Implicit Propagation in Structural Operational Semantics

Peter D. Mosses[1] Mark J. New[2]

Department of Computer Science
Swansea University
Swansea, UK

Abstract

In contrast to a transition system specification in process algebra, a structural operational semantics (SOS) of a programming language usually involves auxiliary entities: stores, environments, etc. When specifying SOS rules, particular auxiliary entities often need to be propagated unchanged between premises and conclusions. The standard technique is to make such propagation explicit, using variables. However, referring to all entities that need to be propagated *unchanged* in each rule can be tedious, and it hinders direct reuse of rules in different language descriptions.

This paper proposes a new interpretation of SOS rules, such that each auxiliary entity is *implicitly* propagated in all rules in which it is not mentioned. The main benefits include significant notational simplification of SOS rules and much-improved reusability. This new interpretation of SOS rules is based on the same foundations as Modular SOS, but avoids the notational overhead of grouping auxiliary entities together in labels.

After motivating and explaining implicit propagation, the paper considers the foundations of SOS and Modular SOS specifications, and defines the meaning of SOS specifications with implicit propagation by translating them to Modular SOS. It then shows how implicit propagation can simplify various rules found in the SOS literature.

Keywords: formal semantics, structural operational semantics, Modular SOS, modularity, reuse

1 Introduction

Structural operational semantics (SOS) is a well-known framework for describing both static and dynamic semantics of programming and specification languages. This paper is about how its pragmatic aspects can be significantly improved, and assumes familiarity with the standard framework (see e.g. [1,8,22,24]).

1.1 Explicit Propagation

Sometimes, the SOS description of particular constructs requires the introduction of *auxiliary entities*. For instance, the SOS of assignment commands requires stores,

[1] Email: p.d.mosses@swan.ac.uk, web pages: www.cs.swan.ac.uk/~cspdm/
[2] Email: csmarkn@swan.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

representing the values last assigned to variables; the SOS of blocks with local declarations may involve environments, representing the current bindings of identifiers.

For many constructs, however, their SOS description does not itself require the introduction of any auxiliary entities at all: the rules merely need to *propagate* the entities that have been introduced for use in the description of *other* constructs. Such propagation is usually specified explicitly in SOS rules by the repeated use of variables that range over the auxiliary entities. For example, consider the following SOS rule for the dynamic semantics of command sequencing '$c_1 \,;\, c_2$' in a language that contains assignments and local declarations (but no synchronisation):

$$\frac{\rho \vdash \langle c_1, \sigma \rangle \longrightarrow \langle c_1', \sigma' \rangle}{\rho \vdash \langle c_1 \,;\, c_2, \sigma \rangle \longrightarrow \langle c_1' \,;\, c_2, \sigma' \rangle} \tag{1}$$

The repeated use of the environment variable ρ above ensures that the current bindings for c_1 are the same as those for '$c_1 \,;\, c_2$', and similarly for the store variable σ, whereas the repetition of σ' ensures that any change made by c_1 to the store is also made by '$c_1 \,;\, c_2$'.

Explicit propagation of auxiliary entities can be quite tedious – not only for the writer of a language description, but also for its readers. The same patterns of propagation are reiterated in the SOS rules for many different constructs. The repeated variables often dominate the rules, distracting attention from the description of the essence of the described constructs. A missing subscript or prime on a variable can lead to erroneous (i.e. unintended) semantics.

A particularly unfortunate consequence of explicit propagation in SOS is that it often prevents direct *reuse* of the rules for a common programming construct when it occurs in different languages: different sets of rules for the same construct may be needed, according to which *other* constructs are included in the described language. In fact many programming constructs (command sequencing, conditionals, loops, blocks, etc.) are common to large groups of languages; relatively few constructs are included in only a single language.

The need for multiple versions of SOS rules for the same construct motivated the development of the TinkerType tool [11]. In TinkerType, each version of the rules for a construct (or a set of related constructs) is tagged with a set of 'features'. The tool supports extraction of compatible sets of rules for a collection of constructs according to their combined features. Pierce found the TinkerType tool useful and appropriate for managing multiple descriptions of constructs in SOS, and used it in connection with producing his book on *Types and Programming Languages* [23]; but TinkerType does not reduce the *need* for multiple descriptions in SOS at all.

Modular SOS (MSOS) [15,16,17] is a simple variant of SOS which appears to *completely* eliminate the need for multiple descriptions of individual constructs. [3] It achieves this by allowing all auxiliary entities to be propagated *together* in labels on transitions, which then include entities such as environments and (pairs of) stores as well as the usual actions. Constructs such as sequencing, which do not themselves require the introduction of auxiliary entities, can be described using a

[3] Adding new aspects to the semantics of a construct, such as a measure of its execution time, does not preserve semantic equivalence, and is regarded as describing a new construct.

single variable, say X, ranging over *arbitrary* labels, to propagate all the auxiliary entities together – irrespective of what they are. For example, SOS rule (1) above can be formulated thus in MSOS:

$$\frac{c_1 \xrightarrow{X} c_1'}{c_1 \,;\, c_2 \xrightarrow{X} c_1' \,;\, c_2} \tag{2}$$

Labels in MSOS are actually record values, and their components can be accessed, when required, using (loose) record patterns of the form $\{i{=}v, \ldots\}$. [4]

In effect, MSOS reduces explicit propagation to the repeated use of a single variable; it does not eliminate it completely. Moreover, it requires *all* transitions to be explicitly labelled, which, although normal practice in specifications of transition systems for process algebra, is quite unusual in SOS descriptions of programming languages, and a potential hindrance to migration from SOS to MSOS. The Modular Rewriting Semantics framework [12] exploits records in a similar way to combine and propagate auxiliary entities, although there the records are components of states, rather than labels.

1.2 Implicit Propagation

One of the (few) examples of an SOS for a complete major programming language is *The Definition of Standard ML* [14]. Standard ML is primarily a language for functional programming, but it also supports the use of imperative variables. The definition of the semantics of Standard ML is formulated in the big-step style. In the dynamic semantics, the formula $\sigma, \rho \vdash e \Rightarrow v, \sigma'$ specifies that evaluation of an expression e with environment ρ and store σ results in a value v and store σ'. In most of the rules used to define the dynamic semantics, the stores σ and σ' are omitted, and the following convention for reinserting them is introduced:

$$\frac{\rho_1 \vdash e_1 \Rightarrow v_1 \quad \cdots \quad \rho_n \vdash e_n \Rightarrow v_n}{\rho \vdash e \Rightarrow v} \tag{3}$$

abbreviates

$$\frac{\sigma_0, \rho_1 \vdash e_1 \Rightarrow v_1, \sigma_1 \quad \cdots \quad \sigma_{n-1}, \rho_n \vdash e_n \Rightarrow v_n, \sigma_n}{\sigma_0, \rho \vdash e \Rightarrow v, \sigma_n} \,. \tag{4}$$

This convention is called the *state convention*, and corresponds to implicit propagation of stores in SOS – between premises, as well as between particular premises and the conclusion. Notice that the order in which the premises are written in the abbreviated rule determines the (intended) order of evaluation of the sub-expressions $e_1, \ldots, e_n$, in contrast to the usual treatment of premises as an unordered set. A further convention, called the *exception convention*, is introduced to provide implicit propagation of exceptions; but propagation of the environment is left explicit. The introduction of the two conventions seems somewhat ad hoc, and it appears that they have not been adopted in SOS descriptions of other languages.

[4] As in Standard ML, our record pattern notation involves explicit use of '...' as a formal symbol, although here we treat it as a variable, rather than as a wild card.

Significantly more systematic conventions allowing implicit propagation in small-step SOS are introduced by Cenciarelli et al. in their work on the Java memory model [6,7]. In their ESOP 2007 paper they write:

> "*Rule conventions.* In writing an axiom $\gamma_1 \to \gamma_2$ we focus only on the relevant parts of the configurations involved, and understand that whatever is omitted from γ_1 remains unchanged in γ_2. For example, we understand that the axiom '$; p \to p$' stands for $T \parallel (\theta, ; p), \eta \to T \parallel (\theta, p), \eta$. On the other hand, rules with a premise are read by assuming that whatever changes occur in the omitted parts of the premise also occur in the conclusion. For example, we understand that:
> $\dfrac{e_1 \to e_2}{e_1 \,\mathtt{op}\, e \to e_2 \,\mathtt{op}\, e}$ means $\dfrac{T_1 \parallel (\theta, (e_1)_{\delta_1}), \eta_1 \to T_2 \parallel (\theta, (e_2)_{\delta_2}), \eta_2}{T_1 \parallel (\theta, (e_1 \,\mathtt{op}\, e)_{\delta_1}), \eta_1 \to T_2 \parallel (\theta, (e_2 \,\mathtt{op}\, e)_{\delta_2}), \eta_2}$."

As with *The Definition of Standard ML*, Cenciarelli et al. found implicit propagation to be of crucial importance for efficient practical use of SOS when describing a major programming language. However, such reliance on conventions undermines the formality of semantic descriptions. It also means that the unabbreviated rules are still needed (e.g. for use in formal reasoning about the described language). How could we remedy this apparent weakness of SOS?

The main aim of the present paper is to develop a *comprehensive formal treatment of implicit propagation* of auxiliary entities in SOS. This will not only avoid the need for multiple versions of SOS rules for the same construct and maximise reusability, but also allow the formulation of most SOS rules to be significantly simplified: implicitly-propagated auxiliary entities simply do not need to be mentioned. For example, SOS rule (1) above can *formally* be written as follows in our framework for implicit propagation:

$$\frac{c_1 \longrightarrow c_1'}{c_1 \,;\, c_2 \longrightarrow c_1' \,;\, c_2} \tag{5}$$

Assuming that **nil** represents normal termination of commands, the following rule completes our highly reusable rules for command sequencing:

$$\mathbf{nil} \,;\, c_2 \longrightarrow c_2 \tag{6}$$

The above rules are clearly as simple as possible, and can be used without change in the description of any (programming or process) language that includes such a sequencing construct. [5]

We achieve our stated aim by interpreting such SOS rules as MSOS rules, constructing the labels required in MSOS from the explicitly-specified auxiliary entities. The primary advantages, compared to direct use of MSOS, are that familiar SOS notation (such as the '$\vdash$' illustrated earlier) can be used for specifying auxiliary entities when they are not simply propagated in particular rules, and that the somewhat inelegant MSOS notation for labels is not needed at all. Let us refer to our new framework as *Implicitly-Modular SOS (I-MSOS)*.

[5] The concrete terminal symbols conventionally used in abstract syntax may however require replacing; it is preferable to introduce language-independent notation for abstract constructs [19,20].

Implicit propagation of unmentioned variables is a familiar concept from imperative programming languages. It has previously been exploited in various semantic frameworks, including abstract state machines, action semantics, monadic semantics, attribute grammars and, most recently, K [9]. Our contribution here is the formal incorporation of implicit propagation in the popular structural style of operational semantics.

Plan.

Section 2 recalls the foundations of SOS and MSOS, then defines how SOS specifications are interpreted as MSOS specifications, allowing implicit propagation. Section 3 considers familiar examples of SOS specifications from the literature, and shows how much simpler they are when reformulated using implicit propagation. Section 4 concludes with a summary of the contribution of the paper, and indicates plans for future development.

2 Foundations

In this section we first recall how structural operational semantics (SOS) can be understood as a special case of inductive definition of relations, and consider the kinds of signatures required for specifying program syntax, auxiliary entities, and semantic relations. We then explain the main differences between SOS and the modular variant MSOS in terms of these signatures. Finally, we show how to obtain implicit propagation of auxiliary arguments of relations in SOS by translating the specifications to MSOS.

2.1 Inductive Operational Semantics

Transition system specifications of concurrent processes and SOS specifications of programming languages both consist of inductive definitions of semantic relations that involve program syntax, using (axioms and inference) rules. When specifying concurrent processes, the semantic relations involve also labels, whereas when specifying programming languages, they generally involve other auxiliary entities, such as stores and environments, and less often make use of labels.

The so-called big-step style of SOS (also known as 'natural semantics' [10]) specifies transitions that go straight from program syntax to computed values. Compared to Plotkin's original 'small-step' style [24], the big-step style is commonly regarded as more appropriate for the *static* semantics of programming languages (also for the semantics of non-computational specification languages, e.g. CASL [3]), and unsuitable for the dynamic semantics of languages that involve concurrency or interleaved effects. Note however that the two styles can be used together in the semantics of the same language (e.g. small-step for command execution, big-step for expression evaluation).

A major benefit of defining language semantics inductively is that sets of inference rules defining different parts of a language can be specified separately and subsequently combined (just as context-free grammars used to specify the syntax of parts of a language can be combined). Care is however needed to ensure consistency

in connection with the specification of sets of auxiliary entities. For example, an SOS description of an integer variable declaration might require the set of stores to be the set of finite mappings from locations to some abstract sort of storable values, and require the storable values to *include* the integers as a subsort; to require the storable values to consist *only* of the integers could well be inconsistent with requirements in other parts of the language. (The systematic use of abstract sorts to support modularity and reuse in incremental semantic descriptions has been fully developed by Meseguer and Braga [12].) Whether auxiliary entities are specified algebraically, set-theoretically, or by other means does not substantially affect the need for consistent definitions of them.

However, as explained in the Introduction, consistency of notation for semantic relations results in the need for explicit propagation of auxiliary entities in SOS rules, and restricts the possibility of reusing the same descriptions of common constructs (without reformulation) in different language descriptions. Let us proceed to address this crucial problem.

The first-order signatures used in the Common Algebraic Specification Language CASL [2,4,18] provide adequate foundations for the notation used in SOS specifications, including the use of subsorts, overloading, and partial operations on auxiliary entities: [6]

Definition 2.1 (Signature [18]) *A signature* $\Sigma = (S, \leq, TF, PF, P)$ *consists of: a set* S *of sorts with pre-order* $\leq$*; (disjoint) sets* $TF_{w,s}$ *and* $PF_{w,s}$ *of total, resp. partial, function symbols for each sequence of argument sorts* $w \in S^*$ *and result sort* $s \in S$*; and a set* P_w *of predicate symbols for each sequence of argument sorts* $w \in S^*$*.*

Definition 2.2 (Term, formula) *Given a signature* $\Sigma = (S, \leq, TF, PF, P)$ *and a family* X *of (disjoint) sets* X_s *of variables for each* $s \in S$*:*

- *A* term t *is either a variable* x *in* X_s *or a constant* f *in* $TF_{(),s} \cup PF_{(),s}$ *for some* $s \in S$*, or a well-sorted application* $f(t_1, \ldots, t_n)$ *of a function symbol* f *in* $TF \cup PF$ *to argument terms* $t_1, \ldots, t_n$*.*
- *An (atomic)* formula *is a well-sorted equation* $t_1 = t_2$ *or inequation* $t_1 \neq t_2$*, a definedness assertion* $\mathrm{def}(t)$*, or a well-sorted application* $p(t_1, \ldots, t_n)$ *of a predicate symbol* p *in* P *to argument terms* $t_1, \ldots, t_n$*.*

A term or formula is called closed *when it does not contain any variables.*

The value of a term and the holding of a formula for an assignment of values to variables is as usual. Note that predicates never hold on arguments whose value is undefined (due to application of a partial function) and that whether they hold is unaffected by subsort embeddings.

Definition 2.3 (Abstract syntax) *The abstract syntax of (part of) a language* L *is represented by a signature of the form* $\Sigma^L = (S^L, \leq^L, TF^L, \emptyset, \emptyset)$*.*

The sorts in S^L classify the phrases of the language (commands, expressions, etc.), the function symbols in TF^L correspond to phrase constructors, and subsort inclu-

[6] For simplicity, we do not consider signatures with higher-order operations that can be used to represent variable binding constructs.

sions correspond to direct inclusions (e.g. of identifiers in expressions). For example, consider the following constructs:

$$e ::= x \mid \mathbf{let}\, d\, \mathbf{in}\, e$$

The corresponding signature has $S^L = \{e, x, d\}$, the pre-order includes $x \leq^L e$, and $TF^L = \{\mathbf{let}\ \_\ \mathbf{in}\ \_ : d \times e \to e\}$. The abstract syntax of a complete language is given by the initial algebra of the union of the signatures [7] for its parts.

Definition 2.4 (Auxiliary entities) *The notation for auxiliary entities is represented by an extension of Σ^L to a signature $\Sigma^A = (S^A, \leq^A, TF^A, PF^A, P^A)$, together with a family X of (disjoint) sets X_s of variables for each $s \in S^A$.*

The notation for auxiliary entities (such as stores and environments) may involve not only further sorts S^A, subsorts $\leq^A$ and total functions TF^A but also partial functions PF^A and predicates P^A. For example, the notation for environments ρ may include a total function $\_[\_]$ for combining environments, and a partial function $\_(\_)$ such that $\rho(x)$ returns the value of x in ρ when it exists, and is otherwise undefined. An example of a predicate is the ordering $m < n$ on numbers. Whether the intended interpretation of this notation is specified algebraically (e.g. in Casl) or set-theoretically does not affect its use in SOS rules: all we need is its signature, and variables for each sort.

An SOS specification extends the specification of abstract syntax and auxiliary entities by introducing some relation symbols, and defining the relations inductively by rules. In process algebra, transition system specifications usually involve ternary relation symbols (corresponding to labelled transitions between states) and unary predicate symbols (corresponding to subsets of states), and the introduction of the symbols is often left implicit. In descriptions of programming languages, in contrast, the argument sorts of relation symbols tend to be less homogeneous, and the explicit introduction of the relation symbol being defined by a set of rules is common practice (cf. [23]).

For the rest of this section, let the signature Σ^A and sets of variables X be given.

Definition 2.5 (Relation symbols) *The introduction of a relation symbol r with argument sorts $s_1, \ldots, s_n$ is specified by its application to distinct variables $x_i \in X_{s_i}$:*

$$\boxed{r(x_1, \ldots, x_n)}$$

The introduction of a family R of relation symbols corresponds to extending Σ^A to a signature Σ^R.

For example, the transition relation used in (1) in Section 1 is introduced by:

$$\boxed{\rho \vdash \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle}$$

where the variable ρ ranges over a set of environments, c, c' range over the abstract syntax of commands, and σ, σ' range over a set of stores. The usual convention is to reserve the use of primes ($'$) for variables which correspond to components of the *results* of transitions.

[7] Union of Casl signatures is defined in the obvious way.

When a computation for a phrase of sort $s \in S^L$ terminates normally, it computes a value in an associated sort, the elements of which might be either syntactic (e.g. a '**nil**' command can be regarded as the value computed by arbitrary commands) or auxiliary entities (e.g. declarations compute environments). The relations that represent computations may thus have arguments involving the specified abstract syntax, auxiliary entities and computed values.

Definition 2.6 (Rules) *The interpretation of the relation symbols in R is specified by a set of rules $\frac{H}{C}$, where the* premises *H is a set of (atomic) formulae over Σ^R and the* conclusion *C is an application of a relation symbol in R to terms over Σ^A. When H is the empty set, the rule is written simply as C.*

An application of a relation symbol in R is sometimes called a *judgement*; other (atomic) formulae in H are called *side-conditions*.

Definition 2.7 (Relations defined by rules) *Given a model M with signature Σ^A, the interpretation of the relation symbols in R is the least set of relations between elements of the carrier sets of M such that for each rule $\frac{H}{C}$ in R, whenever all the premises in H hold for a particular assignment of elements of M to the variables used in the rule, so does C.*

We consider only *positive* rules in this paper (partly for simplicity, but note also that rules with negated judgements are generally eschewed in SOS descriptions of programming languages). Then, by the classic theory of inductive definitions, the interpretation of relation symbols defined by rules always exists. It corresponds to the semantics of a structured free-extension specification in Casl, where the inference rules are expressed as universally-quantified implications (under the assumption that the sets of premises are finite).

2.2 SOS and MSOS

The small-step style in SOS and MSOS usually involves the gradual replacement of phrases by their computed values. For each phrase sort $s \in S^L$ specified in small-step style, let the sort of computed values for s be included as a subsort of s in the extension of Σ^L to Σ^A (except when computed values are syntactic, and already a subsort in Σ^L); then the initial algebra of the extended signature provides sets of so-called *value-added syntax trees*, which include mixtures of syntax and computed values. In contrast, the big-step style relates phrases directly to their computed values, so value-added syntax trees are not needed there.

The argument sorts of the relations in R depend to some extent on the style of the rules used to specify them:

TSS: In the general format for transition system specifications developed by Mousavi et al. [21], each relation $r(x_1, \ldots, x_n)$ is either a transition relation with three or more arguments, the first and last arguments (x_1, x_n) being of the same syntactic sort $s \in S^L$, and the other arguments $(x_2, \ldots, x_{n-1})$ together forming a label; or a predicate with a single syntactic argument. Computed values are here syntactic (e.g. the nil process) so value-added syntax is not needed.

SOS: The arguments can be divided into three groups. Group 1 includes a syntactic argument of some sort $s \in S^L$, which, together with any remaining arguments

in this group, form the states of a labelled transition system. The arguments in group 2 (if any) form the labels on transitions. If group 3 is empty, the relation corresponds to a predicate on states; otherwise, the relation corresponds to a transition, and group 3 includes an argument of sort s (for the small-step style) or of the sort of values computed by phrases of sort s (for big-step). Any further arguments in group 3 are of the same sorts as corresponding arguments in group 1; any remaining components of states are implicitly left unchanged by transitions. For example, consider:

$$\boxed{\rho \vdash \langle c, \sigma \rangle \longrightarrow \langle c', \sigma' \rangle}$$

Group 1 consists of ρ, c and σ, group 2 is empty, and group 3 consists of c' and σ'; the relation is a transition, and it leaves the environment component ρ unchanged. (This grouping of arguments is determined by the association of variables with components of states.)

MSOS: All semantic relations have either two or three arguments. The first argument is of some syntactic sort $s \in S^L$, forming the states of a generalised transition system, which is a labelled transition system where the labels are the morphisms of a category, and the traces of computations are required to be paths in that category [17]. The second argument is of the distinguished sort *Label*, corresponding to the morphisms of the label category. If there is a third argument, it is of sort s (for the small-step style) or of the sort of values computed by phrases of sort s (for big-step), and the relation is a transition relation; otherwise it is a predicate. For example, a small-step transition relation can be introduced as follows in MSOS:

$$\boxed{c \xrightarrow{X} c'}$$

Auxiliary entities can be specified to be components of labels when needed for use in particular (sets of) rules, using set equations such as:

$$\mathit{Label} = \{\rho : \mathit{Env},\ \sigma, \sigma' : \mathit{Store}, \mathbf{...}\} \tag{7}$$

The above equation can be interpreted as the *inclusion* of *Label* in the set of all records that have *at least* the indicated components. Labels are essentially *record values*, as found in some programming languages, and can be regarded mathematically as morphisms of an indexed product category.

Components of labels can be matched and determined in rules using notation such as '$\{\rho = \rho_0[x \mapsto l], X\}$', where X varies over arbitrary records (X is usually written '**...**' when its only use is propagation), and '$\{\sigma = \sigma_0, \sigma' = \sigma_0[l \mapsto v], U\}$', where U varies over records that form *unobservable* labels, i.e. identity morphisms of label categories (we write '—' instead of U when its only use is propagation). Note that when '**...**' is used instead of X, different occurrences of it in the same rule stand for the same components; similarly for '—'.

Whether the indices of label components are written with or without primes ($'$) affects the composability and (un)observability of labels [17]:

- When a component index occurs only without a prime (e.g. ρ) that label component corresponds to *read-only* information, which has to be identical in labels on adjacent transitions (giving the effect of a *relative* labelled transition system).

- When the same index occurs both with and without a prime (e.g. σ, σ') the two label components correspond to current and subsequent *updatable* information, which has to be composable in labels on adjacent transitions (giving the effect of single-threading).
- When a component index occurs only with a prime, the component corresponds to *emitted* information, which is unconstrained in labels on adjacent transitions. The corresponding set of auxiliary entities has to be a *monoid*.

A label is unobservable when its updatable information remains unchanged, and its emitted information is the unit of the corresponding monoid.

An algebraic specification constructing records from multisets of pairs is given in [12]. Our MSOS notation $\{i{=}t, l\}$ corresponds to '$\{i{=}t\}{+}l$', where $l_1{+}l_2$ is the usual partial union operation on records. Algebraically, the unobservable labels form a subsort of *Label*, and the sort *Label* is equipped with a partial composition operation '$l_1; l_2$'. Note that notation for the empty record is not used at all in MSOS specifications.

I-MSOS: Each relation symbol is introduced as in SOS, but with highlighting [8] of all its explicit auxiliary arguments, e.g.:

$$\rho \vdash \langle\, c\,, \sigma\rangle \longrightarrow \langle\, c'\,, \sigma'\rangle$$

In I-MSOS rules, a relation symbol can be applied to all the explicit auxiliary arguments that were specified when it was introduced, just as in ordinary SOS. However, any auxiliary argument can also be omitted (along with the accompanying separators). All missing auxiliary arguments are implicitly propagated between the premises and conclusions of rules.

To avoid mentioning particular auxiliary arguments at all when specifying constructs whose rules do not need to refer to them, relation symbols that leave those arguments implicit can be introduced, e.g.:

$$\rho \vdash c \longrightarrow c'$$

$$\langle\, c\,, \sigma\rangle \longrightarrow \langle\, c'\,, \sigma'\rangle$$

$$c \longrightarrow c'$$

I-MSOS relation symbols that differ only regarding their auxiliary arguments can be used interchangeably in rules, and regarded as equivalent, when those arguments are simply propagated. For example, any of the relation symbols introduced above could be used to specify the rules for command sequencing.

When introducing relations, I-MSOS requires the systematic use of primed variables to distinguish between current and resulting entities.

The full interpretation of I-MSOS specifications is given by a reasonably straightforward translation to MSOS, which is explained and defined below.

[8] When adding new constructs to a language described in conventional SOS, Pierce [23] highlights the changes that needed to be made to existing relation symbols and rules, for pedagogical reasons.

2.3 Implicit Propagation

I-MSOS allows auxiliary arguments to be omitted when introducing and using relations: they are then implicitly propagated between the premises and conclusions of rules, and by unconditional rules. This is achieved by translating each I-MSOS relation to an MSOS relation with the same syntactic arguments, but whose only auxiliary argument is its label; the auxiliary arguments become required components of labels. Thus equivalent I-MSOS relations, which differ only regarding their auxiliary arguments, are translated to the same MSOS relation. The conventional use of primes (to indicate arguments that result from making transitions) is preserved in the indices of the label components. [9]

The translation of each I-MSOS rule to MSOS assumes that the use of explicit auxiliary arguments is consistent throughout the rule: if one relation used in the rule has a particular auxiliary argument, any other relations used in it must have that argument too. This assumption is quite natural in practice (at least in the small-step style, where rules usually have at most one premise) and ensures that the same auxiliary arguments are implicitly propagated by all the relations in the same rule.

Recall that I-MSOS and MSOS specifications both involve:

- a signature Σ^L for abstract syntax;
- its extension to a signature Σ^A with notation for auxiliary entities, and a family X of (disjoint) sets X_s of variables;
- introduction of relation symbols r, specified by their applications $r(x_1, \ldots, x_n)$ to distinct variables $x_i \in X_{s_i}$, determining a family R of relation symbols and the corresponding extension of Σ^A to Σ^R; and
- a set of inference rules $\frac{H}{C}$, where H is a set of atomic formulae over Σ^R and C is an application of a relation symbol in R to terms over Σ^A.

A semantic relation symbol can be introduced more than once in an I-MSOS specification. We assume that each time the same relation symbol is introduced, the same variables are used to indicate its argument sorts; this ensures that the sorts of the argument terms in an application $r(t_1, \ldots, t_n)$ uniquely determine the variables $x_1, \ldots, x_n$ used to introduce r.

Consider an introduction of an I-MSOS semantic relation r, which has the following abstract structure:

$$\boxed{r(u, v' \; \boxed{, w_1, \ldots, w_n} \;)} \tag{8}$$

This determines an MSOS relation symbol $\bar{r}$, where the mapping from r to $\bar{r}$ identifies symbols iff their non-highlighted parts are identical. The syntactic arguments are indicated by an unprimed variable u of some sort s and a primed variable v' of some sort s', and the auxiliary arguments are indicated by the (highlighted, distinct and possibly primed) variables $w_1, \ldots, w_n$ of sorts $s_1, \ldots, s_n$ (the sorts need not be distinct).

[9] Action arguments labelling transitions should therefore always be primed, and come from a monoid.

The translation of (8) consists of the introduction of the corresponding MSOS relation $\bar{r}$ with syntactic argument sorts s and s' (and auxiliary argument X of sort *Label*), together with a specification of the set of labels:

$$\boxed{\bar{r}(u, X, v')} \qquad \mathit{Label} = \{w_1 : s_1, \cdots, w_n : s_n, \mathbf{...}\} \ . \tag{9}$$

The translation of a rule $\frac{H}{C_0}$ from I-MSOS to MSOS affects only the applications of relation symbols r in R, and depends on the number m of such applications that H contains. Let $H = \{C_i \mid 1 \leq i \leq m\} \cup \bar{H}$, where $C_i = r_i(t_i, t_i', t_{i1}, \ldots, t_{in})$ for $0 \leq i \leq m$ (n is independent of i by a previous assumption) and $\bar{H}$ is the set of side-conditions in H. The translation of the I-MSOS rule $\frac{H}{C_0}$ is the MSOS rule $\frac{H'}{C_0'}$ where:

- $H' = \{C_i' \mid 1 \leq i \leq m\} \cup \bar{H} \cup H_m$, and
- $C_i' = \bar{r}_i(t_i, \{w_{i1}{=}t_{i1}, \ldots, w_{in}{=}t_{in}, z_{im}\}, t_i')$ for $0 \leq i \leq m$.

The choice of variables z_{im} and the set of extra side-conditions H_m depend on m:

$m = 0$**:** z_{00} is '—', and $H_m = \emptyset$;

$m = 1$**:** z_{01}, z_{11} are both '**...**', and $H_m = \emptyset$;

$m \geq 2$**:** There are three possibilities:

(i) z_{im} is X_i for $0 \leq i \leq m$, and $H_m = \{X_0 = X_1; \cdots; X_m\}$;

(ii) z_{im} is X_i for $0 \leq i \leq m$, and $H_m = \{\mathit{shuffles}(X_0, X_1, \cdots, X_m)\}$, where the side-condition holds iff X_0 is a composition of $X_1, \ldots, X_m$ in any order; or

(iii) z_{im} is '—' for $0 \leq i \leq m$, and $H_m = \emptyset$.

Translation (i) corresponds directly to the state convention used in *The Definition of Standard ML* [14], whereas (ii) corresponds to a convention that might be appropriate if Standard ML were to allow arbitrary order of evaluation of subexpressions as well as the current sequential evaluation. Translation (iii) is actually a special case of both the other translations, since the unobservability of the implicitly propagated components ensures that they are identity for label composition, and that all their shuffles are the same.

Our inclination is towards translation (iii). This is because it does not depend on the order in which premises are given, and it avoids introducing a nondeterministic choice of order. Moreover, the difference between (iii) and the alternatives is generally significant only when giving a big-step SOS for a language where computations have effects, [10] and there are technical reasons for preferring the use of the small-step style in such cases: the modular semantics for throwing and catching exceptions, as illustrated for the simple case of error propagation in [17], appears not to be possible in big-step SOS. Putting it dogmatically: it seems best to reserve big-step SOS for expressing the semantics of 'mathematical' fragments of programming languages (e.g. static semantics, evaluation of numerals to numbers, pattern-matching), and to use small-step for all constructs whose execution might involve significant computational effects. In such cases, translation (iii) appears to be completely adequate.

[10] It is significant also in small-step rules for synchronisation between processes.

For greater generality, we could allow extra notation in rules, to override the default translation given by (iii). For example, translation (i) could be specified by separating the premises of rules by semicolons instead of commas:

$$\frac{e_1 \Rightarrow v_1; \quad \cdots ; \quad e_n \Rightarrow v_n}{e \Rightarrow v} \tag{10}$$

Such explicit specification of the order of evaluation of the subexpressions would formalise the state-convention used in *The Definition of Standard ML* [14], at the same time allowing implicit propagation of environments (but not of exceptions).

Big-step rules for imperative constructs (such as those given in *The Definition of Standard ML*) involve a particular form of *look-ahead* [1]: variables from the target of one premise are used in the source of another premise. Translation (i) does not itself support less regular forms of look-ahead. However, in I-MSOS it is still possible to specify such propagation: by making explicit those arguments involved. In fact *any* conventional SOS can directly be regarded as an I-MSOS, not exploiting implicit propagation of auxiliary arguments at all; translation (iii) yields an MSOS whose computations correspond to those of the original SOS.

3 Examples

In this section we recall some familiar examples of SOS descriptions from the literature, and illustrates the improvements that can be obtained by exploiting implicit propagation of auxiliary arguments in I-MSOS. We also illustrate the result of translating the I-MSOS descriptions to MSOS.

The programming language constructs described in the examples have the following (abstract) syntax:

$$e ::= \mathit{con} \mid x \mid \mathbf{let}\, d\, \mathbf{in}\, e \mid \ldots$$
$$c ::= x := e \mid \mathbf{nil} \mid c\,;c \mid \ldots$$

Their SOS was described by Plotkin [24] using a (small-step) transition relation. Expressions e compute constants con, and could have side-effects; declarations d compute environments ρ; and commands compute **nil**. The transition relation has environments (mapping identifiers x to constants or locations) and stores σ (mapping locations l to constants) as auxiliary arguments.

The leftmost columns of Tables 1 and 2 show the relevant parts of the SOS of a language including all the above constructs, corresponding closely to examples given by Plotkin [24]. The rightmost columns show *independent* descriptions of the individual constructs I-MSOS. The middle columns show the translation of the I-MSOS descriptions into MSOS. In contrast to the SOS descriptions, the MSOS and I-MSOS descriptions are highly reusable, and do not need reformulating when the described constructs are combined with other constructs – not even when expressions are extended to include abrupt termination or concurrent processes.

The first example in Table 1 describes the evaluation of an identifier that is bound to a constant in the current environment ρ. The side-condition $\rho(x) = \mathit{con}$ holds only when $\rho(x)$ is defined. In the MSOS description, the use of '—' in the label specifies that the transition has no side-effects; this is implicit in the corresponding

SOS:	MSOS:	I-MSOS:
$\boxed{\rho \vdash \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}$	$\boxed{e \xrightarrow{X} e'}$ $\quad Label{=}\{\rho{:}Env, \ldots\}$	$\boxed{\rho \vdash e \longrightarrow e'}$
$\dfrac{\rho(x) = con}{\rho \vdash \langle x, \sigma\rangle \longrightarrow \langle con, \sigma\rangle}$	$\dfrac{\rho(x) = con}{x \xrightarrow{\{\rho,\ldots\}} con}$	$\dfrac{\rho(x) = con}{\rho \vdash x \longrightarrow con}$
$\boxed{\rho \vdash \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}$	$\boxed{e \xrightarrow{X} e'}$ $\quad Label{=}\{\rho{:}Env, \ldots\}$	$\boxed{\rho \vdash e \longrightarrow e'}$
$\dfrac{\rho \vdash \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle}{\rho \vdash \langle \mathbf{let}\, d\, \mathbf{in}\, e, \sigma\rangle \longrightarrow \langle \mathbf{let}\, d'\, \mathbf{in}\, e, \sigma'\rangle}$	$\dfrac{d \xrightarrow{\{\ldots\}} d'}{\mathbf{let}\, d\, \mathbf{in}\, e \xrightarrow{\{\ldots\}} \mathbf{let}\, d'\, \mathbf{in}\, e}$	$\dfrac{d \longrightarrow d'}{\mathbf{let}\, d\, \mathbf{in}\, e \longrightarrow \mathbf{let}\, d'\, \mathbf{in}\, e}$
$\dfrac{\rho[\rho_0] \vdash \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\rho \vdash \langle \mathbf{let}\, \rho_0\, \mathbf{in}\, e, \sigma\rangle \longrightarrow \langle \mathbf{let}\, \rho_0\, \mathbf{in}\, e', \sigma'\rangle}$	$\dfrac{e \xrightarrow{\{\rho=\rho[\rho_0],\ldots\}} e'}{\mathbf{let}\, \rho_0\, \mathbf{in}\, e \xrightarrow{\{\rho,\ldots\}} \mathbf{let}\, \rho_0\, \mathbf{in}\, e'}$	$\dfrac{\rho[\rho_0] \vdash e \longrightarrow e'}{\rho \vdash \mathbf{let}\, \rho_0\, \mathbf{in}\, e \longrightarrow \mathbf{let}\, \rho_0\, \mathbf{in}\, e'}$
$\rho \vdash \langle \mathbf{let}\, \rho_0\, \mathbf{in}\, con, \sigma\rangle \longrightarrow \langle con, \sigma\rangle$	$\mathbf{let}\, \rho_0\, \mathbf{in}\, con \xrightarrow{\{\ldots\}} con$	$\mathbf{let}\, \rho_0\, \mathbf{in}\, con \longrightarrow con$
$\boxed{\rho \vdash \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle}$	$\boxed{c \xrightarrow{X} c'}$ $\quad Label{=}\{\rho{:}Env, \sigma, \sigma'{:}Store, \ldots\}$	$\boxed{\rho \vdash \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle}$
$\dfrac{\rho \vdash \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\rho \vdash \langle x := e, \sigma\rangle \longrightarrow \langle x := e', \sigma'\rangle}$	$\dfrac{e \xrightarrow{\{\ldots\}} e'}{x := e \xrightarrow{\{\ldots\}} x := e'}$	$\dfrac{e \longrightarrow e'}{x := e \longrightarrow x := e'}$
$\dfrac{\rho(x) = l}{\rho \vdash \langle x := con, \sigma\rangle \longrightarrow \langle \mathbf{nil}, \sigma[l \mapsto con]\rangle}$	$\dfrac{\rho(x) = l}{x := con \xrightarrow{\{\rho,\sigma,\sigma'=\sigma[l\mapsto con],\ldots\}} \mathbf{nil}}$	$\dfrac{\rho(x) = l}{\rho \vdash \langle x := con, \sigma\rangle \longrightarrow \langle \mathbf{nil}, \sigma[l \mapsto con]\rangle}$

Table 1
Constants, blocks, and assignment commands

I-MSOS axiom. Notice that stores are not mentioned at all in the MSOS and I-MSOS descriptions, so they could be reused directly for purely functional languages.

The second example describes evaluation of the expression **let** d **in** e, where the scope of the declaration d is local to e. The combination $\rho[\rho_0]$ of the current environment and the environment computed by d is used for the evaluation of e.

The third example describes the execution of the command $x := e$. Notice that the second I-MSOS rule is identical to the corresponding SOS rule, whereas the auxiliary arguments are implicitly propagated in the first I-MSOS rule.

In Table 2 the contrast between the SOS and I-MSOS rules in the description of sequential commands $c_1 ; c_2$ is particularly marked, and the notational advantages of I-MSOS over MSOS are also apparent.

Our final example illustrating the relationship between SOS, I-MSOS and MSOS concerns process algebra, and describes the following constructs from CCS [13]:

$$p ::= \mu.p \mid p{+}p \mid p|p \mid \ldots$$

The variable a ranges over the set Act of actions (which is assumed to be closed under the complementation operation $\bar{a}$), and μ ranges over $Act \cup \{\tau\}$. In the MSOS and I-MSOS descriptions, α' ranges over the free monoid Act^* where τ denotes the unit element. The SOS transition relation has only a single auxiliary argument, but I-MSOS still provides notational simplification of some of the rules; the benefits would be more pronounced if we were to add further auxiliary arguments, such as localities [5].

The last rule has multiple transitions, and illustrates variant (iii) of the translation from I-MSOS to MSOS. It requires that synchronisation between p_1 and p_2 should not be combined with observable effects. This seems reasonable in connection with pure process calculi such as CCS, but a less restrictive translation may be needed for describing synchronisation between processes involving shared variables.

4 Conclusion

The main contribution of this paper is the development of the I-MSOS framework, which provides a novel formal treatment of implicit propagation of auxiliary arguments in SOS. I-MSOS has significant pragmatic advantages over both SOS and MSOS: like MSOS, it allows the operational semantics of individual constructs to be described independently, and combined without reformulation; but I-MSOS also has the advantage over MSOS of allowing the use of the familiar notation of conventional SOS, and of avoiding the need for new notation regarding labels.

This paper is a report on work in progress, and a considerable amount of further work is required to establish I-MSOS more firmly as a sound and useful approach to modular formal semantics. One obvious aspect that needs further investigation is whether I-MSOS is (at least) as convenient as SOS as a basis for proving bisimulation equivalences; initial experiments with bisimulation for MSOS were encouraging, [11] but need repeating for I-MSOS, and extending. Some tools have been developed for animating MSOS descriptions, but they will need to be adapted to support I-MSOS. A format ensuring operational conservative extension needs to be defined. Etc.

[11] Weak bisimulation is naturally defined in terms of unobservable labels in MSOS [17].

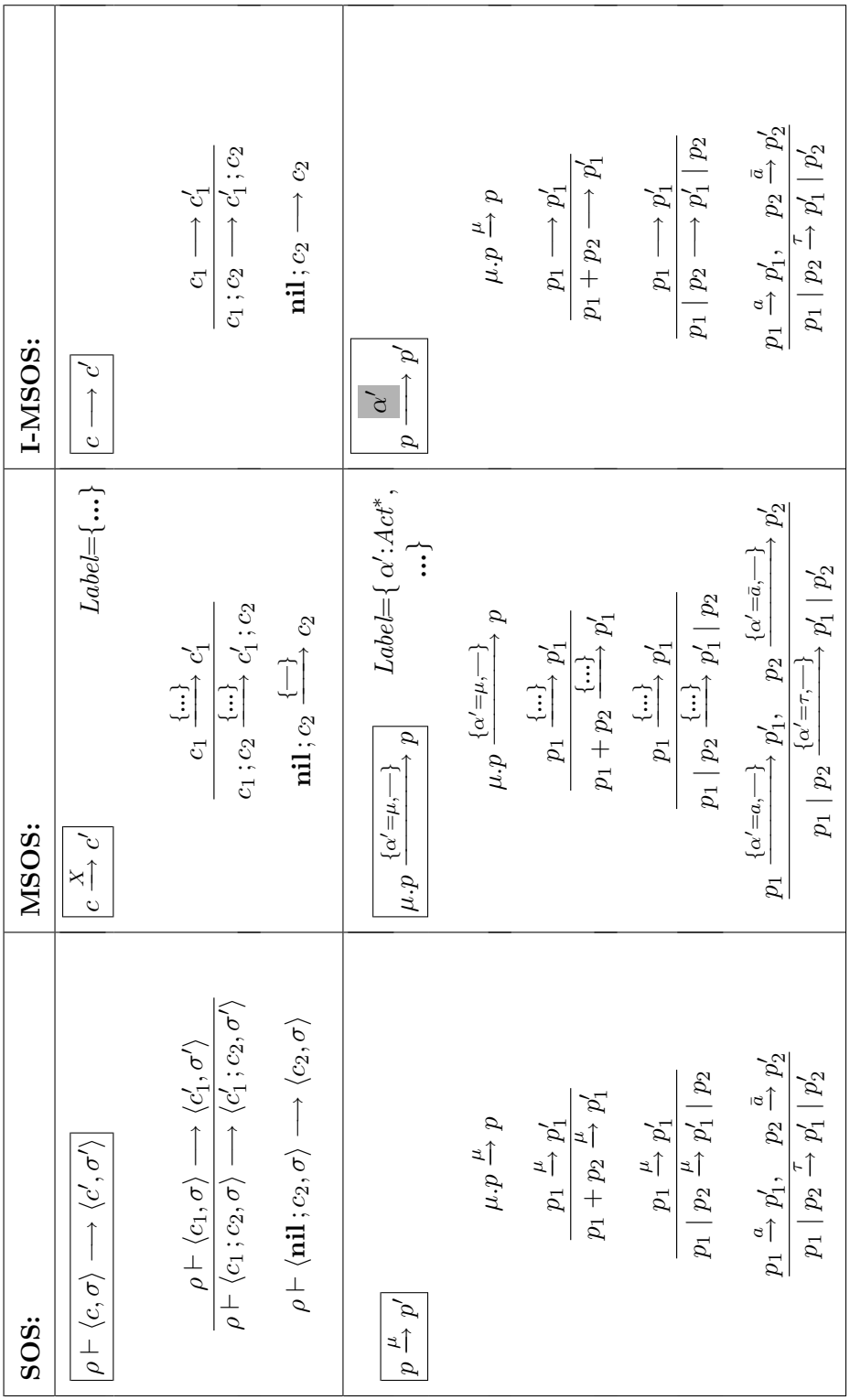

Table 2
Command sequences and concurrent processes

Acknowledgement

The idea of letting labels remain implicit in MSOS was suggested by members of the FSL group at UIUC. The authors are grateful to the anonymous referees and to the workshop participants for perceptive and helpful comments.

References

[1] Aceto, L., W. Fokkink and C. Verhoef, *Structural operational semantics*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra*, Elsevier Science, 2001 pp. 197–292.

[2] Astesiano, E., M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella and A. Tarlecki, Casl*: The Common Algebraic Specification Language*, Theor. Comput. Sci. **286** (2002), pp. 153–196.

[3] Baumeister, H., M. Cerioli, A. Haxthausen, T. Mossakowski, P. D. Mosses, D. Sannella and A. Tarlecki, Casl *semantics*, in: Casl *Reference Manual*, LNCS **2960**, Springer, 2004 pp. 115–271.

[4] Bidoit, M. and P. D. Mosses, "Casl User Manual – Introduction to Using the Common Algebraic Specification Language," LNCS **2900**, Springer, 2004.

[5] Boudol, G., I. Castellani, M. Hennessy and A. Kiehn, *Observing localities*, Theor. Comput. Sci. **114** (1993), pp. 31–61.

[6] Cenciarelli, P., A. Knapp, B. Reus and M. Wirsing, *An event-based structural operational semantics of multi-threaded Java*, in: *Formal Syntax and Semantics of Java*, LNCS **1523** (1999), pp. 157–200.

[7] Cenciarelli, P., A. Knapp and E. Sibilio, *The Java memory model: Operationally, denotationally, axiomatically*, in: *ESOP 2007*, LNCS **4421** (2007), pp. 331–346.

[8] Hennessy, M., "The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics," Wiley, New York, 1990.

[9] Hills, M., T. Serbănuţă and G. Roşu, *A rewrite framework for language definitions and for generation of efficient interpreters*, Electr. Notes Theor. Comput. Sci. **176** (2007), pp. 215–231.

[10] Kahn, G., *Natural semantics*, in: *STACS'87*, LNCS **247** (1987), pp. 22–39.

[11] Levin, M. Y. and B. C. Pierce, *TinkerType: a language for playing with formal systems*, J. Funct. Program. **13** (2003), pp. 295–316.

[12] Meseguer, J. and C. Braga, *Modular rewriting semantics of programming languages*, in: *AMAST 2004*, LNCS **3116** (2004), pp. 364–378.

[13] Milner, R., "Communication and Concurrency," Prentice-Hall, 1989.

[14] Milner, R., M. Tofte, R. Harper and D. MacQueen, "The Definition of Standard ML (Revised)," MIT Press, 1997.

[15] Mosses, P. D., *Foundations of Modular SOS*, in: *MFCS'99*, LNCS **1672** (1999), pp. 70–80.

[16] Mosses, P. D., *Pragmatics of Modular SOS*, in: *AMAST'02*, LNCS **2422** (2002), pp. 21–40.

[17] Mosses, P. D., *Modular structural operational semantics*, J. Log. Algebr. Program. **60-61** (2004), pp. 195–228.

[18] Mosses, P. D., editor, "Casl Reference Manual, The Complete Documentation of the Common Algebraic Specification Language," LNCS **2960**, Springer, 2004.

[19] Mosses, P. D., *A constructive approach to language definition*, J. UCS **11** (2005), pp. 1117–1134.

[20] Mosses, P. D., *Component-based description of programming languages*, in: *Visions of Computer Science*, BCS, 2008, to appear.

[21] Mousavi, M. R., M. A. Reniers and J. F. Groote, *SOS formats and meta-theory: 20 years after*, Theor. Comput. Sci. **373** (2007), pp. 238–272.

[22] Nielson, H. R. and F. Nielson, "Semantics with Applications: A Formal Introduction," Wiley, Chichester, UK, 1992.

[23] Pierce, B. C., "Types and Programming Languages," MIT Press, 2002.

[24] Plotkin, G. D., *A structural approach to operational semantics*, J. Log. Algebr. Program. **60-61** (2004), pp. 17–139.