate our
results to work on morphisms of distributive laws [40, 20], which, for instance, have recently been shown to have applications to secure compilation [35]. Extending the overall paradigm to support higher-order languages is a well-known and, so far, elusive problem. Like in the current work, tackling this problem may require a slight deviation from the standard form of GSOS laws. It is worth noting that rule formats for higher-order languages have been proposed in the past by Howe [15], Bernstein [6] and more recently Hirschowitz and Lafont [14].

Our treatment of resumption and trace semantics and their relationship is generic, and presumably can be transferred to other settings, in particular to constructive and type-theoretic frameworks. Indeed we expect that it can be implemented relatively directly in foundational proof assistants such as Agda, without additional postulates (such as the axiom of choice or the law of excluded middle). In contrast, the domain \((S + 1)^S\) of termination semantics is inherently classical, as it postulates that every computation will either terminate or diverge. This can be remedied by replacing the maybe-monad \((-) + 1\) with a suitable partiality monad [3, 11]. We will explore to what extent our results regarding termination semantics can be rebased on this more general perspective.

References


