Modular Bisimulation Theory for Computations and Values

Martin Churchill and Peter D. Mosses
{m.d.churchill,p.d.mosses}@swansea.ac.uk

Department of Computer Science, Swansea University, Swansea, UK

Abstract. For structural operational semantics (SOS) of process algebras, various notions of bisimulation have been studied, together with rule formats ensuring that bisimilarity is a congruence. For programming languages, however, SOS generally involves auxiliary entities (e.g. stores) and computed values, and the standard bisimulation and rule formats are not directly applicable.
Here, we first introduce a notion of bisimulation based on the distinction between computations and values, with a corresponding liberal congruence format. We then provide metatheory for a modular variant of SOS (MSOS) which provides a systematic treatment of auxiliary entities. This is based on a higher order form of bisimulation, and we formulate an appropriate congruence format. Finally, we show how algebraic laws can be proved sound for bisimulation with reference only to the (M)SOS rules defining the programming constructs involved in them. Such laws remain sound for languages that involve further constructs.

Keywords: structural operational semantics, programming languages, congruence formats, Modular SOS, higher-order bisimulation.

1 Introduction

Background. Structural operational semantics (SOS) [16] is a well-established framework for specifying computational behaviour, where the behaviour of programs is modelled by labelled transition systems, defined inductively by axioms and inference rules. The metatheory of SOS provides various notions of bisimulation [7,15] for proving behavioural equivalence. Bisimilarity is guaranteed to be a congruence when the rules used to define transition relations are restricted to particular formats, e.g. *tyft/tyxt* [3].

SOS is particularly suitable for specifying process calculi such as CCS: the states of the transition system are simply (closed) process terms, and the labels on transitions represent actions corresponding to steps of process execution. For programming languages, however, transition relations often involve auxiliary entities as arguments, e.g. stores (recording the values of imperative variables before and after transitions) and environments (determining the bindings of currently visible identifiers); they also use terminal states to represent computed values.

These extra features entail that rules do not conform to the usual congruence formats.

The need to specify auxiliary entities in SOS rules also undermines their modularity, which can be a significant pragmatic problem for larger languages. Modular SOS (MSOS) [8] is a simple variant of SOS where auxiliary entities are incorporated in labels. The notation used for label terms in MSOS eliminates references to stores and environments (etc.) in most rules. MSOS provides foundations for the component-based approach to semantics [9] currently being developed by the PLanCompS project (`www.plancomps.org`).

Contribution. In this paper, we introduce a notion of bisimulation for MSOS and a corresponding congruence format supporting auxiliary entities and computations (processes) which compute values. This work stands on the shoulders of previous work on congruence formats used for the SOS of process calculi, but develops it in a direction more suitable for use with programming languages.

Our notion of bisimulation is higher-order, and tailored for use with MSOS; in particular, so-called writeable label components (e.g. the resulting store, or a thrown exception) may vary up to the bisimulation relation in the 'step'. We provide an appropriate rule format – an enhancement of the well-studied *tyft* format [3] – which ensures that bisimilarity is a congruence.

In our setting, there is a strict dichotomy between value terms (which may be inspected) and more general computational terms (which may be run, and their behaviour observed) following e.g. [6]. In particular, the treatment of values is disciplined with fixed built-in rules, and our definition of bisimilarity ensures that bisimilar values have bisimilar subterms. We use silent rewrites to deal with operations on values. This mechanism also allows unit laws to hold for our notion of bisimulation, avoiding the need for weak versions of bisimulation.

MSOS rules for a particular construct need only mention the auxiliary entities relevant to that construct, allowing modular specifications. We describe a complementary notion of *modular bisimulation*, allowing laws to be shown just with respect to the rules relevant to the particular constructs mentioned in that law. This notion of bisimulation identifies when a law is *robust* and cannot be broken by the presence or absence of other constructs that might be in the language. Thus, we can specify the operational semantics of programming languages incrementally, proving laws which remain valid (and a congruence) as the language is extended.

Related work. There has been a variety of work on notions of bisimulation and congruence formats, reviewed in [14]. No formats in this review, or any other we are aware of, allow value terms as arguments in the source of the conclusion.

Our notion of higher-order bisimulation for MSOS generalises stateless bisimulation from [13] (but we allow information flow between the data and process components). Like the higher-order PANTH format of [12], labels may vary up to bisimulation in the step. However, the distinction between readable and writeable MSOS label components admits a much simpler rule format.

We can also compare it to applicative bisimulation [4], in which bisimilar abstractions must yield bisimilar outputs for bisimilar inputs. In our framework, bindings and abstractions can be dealt with using an environment, which is 'just another auxiliary entity' and treated as such. Our notion of bisimulation does not explicitly require that bisimilar environments yield bisimilar computation results, but in fact this is a consequence of our rule format.

Outline. The rest of this paper is arranged as follows: In Sect. 2, we introduce a notion of bisimulation and congruence format focusing on the distinction between values and computations. In Sects. 3 and 4, we lift our notion of bisimulation to the higher-order setting of MSOS, and define a liberal congruence format for bisimulation in MSOS. In Sect. 5, we discuss how we can formulate and prove robust bisimulations, which continue to hold as additional constructs are added to the language. In Sect. 6, we consider further directions. Full proofs of results in this paper are available at `http://www.plancomps.org/churchill2013a/`.

2 Value-Computation Bisimulation

Structural operational semantics uses terms over a first-order algebraic signature as the states and labels of a transition system. A key distinction at the heart of our notions of equivalence is between *computational* terms and *value* terms. Values consist of structure that can be interrogated, while computational terms generally have behaviour. An appropriate slogan from [6] is that values *are*, while computations *do*. Examples of values include Booleans, integers, and closed function abstractions. Computations model the potential behaviour of expressions, statements, declarations, processes and entire programs. This distinction is important in programming languages, c.f. call-by-name vs. call-by-value in Algol or Scala, corresponding to whether functions take computations or values as parameters.

From the point of view of program (term) equivalence, the distinction is also important. In particular, equivalences must be sound with respect to observational tests (contexts). Contexts, as with programs in general, must be able to interrogate values – for example, true must be distinguishable from false. On the other hand, it must not be possible for a program to be able to interrogate the structure of a computational term such as 'if true then C else D', for then it could distinguish it from C and equivalence would reduce to syntactic identity. Thus, only values may be interrogated. But note that values may also contain computational terms as subexpressions. For example, a function value may also contain a body (a computational term) and a closing environment (a value term, which may include computations as substructure).

In this section we formalise this distinction and the corresponding notion of equivalence, and define a simple bisimulation congruence format.

2.1 Value-Computation Transition Systems

The terms we consider are freely generated from an algebraic signature. Rather than quotienting by an equational congruence, we equip our systems with a rewriting relation $\Rightarrow$. This represents internal silent functional transitions. Unlike a transition under a distinguished silent label τ, $\Rightarrow$ is context insensitive (a precongruence). This can be used to avoid polluting traces with silent steps – a goal shared by Plotkin in [16] – and it allows unit laws to hold up to strong bisimulation. Rules can also be kept simple – for example, we can define sequencing using $\frac{s \xrightarrow{l} s'}{\mathsf{seq}(s,t) \xrightarrow{l} \mathsf{seq}(s',t)}$ and $\mathsf{seq}(\mathsf{skip}, s) \Rightarrow s$ where skip represents successful termination of a command. The relation $\Rightarrow$ is asymmetric, and we intend that the RHS is simpler than the LHS – this keeps the search space small in bisimulation proofs and animation.

Our notion of value is derived from that of a value constructor, as in [4].

Definition 1 (value-computation signature). *A* value-computation signature *Σ consists of a set of constructors C_Σ (function symbols – f,g,...), each with an arity $\mathsf{ar}_\Sigma : C_\Sigma \to \mathbb{N}$, and a set of value constructors $VC_\Sigma \subseteq C_\Sigma$. We let T_Σ denote the set of (closed) terms (s,t,...), and $V_\Sigma \subseteq T_\Sigma$ the set of* value terms *whose outermost constructor is in VC_Σ.*

A *precongruence* with respect to Σ is a reflexive transitive relation R such that if $f \in C_\Sigma$ with $\mathsf{ar}(f) = n$ and $s_i \; R \; t_i$ for $1 \leq i \leq n$ then $f(s_1, \ldots, s_n) \; R \; f(t_1, \ldots, t_n)$. For symmetric relations, we may also call such an R a *congruence.*

Definition 2 (value-computation transition system). *A* value-computation transition system *is a tuple $(\Sigma, L, \to, \Rightarrow)$ where Σ is a value-computation signature, L a set of* labels, *$\to \subseteq T_\Sigma \times L \times T_\Sigma$ a* transition relation *and $\Rightarrow \subseteq T_\Sigma \times T_\Sigma$ a* rewriting relation *such that:*

- *$\Rightarrow$ is a precongruence*
- *$s \xrightarrow{l} s'$ implies $s \notin V_\Sigma$ (value terms have no computational behaviour)*
- *$s \Rightarrow s'$ with $s = v(s_1, \ldots, s_n)$ for $v \in VC_\Sigma$ implies $s' = v(s'_1, \ldots, s'_n)$ with $s_i \Rightarrow s'_i$ for $1 \leq i \leq n$ (rewriting preserves value constructors)*
- *If $s \Rightarrow s_1$, $s_1 \xrightarrow{l} s_2$ and $s_2 \Rightarrow s'$ then $s \xrightarrow{l} s'$ (saturation).*

A term made entirely out of value constructors is a *ground value.* Ground values are just as they appear: pure syntactic values, which can be constructed and inspected. The meaning of computational terms in $T_\Sigma - V_\Sigma$ is determined by the $\to$ and $\Rightarrow$ relations, representing their behaviour. Non-ground values can be deconstructed to yield computational terms, which may have behaviour.

A value-computation transition system will typically be specified by a set of inductive rules. If Σ is a value-computation signature, let OT_Σ denote the set of *open Σ-terms*, constructed inductively from *term variables* $(x, y, \ldots)$, *value variables* $(v_1, v_2, \ldots)$ and constructors in C_Σ.

Definition 3 (value-computation specification). *A* value-computation specification *consists of a tuple (Σ, L, D) where Σ is a value-computation signature, L a label set, and D a set of rules over formulas $f(s_1, \dots, s_n) \xrightarrow{l} s'$ or $f(s_1, \dots, s_n) \Rightarrow s'$ with $s_i, s' \in OT_\Sigma$ and $f \notin VC$. These rules generate a transition system over T_Σ and L inductively, after being extended with rules for reflexivity, precongruence, transitivity and saturation (below), where value variables range over value terms.*

$$x \Rightarrow x \qquad \frac{x_1 \Rightarrow y_1 \quad \cdots \quad x_n \Rightarrow y_n}{f(x_1, \dots, y_n) \Rightarrow f(x_1, \dots, y_n)}\; f \in C_\Sigma, \mathsf{ar}_\Sigma(f) = n$$

$$\frac{x \Rightarrow y \qquad y \Rightarrow z}{x \Rightarrow z} \qquad \frac{x \Rightarrow x_1 \qquad x_1 \xrightarrow{l} y_1 \qquad y_1 \Rightarrow y}{x \xrightarrow{l} y}$$

Each such specification generates a value-computation transition system.

Example 4. We consider a value-computation system of basic constructs. The signature Σ contains binary sequencing seq; a ternary conditional cond; nullary constants true, false and skip; operations print_l for $l \in \{a, b\}$; and unary operations thunk for wrapping computations as values, and force for forcing evaluation of a thunk. The value constructors are true, false, skip, and thunk. For labels, $L = \{a, b\}$. The rules are given in Fig. 1.

$$\frac{x \xrightarrow{l} x'}{\mathsf{cond}(x, y_1, y_2) \xrightarrow{l} \mathsf{cond}(x', y_1, y_2)} \quad (1) \qquad \frac{x \xrightarrow{l} x'}{\mathsf{seq}(x, y) \xrightarrow{l} \mathsf{seq}(x', y)} \quad (6)$$

$$\mathsf{cond}(\mathsf{true}, y_1, y_2) \Rightarrow y_1 \quad (2) \qquad \mathsf{seq}(\mathsf{skip}, y) \Rightarrow y \quad (7)$$

$$\mathsf{cond}(\mathsf{false}, y_1, y_2) \Rightarrow y_2 \quad (3) \qquad \frac{x \xrightarrow{l} x'}{\mathsf{force}(x) \xrightarrow{l} \mathsf{force}(x')} \quad (8)$$

$$\mathsf{print}_a \xrightarrow{a} \mathsf{skip} \quad (4)$$

$$\mathsf{print}_b \xrightarrow{b} \mathsf{skip} \quad (5) \qquad \mathsf{force}(\mathsf{thunk}(x)) \Rightarrow x \quad (9)$$

Fig. 1. Operational rules for Example 4

We next introduce our notion of equivalence for value-computation transition systems. This consists of extending the usual bisimulation step condition with two further cases dealing with rewriting and values. For example, if two values are bisimilar, the outermost value constructor must be the same (up to rewriting), and the arguments pointwise bisimilar.

Definition 5 (value-computation bisimulation). *A* value-computation bisimulation *(or* vc-bisimulation*) over a given value-computation transition system $(\Sigma, L, \rightarrow, \Rightarrow)$ is a symmetric relation $R \subseteq T_\Sigma \times T_\Sigma$ such that*

1. *If $s\ R\ t$ and $s \xrightarrow{l} s'$ then $\exists t'$ with $s'\ R\ t'$ and $t \xrightarrow{l} t'$.*
2. *If $s\ R\ t$ and $s \Rightarrow s'$ then $\exists t'$ with $s'\ R\ t'$ and $t \Rightarrow t'$.*
3. *If $v(s_1, \ldots, s_n)\ R\ t$ with $v \in VC$, then $t \Rightarrow v(t_1, \ldots, t_n)$ with $s_i\ R\ t_i$ for $1 \leq i \leq n$.*

Two terms s and t are value-computation bisimilar*, written $s \approx_{vc} t$, if there exists a value-computation bisimulation R with $s\ R\ t$.*

In Example 4, for any terms s, t, r we have $\mathsf{seq}(\mathsf{seq}(s, t), r) \approx_{vc} \mathsf{seq}(s, \mathsf{seq}(t, r))$, and also $\mathsf{thunk}(\mathsf{seq}(\mathsf{seq}(s, t), r)) \approx_{vc} \mathsf{thunk}(\mathsf{seq}(s, \mathsf{seq}(t, r)))$. The use of rewrites $\Rightarrow$ also allows us to prove unit laws up to bisimulation, which usually only hold up to weak bisimulation: for example, $\mathsf{seq}(\mathsf{skip}, s) \approx_{vc} s$.

2.2 Congruence Format

We next define a format guaranteeing that value-computation bisimilarity is a congruence.

Definition 6 (pattern). *A* pattern *is a term constructed inductively from variables and value constructors such that each variable appears at most once.*

Definition 7 (value-added tyft). *A rule is in the* value-added tyft *format if it is of the following shape, where each $\rightsquigarrow$, $\rightsquigarrow_i$ may be $\Rightarrow$ or $\xrightarrow{a}$ for some a.*

$$\frac{\{s_i \rightsquigarrow_i u_i : i \in I\}}{f(w_1, \ldots, w_n) \rightsquigarrow t}$$

Here, t, s_i range over arbitrary open terms; and u_i, w_j over patterns. Further, each variable may occur in at most one of u_i or w_j. A value-computation specification is in the value-added tyft *format if all of its rules are.*

By inspecting Fig. 1, we see that Example 4 is in the value-added tyft format. The restriction of certain subterms to patterns ensures that only value constructors may be inspected. To see that this is necessary for congruence, consider any instance of $\mathsf{seq}(\mathsf{seq}(\mathsf{print}_a, t), r) \approx_{vc} \mathsf{seq}(\mathsf{print}_a, \mathsf{seq}(t, r))$. Then w_i must be a pattern, as otherwise f defined by $\mathsf{f}(\mathsf{seq}(\mathsf{seq}(x, y), z)) \Rightarrow \mathsf{true}$ provides a distinguishing context. Each u_i must be a pattern, as otherwise f defined by $\frac{x \xrightarrow{l} \mathsf{seq}(\mathsf{seq}(x,y),z)}{\mathsf{f}(x) \xrightarrow{l} \mathsf{true}}$ provides a distinguishing context. We require uniqueness of variables as otherwise $\mathsf{g}(x, x) \Rightarrow \mathsf{true}$, $\mathsf{f}(x) \Rightarrow \mathsf{g}(x, \mathsf{seq}(\mathsf{seq}(\mathsf{print}_a, t), r))$ provides a distinguishing context.

The above format is built on the *tyft* format of [3], generalised so that u_i, w_j may range over patterns rather than just variables. In *loc. cit.*, *tyxt* rules are also allowed, where the source of the conclusion is just a variable. We have excluded this here just to ensure that values do not perform computational steps; if this is otherwise guaranteed then *tyxt* rules may be added with congruence intact.

We next show that the value-added tyft format ensures that bisimilarity is a congruence. To show our congruence result, we assume that rules are well-founded: that is, the premises of each rule can be ordered such that variables in the conclusion of a premise appear in no earlier premise. This restriction was also required for the *tyft/tyxt* congruence proof in [3]. It was later shown unnecessary via a translation in [2]; such a translation should be possible for this result also.

If σ is a partial mapping from variables to terms, we write $s[\sigma]$ for the substitution replacing each σ-defined variable x in s by $\sigma(x)$. If s is an open term, we write $\mathsf{vars}(s)$ for the variables occurring in s. The *reflexive congruence closure* of a relation R is the least reflexive relation containing R such that $s_1\ R\ t_1, \ldots, s_n\ R\ t_n$ implies $f(s_1, \ldots, s_n)\ R\ f(t_1, \ldots, t_n)$.

Lemma 8. *Let r be a pattern and σ a substitution with $\mathsf{dom}(\sigma) = \mathsf{vars}(r)$. Let R' denote the reflexive congruence closure of a vc-bisimulation R and let $r[\sigma]\ R'\ t$. Then exists τ with $\mathsf{dom}(\tau) = \mathsf{vars}(r)$ such that $t \Rightarrow r[\tau]$ and for each x, $\sigma(x)\ R'\ \tau(x)$.*

Theorem 9. *If all rules in a value-computation transition system are defined in the value-added tyft format and well-founded, then vc-bisimilarity in that system is a congruence.*

Proof. Let R be a vc-bisimulation, and let R' denote the reflexive congruence closure of R. We will show that R' is also a vc-bisimulation, and since R' contains R we can conclude that vc-bisimilarity is a congruence. To show that R' is a vc-bisimulation, we show the three conditions in Definition 5. Conditions 1 and 2 are shown simultaneously, showing that $s \rightsquigarrow s'$ and $s\ R'\ t$ implies there exists t' with $t \rightsquigarrow t'$ with $s'\ R'\ t'$ for any $\rightsquigarrow$ of the form $\Rightarrow$ or $\xrightarrow{a}$. The proof proceeds by induction on the derivation of $s \rightsquigarrow s'$ making use of the known rule shape, together with Lemma 8 for patterns in the targets of premises and source of conclusion. Condition 3 follows immediately from Lemma 8: If $s = v(s_1, \ldots, s_n)\ R'\ t$ then $s = r[\sigma]$ where $r = v(x_1, \ldots, x_n)$ and $\sigma = \{x_i \mapsto s_i\}$. By the lemma, $t \Rightarrow r[\tau]$ with $\tau = \{x_i \mapsto t_i\}$ and $s_i\ R'\ t_i$. Then $t = v(t_1, \ldots, t_n)$ as required. □

3 Modular SOS

In this section, we first recall the differences between Modular SOS (MSOS) [8] and the original SOS framework [16], explaining how MSOS incorporates auxiliary entities in labels. We then enrich the MSOS specifications of [8] with the notion of value, and illustrate our framework by specifying rules for various constructs.

3.1 MSOS Labels

In SOS (and in the value-computation specifications introduced in Sect. 2) the set of labels can be chosen arbitrarily. In practice, however, when specifying the semantics of concurrent or reactive processes, labels usually represent emitted signals or events; and when specifying sequential programming languages, they are often not used at all. Any auxiliary entities, such as environments (ρ) and stores (σ), are incorporated as sub-terms of states, together with the usual process terms. For example, a state might be a triple (s, ρ, σ).[1]

MSOS differs from SOS by incorporating auxiliary entities in labels, instead of in states. Thus states are simply process terms (including computed values). Moreover, the set of labels forms a *category* (with the labels as the morphisms) and the labels on successive transitions have to be *composable*. The constraint of composability is crucial: for instance, it ensures that the environment in the label does not change between adjacent transitions, and that changes to the store are single-threaded. There is also a notion of *unobservable* label, corresponding to identity morphisms.

Since the various auxiliary entities are (in principle) independent, the label category is obtained as a product of a component category for each auxiliary entity. Following [8] we use indexed products, and write a label using ML-style record value notation as '$\{i_1 = t_1, \ldots, i_n = t_n\}$' (the order in which the components are listed is insignificant).

Three simple kinds of label component category, identified in [8], are sufficient to ensure that MSOS is at least as expressive as SOS:

- *Read-only (RO):* label components are composable only when identical, and always unobservable.
- *Write-only (WO):* label components are always composable, and there is a unique unobservable entity corresponding to an identity morphism.
- *Read-write (RW):* label components are pairs of entities, (x, x') is composable with (y, y') iff x' and y are identical, and (x, x') is unobservable iff x and x' are identical.

For notational convenience we write labels using an unprimed index for each read-only component (e.g. $\mathbf{env}{=}\rho$), a primed index for each write-only component (e.g. $\mathbf{output}'{=}o$), and both an unprimed and a primed index for the two entities of each read-write component (e.g. $\mathbf{store}{=}\sigma_0, \mathbf{store}'{=}\sigma_1$). Formally:

Definition 10 (MSOS labels). *A* label profile *is a triple of disjoint sets $\mathcal{L} = (\mathcal{L}_{RO}, \mathcal{L}_{RW}, \mathcal{L}_{WO})$. The set* $\mathsf{reads}(\mathcal{L})$ *consists of the* unprimed elements $\mathbf{x} \in \mathcal{L}_{RO} \uplus \mathcal{L}_{RW}$*. The set* $\mathsf{writes}(\mathcal{L})$ *consists of the* primed elements $\{\mathbf{x}' : \mathbf{x} \in \mathcal{L}_{WO} \uplus \mathcal{L}_{RW}\}$*. For any set T, the* label set $\mathcal{L}(T)$ *is the set of maps* $\mathsf{reads}(\mathcal{L}) \uplus \mathsf{writes}(\mathcal{L}) \to T$*. For a label $L \in \mathcal{L}(T)$, we write* $\mathsf{reads}(L)$ *and* $\mathsf{writes}(L)$ *for the restriction of L to* $\mathsf{reads}(\mathcal{L})$ *and* $\mathsf{writes}(\mathcal{L})$ *respectively.*

We intend to instantiate T with a set of terms – for example, we can represent stores and environments as terms by using applicative lists. Accordingly, we will use σ, ρ, ... as additional term variables.

3.2 MSOS Specifications

An MSOS specification with respect to a label profile $\mathcal{L}$ and set of terms T generates a transition system specification with states in T and labels in $\mathcal{L}(T)$. Such

[1] Transitions which do not change ρ are usually written $\rho \vdash (s, \sigma) \to (s', \sigma')$.

specifications typically only mention a relevant subset of the label components, treating ellipses '$\ldots$' as variables ranging over the remaining components, which may be propagated between premises and conclusion. A dash '$-$' indicates that the rest of the label L is unobservable: concretely, if $\mathbf{x} \in \mathcal{L}_{RW}$ then $L(\mathbf{x}') = L(\mathbf{x})$ and if $\mathbf{x} \in \mathcal{L}_{WO}$ then $L(\mathbf{x}') = \iota_\mathbf{x}$ where $\iota_\mathbf{x}$ is a distinguished nullary constant associated to $\mathbf{x}$.

Rules may combine labels using *composition* ($\circ$). A pair of labels (L_1, L_2) is *composable* if for $\mathbf{x} \in \mathcal{L}_{RO}$, $L_1(\mathbf{x}) = L_2(\mathbf{x})$ and for $\mathbf{x} \in \mathcal{L}_{RW}$, $L_1(\mathbf{x}') = L_2(\mathbf{x})$. Given a composable pair (L_1, L_2) the *composition* $L_2 \circ L_1$ is defined to be:

- For $\mathbf{x} \in \mathcal{L}_{RO}$, $(L_2 \circ L_1)(\mathbf{x}) = L_1(\mathbf{x}) = L_2(\mathbf{x})$.
- For $\mathbf{x} \in \mathcal{L}_{WO}$, $(L_2 \circ L_1)(\mathbf{x}') = \odot_\mathbf{x}(L_1(\mathbf{x}'), L_2(\mathbf{x}'))$ where $\odot_\mathbf{x}$ is a distinguished binary constructor associated to $\mathbf{x}$. Typically, $(\odot_\mathbf{x}, \iota_\mathbf{x})$ will form a monoid on a subset of the terms.
- For $\mathbf{x} \in \mathcal{L}_{RW}$, $(L_2 \circ L_1)(\mathbf{x}) = L_1(\mathbf{x})$ and $(L_2 \circ L_1)(\mathbf{x}') = L_2(\mathbf{x}')$.

In rules, we use label variables (l, '$\ldots$'). In a given rule, each label must have the same set of explicitly mentioned label components E. Labels in that rule then consist of a map $E \to OT_\Sigma$ denoted by a list of equations, followed by a composition (sequence) of label variables (the empty sequence is denoted '$-$', representing an unobservable label).

Definition 11 (MSOS specification). *An* MSOS specification *consists of a tuple $(\mathcal{L}, \Sigma, D, M)$ where $\mathcal{L}$ is a label profile, Σ a value-computation signature, and D is a set of rules over formulas $f(s_1, \ldots, s_n) \xrightarrow{l} s'$ or $f(s_1, \ldots, s_n) \Rightarrow s'$ with $s_i, s' \in OT_\Sigma$, $f \notin VC_\Sigma$ with labels as immediately above. Finally, M specifies for each $\mathbf{x} \in \mathcal{L}_{WO}$ a nullary $\iota_\mathbf{x} \in C_\Sigma$ and binary $\odot_\mathbf{x} \in C_\Sigma$. There are* built-in *rules for reflexivity, precongruence, transitivity and saturation, consisting of those in Definition 3 in addition to:*

$$\frac{x \Rightarrow z \qquad y \xrightarrow{\{\mathbf{x}=z,\ldots\}} t'}{y \xrightarrow{\{\mathbf{x}=x,\ldots\}} t'} \qquad \frac{y \xrightarrow{\{\mathbf{x}'=x,\ldots\}} t' \qquad x \Rightarrow z}{y \xrightarrow{\{\mathbf{x}'=z,\ldots\}} t'}$$

An MSOS specification generates a value-computation transition system over $\mathcal{L}(T_\Sigma)$ inductively after being extended with the built-in rules, where value variables range over value terms. Unobservability, composability and composition are interpreted as described above.

Example 12. In Fig. 2, we give some example constructors and their MSOS rules. The label profile includes read-only **env**, write-only **exc** and **output**, and read-writeable **store**. We include all constructors and rules from Example 4, except for print, which has been generalised.

We add value constructors for maps (ternary update, nullary empty), lists (binary cons, nullary nil) and function values (ternary abs). The value $\mathsf{abs}(x, s, \rho)$ denotes a closed function, with formal parameter x, body s, and closing environment ρ. We include a set of nullary values $I = \{\mathsf{x}, \mathsf{y}, \ldots\}$ for identifiers and imperative variables.

$$\mathsf{bound}(x) \xrightarrow{\{\mathbf{env}=\rho,\ldots\}} \mathsf{lookup}(\rho, x) \tag{10}$$

$$\frac{y \xrightarrow{\{\ldots\}} y'}{\mathsf{let}(x, y, t) \xrightarrow{\{\ldots\}} \mathsf{let}(x, y', t)} \tag{11}$$

$$\frac{y \xrightarrow{\{\mathbf{env}=\mathsf{update}(\rho,x,v),\ldots\}} y'}{\mathsf{let}(x, v, y) \xrightarrow{\{\mathbf{env}=\rho,\ldots\}} \mathsf{let}(x, v, y')} \tag{12}$$

$$\mathsf{let}(x, v_1, v_2) \Rightarrow v_2 \tag{13}$$

$$\mathsf{throw}(x) \xrightarrow{\{\mathbf{exc}'=\mathsf{cons}(x,\mathsf{nil}),-\}} \mathsf{stuck} \tag{14}$$

$$\frac{x \xrightarrow{\{\mathbf{exc}'=\mathsf{nil},\ldots\}} x'}{\mathsf{catch}(x, z) \xrightarrow{\{\mathbf{exc}'=\mathsf{nil},\ldots\}} \mathsf{catch}(x', z)} \tag{15}$$

$$\frac{x \xrightarrow{\{\mathbf{exc}'=\mathsf{cons}(y,\mathsf{nil}),\ldots\}} x'}{\mathsf{catch}(x, z) \xrightarrow{\{\mathbf{exc}'=\mathsf{nil},\ldots\}} \mathsf{apply}(z, y)} \tag{16}$$

$$\mathsf{catch}(v, z) \Rightarrow v \tag{17}$$

$$\mathsf{assign}(x, v) \xrightarrow{\substack{\{\,\mathbf{store} = \sigma, \\ \mathbf{store}' = \mathsf{update}(\sigma, x, v), -\}}} \mathsf{skip} \tag{18}$$

$$\frac{y \xrightarrow{\{\ldots\}} y'}{\mathsf{assign}(x, y) \xrightarrow{\{\ldots\}} \mathsf{assign}(x, y')} \tag{19}$$

$$\mathsf{deref}(x) \xrightarrow{\{\mathbf{store}=\sigma,-\}} \mathsf{lookup}(\sigma, x) \tag{20}$$

$$\mathsf{print}(x) \xrightarrow{\{\mathbf{output}'=\mathsf{cons}(x,\mathsf{nil}),-\}} \mathsf{skip} \tag{21}$$

$$\frac{x \xrightarrow{\{\ldots\}} x'}{\mathsf{apply}(x, y) \xrightarrow{\{\ldots\}} \mathsf{apply}(x', y)} \tag{22}$$

$$\frac{y \xrightarrow{\{\ldots\}} y'}{\mathsf{apply}(v, y) \xrightarrow{\{\ldots\}} \mathsf{apply}(v, y')} \tag{23}$$

$$\frac{y \xrightarrow{\{\mathbf{env}=\mathsf{update}(\rho,x,v),\ldots\}} y'}{\mathsf{apply}(\mathsf{abs}(x, y, \rho), v) \xrightarrow{\{\mathbf{env}=\rho_1,\ldots\}} \mathsf{apply}(\mathsf{abs}(x, y', \rho), v)} \tag{24}$$

$$\mathsf{apply}(\mathsf{abs}(x, v_1, \rho), v_2) \Rightarrow v_1 \tag{25}$$

$$\mathsf{lambda}(x, y) \xrightarrow{\{\mathbf{env}=\rho,\ldots\}} \mathsf{abs}(x, y, \rho) \tag{26}$$

$$\frac{x \xrightarrow{l_1} x' \qquad \mathsf{atomic}(x') \xrightarrow{l_2} v}{\mathsf{atomic}(x) \xrightarrow{l_2 \circ l_1} v} \tag{27}$$

$$\mathsf{atomic}(v) \xrightarrow{\{-\}} v \tag{28}$$

$$\mathsf{append}(\mathsf{cons}(x, y), z) \Rightarrow \mathsf{cons}(x, \mathsf{append}(y, z)) \tag{29}$$

$$\mathsf{append}(\mathsf{nil}, x) \Rightarrow x \tag{30}$$

$$\frac{\mathsf{lookup}(\mu, j) \Rightarrow v_1}{\mathsf{lookup}(\mathsf{update}(\mu, i, v), j) \Rightarrow v_1}\; i \neq j \in I \tag{31}$$

$$\frac{}{\mathsf{lookup}(\mathsf{update}(\mu, i, v), i) \Rightarrow v}\; i \in I \tag{32}$$

Fig. 2. Operational rules for Example 12

Additional computational constructors include static bindings (let and bound), volatile store (assign and deref), functions (lambda and apply) and exceptions (throw and catch). The term $\mathsf{let}(x, s, t)$ binds x to s in t.

We include binary operations lookup and append for maps and lists respectively. Note that $\mathsf{lookup}(\mathsf{empty}, \mathsf{x})$ is a stuck computational term – this is an example of how undefinedness can be handled in our setting.

We also include an atomic constructor. The computation $\mathsf{atomic}(s)$ runs s and combines the trace into a single transition. This can be used to block the context interrupting the trace, e.g. by catching a thrown exception. Note that the first rule for atomic only applies when (l_1, l_2) is a composable pair.

We set $\iota_{\mathbf{output}} = \iota_{\mathbf{exc}} = \mathsf{nil}$ and $\odot_{\mathbf{output}} = \odot_{\mathbf{exc}} = \mathsf{append}$.

The notational burden of heavily loaded arrows can be avoided by writing the MSOS rules using conventional SOS notation, following techniques in [11].

4 Bisimulation Metatheory for MSOS

We next revisit our goal of ensuring that bisimilarity is a congruence, this time in the MSOS setting. Even though we still generate value-computation transition systems, the value-added tyft format is of limited use, since it does not allow information flow between labels and other computational terms in rules (this is needed in Example 12 for e.g. Rule (12) for let). Note that we cannot allow such flow arbitrarily: if $s_1 \approx s_2$ is to imply $\mathsf{let}(\mathsf{x}, s_1, t) \approx \mathsf{let}(\mathsf{x}, s_2, t)$ then t can only test the **env** label component up to pointwise bisimilarity.

4.1 Bisimulation in MSOS

We next generalise vc-bisimulation to a higher-order version for the MSOS setting. In particular, in the step writeable label components may themselves vary up to bisimulation. This is required, for example, so that $s \approx t$ implies $\mathsf{assign}(\mathsf{x}, s) \approx \mathsf{assign}(\mathsf{x}, t)$. Given a relation R, for maps σ and τ we write $\sigma\ R\ \tau$ just if $\mathsf{dom}(\sigma) = \mathsf{dom}(\tau)$ and for each $x \in \mathsf{dom}(\sigma)$, $\sigma(x)\ R\ \tau(x)$.

Definition 13 (MSOS bisimulation). *Given a value-computation transition system $(\Sigma, \mathcal{L}(T_\Sigma), \to, \Rightarrow)$ generated from an MSOS specification, an* MSOS bisimulation *is a symmetric relation $R \subseteq T_\Sigma \times T_\Sigma$ such that:*

1. *If $s\ R\ t$ and $s \xrightarrow{L} s'$ then $\exists t', L'$ with $s'\ R\ t'$, $t \xrightarrow{L'} t'$, $\mathsf{reads}(L') = \mathsf{reads}(L)$ and $\mathsf{writes}(L)\ R\ \mathsf{writes}(L')$.*
2. *If $s\ R\ t$ and $s \Rightarrow s'$ then $\exists t'$ with $s'\ R\ t'$ and $t \Rightarrow t'$.*
3. *If $v(s_1, \ldots, s_n)\ R\ t$ with $v \in VC_\Sigma$, then $t \Rightarrow v(t_1, \ldots, t_n)$ with $s_i\ R\ t_i$ for $1 \leq i \leq n$.*

Two terms s and t are MSOS bisimilar, *written $s \approx_{msos} t$, if there exists an MSOS bisimulation R with $s\ R\ t$.*

In MSOS rules, usually only a few label components are mentioned explicitly, while in the above definition all label components are mentioned. However, in any particular bisimulation proof, one can set $L'(i) = L(i)$ for unmentioned i.

Since vc-bisimulations are also MSOS bisimulations, the associativity and unit laws for seq hold up to MSOS bisimilarity in Example 12. We can also show $\mathsf{catch}(\mathsf{print}(v), x) \approx_{msos} \mathsf{print}(v)$, for example. We may seek to prove laws for state such as $\mathsf{seq}(\mathsf{assign}(\mathsf{x}, v), \mathsf{deref}(\mathsf{x})) \approx_{msos} \mathsf{seq}(\mathsf{assign}(\mathsf{x}, v), v)$. However, this law is not sound with respect to arbitrary contexts. In particular, $C[s]$ may run one step of computation of s and then roll back the store before continuing. Instead, we may prove a modified law which blocks interruption of the trace:

$$\mathsf{atomic}(\mathsf{seq}(\mathsf{assign}(\mathsf{x}, v), \mathsf{deref}(\mathsf{x}))) \approx_{msos} \mathsf{atomic}(\mathsf{seq}(\mathsf{assign}(\mathsf{x}, v), v)).$$

4.2 Congruence Format

We now present a rule format which ensures that MSOS bisimilarity is a congruence. We will need to consider the substructure of labels in rules.

Definition 14 (well-founded MSOS tyft). *A rule is in the* well-founded MSOS tyft *format if it has the following form:*

$$\frac{\{s_i \rightsquigarrow_i u_i : i \in I\}}{f(w_1, \ldots, w_n) \rightsquigarrow t}$$

where premises are ordered and:

- *t, s_i range over arbitrary open terms and u_i, w_j over patterns.*
- *$\rightsquigarrow_i$ is either $\Rightarrow$ or $\xrightarrow{L_i}$ where L_i consists of a sequence of equations $\{l = t_{l,i}\}$ possibly followed by a label variable. Further, if l is primed then $t_{l,i}$ must be a pattern.*
- *$\rightsquigarrow$ is either $\Rightarrow$ or $\xrightarrow{L}$, where L consists of a sequence of equations $\{l = t_l\}$ possibly followed by a composition of label variables. Each such label variable must occur in the premise. If label variable X is to the left of label variable Y in the composition, Y must occur in an earlier premise than X. Further, if l is unprimed then t_l is a pattern.*
- *The set of variables must be disjoint for u_i, w_j, $t_{l,i}$ for primed l, t_l for unprimed l. Variables in u_i or $t_{l,i}$ for primed l must not appear in an earlier premise.*

An MSOS specification is in the well-founded MSOS tyft format just if all its rules are.

This follows a discipline of information flow from readable components in the conclusion to readable components in the premise, to writeable components in the premise to writeable components in the conclusion. Writeable components ($t_{l,i}$ and t_l for primed l) are treated like additional targets, and readable components ($t_{l,i}$ and t_l for unprimed l) like additional sources. To see why each t_l

must be a pattern for unprimed l, consider $\mathsf{g} \xrightarrow{\{\mathbf{env}=\mathsf{update}(y,\mathsf{x},\mathsf{thunk}(\mathsf{seq}(z,w)))\}} \mathsf{true}$, $\mathsf{f}(y) \Rightarrow \mathsf{let}(\mathsf{x}, \mathsf{thunk}(y), \mathsf{g})$ and note that f provides a distinguishing context for $\mathsf{seq}(\mathsf{skip}, \mathsf{print}(\mathsf{true})) \approx_{msos} \mathsf{print}(\mathsf{true})$. To see why each $t_{l,i}$ must be a pattern for primed l, note that $\frac{\mathsf{throw}(x) \xrightarrow{\{\mathbf{exn}'=\mathsf{cons}(\mathsf{seq}(z,w),\mathsf{nil}),\dots\}} y}{\mathsf{f}(x) \xrightarrow{\{\mathbf{exn}'=\mathsf{nil},\dots\}} \mathsf{true}}$ provides a distinguishing context for the same equation. The same examples given in Sect. 2.2 show why the u_i, w_j must be patterns and why variables may not be shared.

Note that in this format composition expressions and unobservable labels may only occur in the conclusion of a rule. The restriction on ordering of label variables in the conclusion ensures that when composition is made explicit, the pattern restrictions above are satisfied.

The distinction between readable and writeable label components is related to label arguments in [1] and the notion of *volatility* from [12]. In each, certain terms in the label are restricted to be a generalised notion of fresh variable and replacement of bisimilar terms in this component will lead to bisimilar outputs.

By inspection of Figs. 1 and 2 we see that Example 12 is in MSOS tyft format (we view the rules for lookup as a family of rules indexed over I). In the rest of this section we will show that for systems with rules in the well-founded MSOS tyft format, MSOS bisimilarity is a congruence.

Definition 15 (explicit MSOS tyft). *An MSOS specification is in* explicit MSOS tyft format *if it is in the well-founded MSOS tyft format and contains no label variables.*

Proposition 16. *Each well-founded MSOS tyft system is equivalent to one in the explicit MSOS tyft format.*

Given an MSOS transition system T over label profile $\mathcal{L}$ we produce an equivalent set of rules removing all uses of label variables, exhibiting all information flow in labels explicitly following the definitions in Sect. 3. We only give an example. The rules for the atomic constructor are translated as follows:

$$\frac{s \xrightarrow{\lambda_1} s' \qquad \mathsf{atomic}(s') \xrightarrow{\lambda_2} v}{\mathsf{atomic}(s) \xrightarrow{\lambda} v} \qquad \mathsf{atomic}(v) \xrightarrow{\mu} v$$

where

$$\begin{aligned}
\lambda_1 &= \{\mathbf{env} = \rho, \mathbf{store} = \sigma_1, \mathbf{store}' = \sigma_2, \mathbf{output}' = \alpha_1, \mathbf{exc}' = \eta_1\} \\
\lambda_2 &= \{\mathbf{env} = \rho, \mathbf{store} = \sigma_2, \mathbf{store}' = \sigma_3, \mathbf{output}' = \alpha_2, \mathbf{exc}' = \eta_2\} \\
\lambda &= \{\mathbf{env} = \rho, \mathbf{store} = \sigma_1, \mathbf{store}' = \sigma_3, \mathbf{output}' = \mathsf{append}(\alpha_1, \alpha_2), \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{exc}' = \mathsf{append}(\eta_1, \eta_2)\} \\
\mu &= \{\mathbf{env} = \rho, \mathbf{store} = \sigma, \ \mathbf{store}' = \sigma, \ \mathbf{output}' = \mathsf{nil}, \mathbf{exc}' = \mathsf{nil}\}
\end{aligned}$$

Proposition 17. *Consider an MSOS specification in explicit MSOS tyft format. Let R be an MSOS bisimulation over the generated transition system and let R' denote the reflexive transitive congruence closure of R. Suppose $s \; R' \; t$. Then:*

1. *If $s = r[\sigma]$ with $\mathsf{dom}(\sigma) = \mathsf{vars}(r)$ and r is a pattern then there exists τ with $\mathsf{dom}(\tau) = \mathsf{vars}(r)$ such that $t \Rightarrow r[\tau]$ with $\sigma(x)\ R'\ \tau(x)$ for each $x \in \mathsf{vars}(r)$.*
2. *If $s \Rightarrow s'$ and $s\ R'\ t$ then there exists t' with $t \Rightarrow t'$ and $s'\ R'\ t'$.*
3. *If $s \xrightarrow{L} s'$ and $\mathsf{reads}(L)\ R'\ trs$ then there exists t', tws such that $s'\ R'\ t'$, $\mathsf{writes}(L)\ R'\ tws$ and $t \xrightarrow{L'} t'$ for $\mathsf{reads}(L') = trs$ and $\mathsf{writes}(L') = tws$.*

Proof. The proof proceeds by simultaneous induction on R'. Condition 1 corresponds to Lemma 8 but must be proved simultaneously due to the fact that we also close our relation up to transitivity. For Conditions 2 and 3, we perform an inner induction on the proof of the transition, exploiting the rule format. □

Theorem 18. *Consider an MSOS specification T in the well-founded MSOS tyft format. Let R be an MSOS bisimulation and R' denote the reflexive transitive congruence closure of R. Then R' is an MSOS bisimulation.*

Proof. We first convert T into an equivalent system in explicit MSOS tyft format following Proposition 16. We then show that R' is an MSOS bisimulation by considering the three conditions in turn, each of which follows straightforwardly from Proposition 17. □

Corollary 19. *MSOS-bisimilarity is a congruence for specifications in the well-founded MSOS tyft format.*

Proposition 17 claim 3 ensures that if $s \approx_{msos} t$ then each composable trace from s can be matched by a corresponding composable trace from t, up to bisimilarity in the subsequent steps and labels. In the case of Example 12, it also has the following consequence: for any term, (pointwise) bisimilar environments yield bisimilar outputs. The fact that bisimilarity is a congruence ensures that bisimilar abstractions yield bisimilar outputs when applied. This is part of the definition of bisimilar abstractions in applicative bisimulation [4].

5 Modular Bisimulations

Bisimulation examples in this paper were given explicitly with respect to our example systems. But in fact the proofs did not make use of the particular closed set of constructors. For example, for $\mathsf{seq}(\mathsf{skip}, s) \approx s$, presence of constructors other than seq and skip had no influence whatsoever on the proof of bisimulation. The proof would work just as well in any system with those rules for the constructors in question; the law and proof are *modular* in nature. On the other hand, if a bisimulation proof performs explicit case analysis on all terms or label components, this is not possible. How can we formalise this distinction?

Given a constructor f, an *f-defining rule* is a rule where the source of the conclusion has f as its outermost symbol. A *disjoint extension* of an MSOS system $(\Sigma, \mathcal{L}, D)$ is an MSOS system $(\Sigma', \mathcal{L}', D')$ with $\Sigma' \supseteq \Sigma$, $\mathcal{L}' \supseteq \mathcal{L}$ and such that each rule in $D' - D$ is f-defining for some f in $\Sigma' - \Sigma$.

Let S be a subset of the constructors of Example 12. We define $\hat{S}$ to be the least set containing S such that for all f in $\hat{S}$, any constructor appearing in an f-defining rule also appears in $\hat{S}$. We define $\mathbf{E}_S$ to be the subsystem of Example 12 restricted to the constructors in $\hat{S}$ and rules that are f-defining for some $f \in \hat{S}$. Given a candidate algebraic law for MSOS, we advocate proving this law with respect to all disjoint extensions of $\mathbf{E}_S$, where S is the set of constructors appearing in that law. We isolate the particular subsystem that makes the law hold, and are guaranteed that any system containing this will validate the law. For associativity of seq, we show: $\mathsf{seq}(\mathsf{seq}(s,t),r) \approx_{msos} \mathsf{seq}(s,\mathsf{seq}(t,r))$ in any disjoint extension of $\mathbf{E}_{\{\mathsf{seq}\}}$.

We call such statements *modular bisimulations.* Since the quantification over extensions is external to the particular notion of bisimulation, meta-results such as congruence can be used directly. All examples of bisimulations in this paper can indeed be formulated and proved as modular bisimulations.

If we wish to internalise this notion, we are led to *fh-bisimulation* [10]. In this setting, the step conditions must hold in the presence of arbitrary *hypotheses*, of the form $x \xrightarrow{a} y$ for variables x, y. More specifically, it is *provable ruloids* that must step – a provable ruloid of $\frac{\Gamma}{s \xrightarrow{a} s'}$ is a proof of $s \xrightarrow{a} s'$ which may have open leaves found in Γ. In [10], it was shown that fh-bisimilarity is preserved under disjoint extensions which preserve the label set for the positive GSOS format. (We have subsequently generalised this result to arbitrary positive source-dependent rules.) In future, we hope to adapt these results to our MSOS bisimulation format.

6 Further Directions

We intend to use our framework to give formal semantics, and prove laws about, real-world programming languages. One reason this has been lacking in the literature is due to the scalability of the usual techniques, and our use of MSOS and modular bisimulations help to address these issues. As a start, we are currently providing dynamic semantics for Caml Light [5] by translating it into the kind of basic constructors found in Example 12, called *funcons*[2] [9]. This includes higher-order functions, pattern matching, records and variants, mutually recursive declarations, exceptions and reference cells. Crucially, all rules for constructors used are in the MSOS tyft format. Thus, if program fragments P and Q have funcon translations P' and Q' respectively, and P' and Q' have been proved equivalent using our techniques, we can conclude that P and Q are equivalent and soundly interchangeable in Caml Light programs.

A possible useful extension of this work would be treatment of multisorted algebras. In particular, the right unit law for `seq` only holds if the only value that left operand could compute to is `skip`, i.e. it has type `unit`, or is a command. We could also consider parametrising bisimulations by the current label components cf. state-based bisimilarity [13], which would increase the number of equivalences one could prove. This could be particularly interesting in the MSOS setting, where labels are both open and higher-order.

Another further direction is to consider rules with negative premises, which we have avoided here by matching on values. We have also avoided special treatment

[2] See `http://www.plancomps.org/churchill2013a/`

for variable binders/names, which are handled by the environment. Fresh name generation is possible using read-write label components.

Acknowledgements. Many thanks to Mohammad Mousavi, Cristian Prisacariu, Paolo Torrini 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. Bernstein, K.L.: A congruence theorem for structured operational semantics of higher-order languages. In: 13th Annual IEEE Symposium on Logic in Computer Science. pp. 153–164. IEEE (1998)
2. Fokkink, W.: The tyft/tyxt format reduces to tree rules. In: Hagiya, M., Mitchell, J.C. (eds.) TACS'94. pp. 440–453. LNCS, Vol. 789, Springer, Heidelberg (1994)
3. Groote, J.F., Vaandrager, F.: Structured operational semantics and bisimulation as a congruence. Inf. and Comput. 100(2), 202 – 260 (1992)
4. Howe, D.J.: Equality in lazy computation systems. In: Fourth Annual IEEE Symposium on Logic in Computer Science. pp. 198–203. IEEE (1989)
5. Leroy, X.: The Caml Light system, documentation and user's guide. `http://caml.inria.fr/pub/docs/manual-caml-light/` (1997)
6. Levy, P.B.: Call-by-push-value: A subsuming paradigm. In: Girard, J.Y. (ed.) TLCA'99. pp. 228–242. LNCS, Vol. 1581, Springer, Heidelberg (1999)
7. Milner, R.: A Calculus of Communicating Systems. LNCS, Vol. 92, Springer, Heidelberg (1980)
8. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. 60-61, 195–228 (2004)
9. Mosses, P.D.: Component-based semantics. In: Huisman, M. (ed.) Eighth Intl. Workshop on Specification and Verification of Component-Based Systems. pp. 3–10. ACM, New York (2009)
10. Mosses, P.D., Mousavi, M.R., Reniers, M.A.: Robustness of equations under operational extensions. In: Fröschle, S., Valencia, F.D. (eds.) 17th International Workshop on Expressiveness in Concurrency. pp. 106–120. EPTCS, Vol. 41, arXiv (2010)
11. Mosses, P.D., New, M.J.: Implicit propagation in structural operational semantics. In: Hennessy, M., Klin, B. (eds.) Fifth Workshop on Structural Operational Semantics. pp. 49–66. Electr. Notes Theor. Comput. Sci., Vol. 229(4), Elsevier, Amsterdam (2009)
12. Mousavi, M.R., Gabbay, M., Reniers, M.: SOS for higher order processes. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. pp. 308–322. LNCS, Vol. 3653, Springer, Heidelberg (2005)
13. Mousavi, M.R., Reniers, M.A., Groote, J.F.: Notions of bisimulation and congruence formats for SOS with data. Inf. and Comput. 200(1), 107 – 147 (2005)
14. Mousavi, M.R., Reniers, M.A., Groote, J.F.: SOS formats and meta-theory: 20 years after. Theor. Comput. Sci. 373(3), 238 – 272 (2007)
15. Park, D.: Concurrency and automata on infinite sequences. In: Proc. 5th GI-Conference on Theoretical Computer Science. pp. 167–183. Springer, London, UK (1981)
16. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004), (Originally Tech. Rep. DAIMI FN-19, Dept. of Computer Science, Univ. Aarhus, 1981)