SOS Rule Formats for Parameterized and State-Bearing Processes

DRAFT

Bard Bloom *
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
bard@cs.cornell.edu

Frits Vaandrager †
CWI
P.O. Box 4079, NL-1009 AB Amsterdam
fritsv@cwi.nl
Programming Research Group
University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam

July 1994

Abstract

We propose a general format of Structural Operational Semantics rules that can deal with parameterized and state-bearing processes. Plotkin's original SOS definitions of imperative programs and of CSP, as well as the operational semantics of value passing process calculi like LOTOS, μCRL and real-time ACP are covered by this format. We prove that the format provides a well-defined operational semantics with respect to an appropriate notion of bisimulation, and identify sub-formats with desirable finiteness and decidability properties.

1 Introduction

Plotkin's [19, 20] method of *Structural Operational Semantics (SOS)* is widely used to define the operational meaning of programming languages and process calculi. In recent years it has become more and more clear that many of the key theorems in operational semantics are to a large extent language independent: many important properties of SOS style definitions follow from the syntactic form of the inference rules, and can be established for whole classes of languages ("formats") at the same time. Examples of contributions along these lines are [1, 4, 7, 10, 14, 16, 21, 24]. The resulting theory allows one to develop a simple process algebra for concurrency with substantial mathematical support. For example, one may build a process algebra in the general style of CCS, giving operations their behavior by rules in a standard format. The SOS theory will allow one to automatically conclude that bisimulation is a congruence for the language, and will provide a complete equational axiom system (with one induction principle) for proving equality of processes. Baeten and Verhoef's [5] recent Chapter on Concrete Process Algebra in the Handbook of Logic in Computer Science is an excellent illustration of the importance of SOS theory in the area of process algebra.

*Supported by NSF grants CCR-9003441 and CCR-9223183. The work on this paper was initiated during a visit of the author to Amsterdam paid by the HCM network EXPRESS.

†Supported by ESPRIT BRA 7166 Concur2.

However, thus far SOS theory only applies to what De Bakker et.al. [6] call *uniform* languages: the atomic actions of the language remain uninterpreted and there is no specification for these actions in terms of state transforming functions. This means in particular that there is no general SOS format that covers Plotkin's original semantics of imperative programs [19] and of CSP [20]. Also binding mechanisms, which play an important role in more advanced process calculi like LOTOS [18], μCRL [15], real-time ACP [2] are not covered by existing theory. The difficulty is that the theoretical formats have been designed to describe process interactions, but not data; the rule format has no way of calculating with any data other than processes. We think that there is a need for an extension of SOS theory that deals with parameterized and state-bearing processes. By introducing a general format, it becomes possible to present the operational semantics of widely different languages in a uniform framework, thus preparing the ground for a better understanding of the relationships between these languages. Proving results for SOS formats rather than for individual languages is not only an efficient way of developing theory, but also provides a deeper insight why an SOS semantics for a specific language has certain properties.

In this paper, we extend the theory of SOS to work with complex data. The results are quite general. Our formats cover all the languages we have checked, including all the ones mentioned above, with little more than changes of notation. We develop the basic metatheory of these formats, investigating questions of computability, finiteness, bisimulation as an equivalence, and connection with other semantics and formats.

Briefly, we introduce three formats: Super-SOS the most general, Data-SOS, and Special-SOS the most specific. All three provide well-defined operational semantics which respect appropriate notions of bisimulation. Super-SOS and Data-SOS can define the same operations; Super-SOS descriptions are more compact. Special-SOS is strictly less expressive than the other two; but that restriction in expressiveness gains desirable finiteness and computability properties that the other two lack, and is still general enough for most purposes.

2 Examples of SOS

We present fragments of several standard calculi defined by SOS rules. We try to exemplify problems rather than be complete. Experience has shown that SOS rules are a simple and effective specification technique. They require no deep mathematical knowledge to understand, and resemble programming enough for programmers to feel comfortable. Nonetheless, SOS systems admit powerful methods of reasoning.

2.1 Plotkin-style Semantics of Imperative Programs

The standard Plotkin-style SOS specification of a language generally has the following form. There is a grammar of programs, *control states*, for example:

$$p ::= \mathtt{x} := e \,|\, p \,;\, p \,|\, \textbf{while}\ e\ \textbf{do}\ p \,|\ \cdots \tag{1}$$

where e ranges over a class of expressions and $\mathtt{x}$ over program variables. A state (or, to avoid confusion, *data state*) for such a language is a function σ from variable names to values. Expression e in state σ has the value $[\![\sigma(e)]\!]$. A snapshot of computation is specified by a pair $\langle p, \sigma\rangle$, giving the state of the control and of the data.

The operational semantics of the language has two judgements. $\langle p,\sigma\rangle \to \langle p', \sigma'\rangle$ indicates that a program with snapshot $\langle p,\sigma\rangle$ can compute for one step, and thereafter behave like the program p' in state σ'. The judgement $\langle p,\sigma\rangle \downarrow \sigma'$ means that a program with snapshot $\langle p,\sigma\rangle$ can perform one step of computation and terminate with final state σ'. These judgements are defined by mutual recursion on the structure of p.

We present some typical rules for this language. The first rule explains that assignments terminate in a modified state. The second rule lets the first subprogram in a sequential composition run; the third rule lets the second subprogram take over when the first is done.

$$\frac{}{\langle \mathtt{x} := e, \sigma\rangle \downarrow \sigma[\mathtt{x} \mapsto [\![\sigma(e)]\!]]} \qquad \frac{\langle p_1, \sigma\rangle \to \langle p_1', \sigma'\rangle}{\langle (p_1 \,;\, p_2), \sigma\rangle \to \langle (p_1' \,;\, p_2), \sigma'\rangle} \qquad \frac{\langle p_1, \sigma\rangle \downarrow \sigma'}{\langle (p_1 \,;\, p_2), \sigma\rangle \to \langle p_2, \sigma'\rangle} \tag{2}$$

When one calculates the behavior of p in state σ from such rules, one first looks at all rules which match the structure of p. Each antecedent is matched in all possible ways, and the rule applies in each of those ways. For example, the assignment rule can apply in only one way. The second sequencing rule can apply in as many ways as p_1 can terminate in state σ.

Note the information flow in this computation. One takes the source — the item on the left of the arrow in the conclusion, $\langle (p_1 \,;\, p_2), \sigma\rangle$ in the sequencing rules — as inputs to the process. Each antecedent uses the the source variables as input, and produces some other information — p_1' and σ', above — as output. The target of the rule — the item on the right-hand side of the arrow in the conclusion, $\langle (p_1' \,;\, p_2), \sigma'\rangle$ — may use information from the source, and information produced by the target. This pattern of information flow appears in most SOS rules, and will enter our formal theory later on.

The remaining operations can be given similar specifications. For example, consider the standard **while** loop: first see if the test is true, and if it is, perform the body of the loop and then re-try the loop. Note that the rule makes a copy of the body: there is one p in the source and two p's in the target[1], and the target is a larger term than the source. This hints at the complexities that a general metatheory must deal with.

$$\frac{\sigma[\![e]\!] = \mathsf{true}}{\langle \textbf{while}\ e\ \textbf{do}\ p, \sigma\rangle \to \langle (p \,;\, \textbf{while}\ e\ \textbf{do}\ p), \sigma\rangle} \qquad \frac{\sigma[\![e]\!] = \mathsf{false}}{\langle \textbf{while}\ e\ \textbf{do}\ p, \sigma\rangle \downarrow \sigma} \tag{3}$$

2.2 SOSses for Process Algebras

Process calculi are generally given operational semantics in a similar but distinct style. Again, we have a grammar of operations, such as

$$p ::= a!v.p \mid a?x.p \mid p + p \mid \mathbf{0} \mid p \| p \mid \cdots \tag{4}$$

The intended meaning of these constructs is: $a!v.p$ sends value v on channel a and thereafter behaves like p; $a?x.p$ reads a value on a, calls it x, and thereafter behaves like p. $p + p'$ can behave like either p or p'; $\mathbf{0}$ is a stopped process; $p\|p'$ is a parallel composition. There are several ways to phrase the semantics, which may be divided into *early* and *late* semantics. The early semantics will be seen to be typical uses of our Data-SOS format; the late, typical uses of our Special-SOS format.

2.2.1 Early Semantics

We have three judgements in this calculus: $p \xrightarrow{a!v} p'$ and $p \xrightarrow{a?v} p'$, meaning that p can send or receive v on channel a and thereafter behave like p'; and $p \xrightarrow{\tau} p'$, indicating that p can perform some internal

[1]As usual in specifications, this does not imply that p must be copied in an implementation.

activity and thereafter behave like p'. The state is not an explicit object σ; as in the λ-calculus, values are substituted for variables when they are available.

Again, these judgements are defined by structural induction on p. The first rule is clear enough. The rule for $a?x.p$ requires the process to nondeterministically choose which value v is to be received, and substitute that value for x in p. The third rule requires the receiver's guess to match the sender's transmission, and the fourth rule allows asynchronous computation.[2]

$$\frac{}{a!v.p \xrightarrow{a!v} p} \qquad \frac{}{a?x.p \xrightarrow{a?v} p[x := v]} \qquad \frac{p \xrightarrow{a!v} p', \quad q \xrightarrow{a?v} q'}{p\|q \xrightarrow{\tau} p'\|q'} \qquad \frac{p \xrightarrow{\alpha} p'}{p\|q \xrightarrow{\alpha} p'\|q} \tag{5}$$

For example, $a!3.\mathbf{0}\|a?x.b!(x+1).\mathbf{0} \xrightarrow{\tau} \mathbf{0}\|b!(3+1).\mathbf{0}$.

2.2.2 Late Semantics

The infinitary nondeterministic guessing required in the $a?x.p$ rule is aesthetically displeasing to some researchers, and indeed causes mathematical problems in some aspects of the theory [17, 12]. Thus, *late* semantics were invented. In these semantics, $a?x.p$ simply indicates that it is willing to read a value on a. The early judgement $p \xrightarrow{a?v} p'$ is replaced by the judgement $p \xrightarrow{a?} q$, where q is an entity which can accept a value x and thereafter be a process. We may use familiar mathematics, representing q as the abstraction $\lambda x.r$, a function from values to processes. The parallel communication rule is modified to take account of this change as well:

$$\frac{}{a!v.p \xrightarrow{a!v} p} \qquad \frac{}{a?x.p \xrightarrow{a?} \lambda x.p} \qquad \frac{p \xrightarrow{a!v} p', \quad q \xrightarrow{a?} q'}{p\|q \xrightarrow{\tau} p'\|(q'v)} \qquad \frac{q \xrightarrow{a?} q'}{p\|q \xrightarrow{a?} \lambda x.(p\|(q'x))} \tag{6}$$

The last rule allows one of the processes in a parallel composition to accept input from outside; input to the composition is routed to the process that can accept it. [12] presents a more abstract phrasing of this, involving an additional sort of *input-enabled processes*, and operations which handled routing data.

Now, we have

$$a!3.\mathbf{0}\|a?x.b!(x+1).\mathbf{0} \xrightarrow{\tau} \mathbf{0}\|((\lambda x.b!(x+1).\mathbf{0})3) \Rightarrow \mathbf{0}\|b!(3+1).\mathbf{0}$$

where $\Rightarrow$ is β-reduction. This is essentially the same computation as in the early case, though the $\Rightarrow$ step makes the act of transmitting values to their receivers a bit more explicit. Late semantics have slight technical advantages over early semantics (in particular, the transition relation is finitely branching) at the cost of some added complexity (the addition of basic λ-calculus). Our rule formats incorporate both concepts uniformly.

3 Elements of SOS Specifications

In this section we sketch the rule formats for parameterized processes. A table of conventions and notation appears in Figure 1 on 15. We design our system to handle all of the preceding styles of definition, with (ideally) little more than changes of notation.

[2]Here and elsewhere we present only exemplary rules. In particular, symmetric rules for commutative operators are omitted. Also, we casually use α to range over appropriate actions.

3.1 Signature, Type System and Terms

A *Data-SOS signature* consists of the following information:

- A collection of *data types*, ranged over by $\mathcal{V}, \ldots$. We require a unit type $\mathbf{U}$, a Boolean type $\mathbf{B}$, and assume that the set of data types is closed under products.

  The data types include everything the *semantics* computes with, which may be more than what the language itself computes with: *e.g.*, the states σ in Section 2.1 are values in the semantics but not in the programming language.

- For each data type, a collection of *data expressions*, ranged over by $e, \ldots$. We make minimal demands on the data types: we assume that for each type the set of data expressions contains an infinite set of *data variables*, ranged over by $v, \ldots$. Further, we assume a function $\mathsf{vars}(e)$ that assigns to each term a finite set of *free variables* appearing in e, and a function $\mathbf{eval}(e)$ that assigns to each closed term e a closed term that represents the fully-evaluated form of e, its semantics. We require $\mathbf{eval}(\mathbf{eval}(e)) = \mathbf{eval}(e)$. We assume a substitution operation $e'[x := e]$ that is well-behaved for closed e (so, for instance $\mathsf{vars}(e'[x := e]) = \mathsf{vars}(e') - \{x\}$ and $\mathbf{eval}(e'[x := e]) = \mathbf{eval}(e'[x := \mathbf{eval}(e)])$).

  We also require that the unit type $\mathbf{U}$ contains a single value $\star$, and that the Boolean type $\mathbf{B}$ data values true and false, which are fixed points of **eval**. Finally, we assume that the data expression language contains tupling $\langle e_0, e_1, e_2 \rangle$ and projections $\pi_i(\langle \vec{v} \rangle) = e_i$; this reduces multiple value arguments to a single value argument.

- A *process type* $\mathbf{P}$. We define the *parameterized process type* $\mathcal{P}$ to be[3]

$$\mathcal{P} = \mathcal{V} \to \mathbf{P}. \tag{7}$$

  We identify $\mathbf{P} = \mathbf{U} \to \mathbf{P}$, and use products, so that all parametrized processes have exactly one parameter; this is a convenience with no technical importance. We use $\mathcal{T}, \ldots$ to range over both parametrized process and data types.

- A set of *operation symbols* $o \in \mathsf{OpSym}$, each with a type $\mathcal{T}_{o,1} \times \cdots \times \mathcal{T}_{o,n_o} \to \mathcal{P}_o$. Note that the arguments may have arbitrary types, but that the result must be a parameterized process.

  We assume an infinity of *parametrized process variables* $x, \ldots$ for each parameterized process type. Using these variables and the operation symbols, we define the set of *parametrized process terms*, ranged over by $p, \ldots$. These terms are built in the obvious way: by applying operation symbols to terms of the correct types — $o(\vec{t})$ — and by λ-abstraction and application of data types, $\lambda v.p$ and pe. Here we use $t, \ldots$ to range over both parametrized process terms and data expressions. We will use $z, \ldots$ to range over both data and parametrized process variables.

  For example, in Section 2.2, the term $a!e.p$ with e an integer expression is interpreted as an operation $o_!$ with type $\mathtt{Chan} \times \mathtt{int} \times \mathbf{P} \to \mathbf{P}$. Similarly, the receive term $a?v.p$ will be interpreted as a binary operation $o_? : \mathtt{Chan} \times (\mathtt{int} \to \mathbf{P}) \to \mathbf{P}$, and the syntax $a?v.p$ will be regarded as syntactic sugar for $o_?(a, \lambda v.p)$.

[3] Our types are all first-order: functions from values to processes. Higher-order processes introduce a variety of complications; see [23, 9].

- A set of *transition symbols* $\kappa \in \mathsf{TransSym}$, each with a type $\mathbf{P} \times \mathcal{V}_\kappa^{\text{in}} \times \mathcal{V}_\kappa^{\text{out}} \times \mathcal{P}_\kappa^{\text{out}}$. As mentioned in Section 2.1, we will keep track of where information comes from: the input types $\mathbf{P}$ and $\mathcal{V}_\kappa^{\text{in}}$ tell what values enter the predicate, and $\mathcal{V}_\kappa^{\text{out}}$ and $\mathcal{P}_\kappa^{\text{out}}$ tells what values leave it. To emphasize this information flow, we write *transitions* in the notation $p \overset{e,e'}{\rightarrow}_\kappa p'$, where p, e, e', and p' are terms which may contain free variables. We write $p \overset{e'}{\rightarrow}_\kappa p'$ as an abbreviation for $p \overset{\star,e'}{\rightarrow}_\kappa p'$.

- A set of *test symbols* $\kappa \in \mathsf{TestSym}$, each with a type $\mathbf{P} \times \mathcal{V}_\kappa^{\text{in}} \times \mathcal{V}_\kappa^{\text{out}}$. Again, we will keep track of where information comes from: the input types $\mathbf{P}$ and $\mathcal{V}_\kappa^{\text{in}}$ tell what values enter the predicate, and $\mathcal{V}_\kappa^{\text{out}}$ tells what value leaves it. We write *tests* in the notation $p \overset{e,e'}{\rightarrow}_\kappa$, where p, v and v' are terms which may contain free variables. Also, we write $p \overset{e'}{\rightarrow}_\kappa$ as an abbreviation for $p \overset{\star,e'}{\rightarrow}_\kappa$.

  Judgements, which are ultimately what the SOS system computes, are either transitions or tests. Frequently, it is convenient to treat judgements uniformly. To this end, we introduce a special *unit process* $\mathbf{p}_\star$, and write $p \overset{e,e'}{\rightarrow}_\kappa \mathbf{p}_\star$ for the test $p \overset{e,e'}{\rightarrow}_\kappa$.

3.2 Formulas

Formulas will be used for antecedents of rules. They have the following forms:

- Any judgement is a formula.
- Any data expression of type $\mathbf{B}$ is a formula.
- $\mathbf{pick}\, x^{\mathcal{V}} : \mathcal{V}$ is a formula, called a *picking formula*.

Judgements and Boolean expressions have the obvious intent. Picking formulas are used in rules involving unbounded choice. For example, the early rule for input, (5), will have an antecedent choosing v. We present both sugared and unsugared forms.

$$\frac{\mathbf{pick}\, v : \mathtt{int}}{a?w.p \overset{a?v}{\rightarrow} p[w := v]} \qquad \frac{\mathbf{pick}\, v : \mathtt{int}}{o_?(a, x^{\mathtt{int}\rightarrow\mathbf{P}}) \overset{\langle a,v\rangle}{\rightarrow}_? xv}$$

Note that we use λ-calculus application in place of an explicit substitution operator.

3.3 Rules

We now define the class of Data-SOS rules. We define three classes: Super-SOS, Data-SOS, and Special-SOS, in order of decreasing generality. We must first define the input and output variables for each formula. Let $\mathsf{vars}(\vec{t})$ be the set of free variables appearing in $\vec{t}$.

$\mathsf{IV}\left(p \overset{e,e'}{\rightarrow}_\kappa p'\right)$	$=$	$\mathsf{vars}(p,e)$	$\mathsf{OV}\left(p \overset{e,e'}{\rightarrow}_\kappa p'\right)$	$=$	$\mathsf{vars}(p',e')$
$\mathsf{IV}(e)$	$=$	$\mathsf{vars}(e)$	$\mathsf{OV}(e)$	$=$	$\emptyset$
$\mathsf{IV}(\mathbf{pick}\, x : \mathcal{V})$	$=$	$\emptyset$	$\mathsf{OV}(\mathbf{pick}\, x : \mathcal{V})$	$=$	$\{x\}$

(8)

If $\mathcal{H}$ is a sequence of formulas, let $\mathsf{OV}(\mathcal{H}) = \bigcup_{\varphi\in\mathcal{H}} \mathsf{OV}(\varphi)$.

Definition 3.1 *A Super-SOS rule for a transition or test symbol κ has the form*

$$\frac{\mathcal{H}}{o(\vec{z})v_0 \stackrel{v,e}{\rightarrow}_\kappa p} \tag{9}$$

where:

1. *o is an operation symbol, $\vec{z}$, v_0, and v are variables*[4]*, e is a data expression, p is a parameterized-process term if κ is a transition symbol and $\mathbf{p}_\star$ otherwise, and $\mathcal{H}$ a finite or infinite sequence of formulas.*[5] *Let φ_0 be the entire conclusion, which is a formula.*
2. *Each transition appearing in $\mathcal{H}$ has the form $p' \stackrel{e',v'}{\rightarrow}_{\kappa'} y$, where v' and y are variables.*
3. *Each test appearing in $\mathcal{H}$ has the form $p' \stackrel{e',v'}{\rightarrow}_{\kappa'}$, where v' is a variable.*
4. *The rule has proper dataflow. That is, the input variables of each antecedent are either inputs of the rule as a whole, or of preceding antecedents; and the output variables of each antecedent are fresh. Formally, if $\mathcal{H} = \mathcal{H}_0 \cdot \varphi \cdot \mathcal{H}_1$, then $\mathsf{IV}(\varphi) \subseteq \mathsf{IV}(\varphi_0) \cup \mathsf{OV}(\mathcal{H}_0)$, and $\mathsf{OV}(\varphi)$ is disjoint from $\mathsf{IV}(\varphi_0) \cup \mathsf{OV}(\mathcal{H}_0)$.*
5. *The variables in the conclusion are either inputs to the rule, or outputs from some antecedent. That is, $\mathsf{OV}(\varphi_0) \subseteq \mathsf{IV}(\varphi_0) \cup \mathsf{OV}(\mathcal{H})$.*

A Super-SOS language consists of a Data-SOS signature and a set of Super-SOS rules over that signature.

For example, these are not Super-SOS:

$$\frac{\mathbf{pick}\, n : \mathtt{int}, \quad x \clubsuit y \stackrel{4,2n}{\rightarrow}_1 w}{x \spadesuit y \stackrel{z,n}{\rightarrow}_2 w} \qquad \frac{}{(x \spadesuit y) \spadesuit z \stackrel{\star}{\rightarrow}_3 x} \qquad \frac{x_1 \stackrel{\star}{\rightarrow}_5 y, \quad x_2 \stackrel{\star}{\rightarrow}_5 y}{x_1 \spadesuit x_2 \stackrel{\star}{\rightarrow}_6 y}$$

but these are:

$$\frac{\mathbf{pick}\, n : \mathtt{int}, \mathsf{even}(n), \quad x \clubsuit y \stackrel{4,n}{\rightarrow}_1 w}{x \spadesuit y \stackrel{z,n/2}{\rightarrow}_2 w} \qquad \frac{}{(x \spadesuit y) \stackrel{\star}{\rightarrow}_3 x} \qquad \frac{x_1 \stackrel{\star}{\rightarrow}_5 y, \quad y \stackrel{\star}{\rightarrow}_5 z}{x_1 \spadesuit x_2 \stackrel{\star}{\rightarrow}_6 z} \tag{10}$$

Terms of the form x and xe, where x is a parameterized process variable and e is a data expression, are called *source items*; they will be used heavily in antecedents.

Definition 3.2 *A Super-SOS rule is in* Data-SOS *format if the judgement antecedents concern only source items; that is, if they have the form $xe \stackrel{e',w}{\rightarrow}_{\kappa_i} z$, where x, w, and z are variables, and e and e' are data expressions. A Data-SOS language consists of a Data-SOS signature and a set of Data-SOS rules over that signature.*

The second and third rules in (10) are Data-SOS, with the convention that x is sugar for $x\star$.

Making suitable finiteness and inductiveness assumptions gains some desirable mathematical properties at some loss of expressive power; see Section 5.

[4]Except at our most pedantic, we omit variables of type $\mathbf{U}$, or replace them by their only possible value $\star$.

[5]Formally, $\mathcal{H}$ is a function from an interval of ordinals $[0, \alpha)$ to formulas, for some ordinal α which need not be finite.

Definition 3.3 *A Data-SOS rule is in Special-SOS format if*

1. $\mathcal{H}$ *is finite, and*
2. *there are no picking antecedents, and*
3. *for each antecedent* $xe \stackrel{e',v}{\to}_{\kappa'} y$*, we have* $x \in \mathsf{IV}(\varphi_0)$.

A Special-SOS language consists of an effectively presented Data-SOS signature (with **eval** *function computable, etc.), and a finite set of Special-SOS rules over that signature.*

3.4 Examples

With liberal use of syntactic sugar, all of the examples in Section 2 are in Data-SOS format. In the full version of this paper we will show that also the process calculi LOTOS [18], μCRL [15], and real-time ACP [2] are covered by our formats.

3.4.1 Imperative Programs

The Plotkin-style SOS requires types for the values that programs compute with (say, `int`), plus a type `var` of the names of variables `x` in the program, plus a type `state` = `var` $\to$ `int` for states σ with a pointwise update operation $\sigma[\mathtt{x} \mapsto v]$. The programming constructs are translated to operations of appropriate types. The type $\mathbf{P}$ is reserved for those entities which have all the information necessary to compute, represented in the Plotkin semantics by $\langle p, \sigma\rangle$. The control states need a data state to compute; that is, they have type $\mathtt{sP} = (\mathtt{state} \to \mathbf{P})$.

$o_{:=}$	:	$\mathtt{var} \times \mathtt{int} \to \mathtt{sP}$	$o_{;}$	:	$\mathtt{sP} \times \mathtt{sP} \to \mathtt{sP}$	(11)
o_{while}	:	$\mathtt{int} \times \mathtt{sP} \to \mathtt{sP}$	...			

There are two types of judgments, $\kappa_\downarrow$ and $\kappa_\to$, as in Plotkin's semantics.

Typing			Judgement	
$\kappa_\downarrow$	:	$\mathbf{P} \times \mathbf{U} \times \mathtt{state}$	$p\sigma \stackrel{\sigma'}{\to}_{\kappa_\downarrow}$	(12)
$\kappa_\to$	:	$\mathbf{P} \times \mathbf{U} \times \mathtt{state} \times \mathtt{sP}$	$p\sigma \stackrel{\sigma'}{\to}_{\kappa_\to} p'$	

The operational rules are the obvious translations of (2). Suitable syntactic sugar will allow them to be written exactly as Plotkin did.

$$\frac{}{o_{:=}(x,e)\sigma \stackrel{\sigma[x\mapsto e]}{\to}_{\kappa_\downarrow}} \qquad \frac{x_1\sigma \stackrel{\sigma'}{\to}_{\kappa_\to} y}{o_{;}(x_1,x_2)\sigma \stackrel{\sigma'}{\to}_{\kappa_\to} o_{;}(y,x_2)} \qquad \frac{x_1\sigma \stackrel{\sigma'}{\to}_{\kappa_\downarrow}}{o_{;}(x_1,x_2)\sigma \stackrel{\sigma'}{\to}_{\kappa_\to} x_2}$$

3.4.2 Translating Channels

The Data-SOS format admits several ways of phrasing communication along channels. Consider the $p \stackrel{a!v}{\to} p'$ of value-passing process algebra. We could express this with a single transition symbol "!", of type $\mathbf{P} \times \mathbf{U} \times (\mathtt{Chan} \times \mathtt{int}) \times \mathbf{P}$. On the other extreme, we could have a family of transition symbols κ_{an} of type $\mathbf{P} \times \mathbf{U} \times \mathbf{U} \times \mathbf{P}$, indexed by channels and values, much in the same way that core languages handle channels and values. Both intermediate cases are possible; *e.g.*, we may have a family of transition symbols κ_a with types $\mathbf{P} \times \mathbf{U} \times \mathcal{V}_a \times \mathbf{P}$. (This form is useful when channels may transmit differently-typed values.)

4 Operational Semantics

4.1 Transition Systems

We generalize the notion of labelled transition system (lts) to cover Data-SOS languages.

Definition 4.1 *A Data-SOS transition system (dts) over a given signature consists of:*

- *A set* $[\![\mathcal{V}]\!]$ *for each data type* $\mathcal{V}$*.*
- *A set* $[\![\mathbf{P}]\!]$ *of* process states. *Define* $[\![\mathcal{V} \to \mathbf{P}]\!]$ *to be the set of all functions*[6] *from* $[\![\mathcal{V}]\!]$ *to* $[\![\mathbf{P}]\!]$*.*
- *A relation* $[\![\kappa]\!] \subseteq [\![\mathbf{P}]\!] \times [\![\mathcal{V}_\kappa^{\mathrm{in}}]\!] \times [\![\mathcal{V}_\kappa^{\mathrm{out}}]\!] \times [\![\mathcal{P}_\kappa^{\mathrm{out}}]\!]$ *for each judgement symbol* κ*.*

A dts gives semantics to a Data-SOS signature in the same way a lts gives semantics to the signature of core CCS.

4.2 Semantics

We may associate a dts to a Data-SOS language $\mathcal{D}$ in a fairly obvious way. First, we define the *evaluation* semantics, which are utterly standard typed call-by-value λ-calculus, with rules

$$\frac{e \text{ closed}}{(\lambda x.p)e \Rightarrow p[x := e]} \qquad \frac{e \text{ closed} \quad e \neq \mathbf{eval}(e)}{e \Rightarrow \mathbf{eval}(e)}$$

$$\frac{p \Rightarrow p'}{pe \Rightarrow p'e} \qquad \frac{e \Rightarrow e'}{pe \Rightarrow pe'} \qquad \frac{p \Rightarrow p'}{\lambda x.p \Rightarrow \lambda x.p'} \qquad \frac{t \Rightarrow t'}{o(\ldots,t,\ldots) \Rightarrow o(\ldots,t',\ldots)}$$

The second rule makes the operational semantics of data expressions trivial; we assume that we can evaluate them instantaneously. (This is of course false if the value domains include, say, untyped λ calculus.) Relation $\Rightarrow$ is confluent and strongly normalizing on closed terms. The normal forms of closed terms of type $\mathbf{P}$ have the form $o(\vec{t})e$; that is, they have a leading operation symbol with its arguments, possibly applied to a value argument.

We next define a *proof* of a closed judgement $\varphi = \left(p \overset{e,e'}{\to}_\kappa p'\right)$. The main subtlety (which is not very subtle) in the definition, is that we require that p, e, e' and p' must be reduced to their $\Rightarrow$-normal form. A proof-tree for φ has as its root a node labelled with a rule ρ for o and an instantiation I of the variables of ρ as closed terms of the proper type. For judgement antecedent φ' of ρ, the proof tree has a child of the root containing a proof of the normal form of $I(\varphi')$. Each Boolean antecedent e must have $\mathbf{eval}(I(e)) = \mathsf{true}$. Picking antecedents are automatically satisfied, as **pick** $x : \mathcal{V}$ simply requires $I(x)$ to be of type $\mathcal{V}$, which is part of the definition of instantiation.

Definition 4.2 *The* standard dts of $\mathcal{D}$ *has* $[\![\mathcal{T}]\!]$ *equal to the set of normal forms of closed terms of type* $\mathcal{T}$*, for each type* $\mathcal{T}$*, and* $[\![\kappa]\!]$ *the provable quadruples.*

The standard dts is the "right" dts in a sense that can be formalized in familiar ways.

We ambiguously write $p \overset{v,v'}{\to}_\kappa p'$ when this holds in the standard dts. This accords with the standard use of, say, $a!v.p \overset{a!v}{\to} p$ as both a formula and the statement that a transition happens.

[6]Note that there may be spurious values in functional domains. For example, if $\mathcal{V}$ is a Scott domain, there are set-theoretic functions which do not respect the domain order and thus ought to be meaningless. We know no problems arising from these spurious values: they do not invalidate our theory, though they may render the model inappropriate. The matter remains to be investigated more fully.

4.3 Comparison with Other Rule Formats

We have indeed gained expressive power over most previous rule formats. In the following theorem, the notion of language equivalence is very fine: it holds up to isomorphism of proof trees.

Theorem 4.3

1. *Let $\mathcal{L}$ be a GSOS [10] language. Then there is a Special-SOS language $\mathcal{L}'$ with equivalent operational semantics.*
2. *Let $\mathcal{L}$ be a pure tyft [16] language. Then there is a Super-SOS language $\mathcal{L}'$ with equivalent operational semantics.*
3. *The converses to 1 and 2 do not hold.*

Proof: For pure tyft, this is straightforward. GSOS languages allow negative antecedents $x \not\xrightarrow{a}$, but these can be programmed as positive tests $\kappa_{\neg a} : \mathbf{P} \times \mathbf{U} \times \mathbf{U}$. $\triangle\!\!\!\!+$

4.4 Bisimulation

The definition of strong bisimulation is the natural adaptation of the usual notion to this setting.

Definition 4.4 *Let $\smile$ be a binary relation on process states. We extend $\smile$ to parametrized process types pointwise: $q \smile q'$ (for $q, q' \in [\![\mathcal{V} \to \mathbf{P}]\!]$) iff $\forall u \in [\![\mathcal{V}]\!].qu \smile q'u$.*

A symmetric binary relation $\smile$ between process states of the same type is a bisimulation relation *iff, whenever $p \smile p'$ and $p \xrightarrow{v,w}_{\kappa} q$, then $p' \xrightarrow{v,w}_{\kappa} q'$ for some q' such that $q \smile q'$.*

Two processes are strongly bisimilar, *$p \leftrightarrow p'$, iff there is a bisimulation relation $\smile$ with $p \smile p'$.*

The results of [10, 16] extend to this setting, giving the main theorem:

Theorem 4.5 *Bisimulation is a congruence for all Super-SOS languages. That is, if $\vec{p} \leftrightarrow \vec{p}'$, then $o(\vec{p}) \leftrightarrow o(\vec{p}')$. Bisimulation is a fortiori a congruence for all Data-SOS and Special-SOS languages.*

5 Finiteness and Effectiveness

The operational semantics for Data-SOS languages are in general not recursive. Let $\mathsf{halts}(m, i)$ be the recursive predicate, "Turing machine number m halts in $\leq i$ steps."

$$\frac{\textbf{pick}\, i : \mathsf{int}, \quad \mathsf{halts}(m, i)}{o(m) \xrightarrow{a} \mathbf{0}} \tag{13}$$

Then $o(m) \xrightarrow{a} \mathbf{0}$ iff Turing machine m halts, which is of course undecidable. A somewhat more subtle example appears in [7], showing that $\xrightarrow{a}$ is undecidable in CCS plus the operations[7]

$$\frac{x \xrightarrow{a} y, \quad y \xrightarrow{a} z}{x/a \xrightarrow{a} z} \qquad \frac{x \xrightarrow{a} y, \quad y \xrightarrow{b} z}{x/a \xrightarrow{b} z} \qquad \frac{}{\beta(x) \xrightarrow{a} x + \beta(x)/a/a} \tag{14}$$

[7]Briefly, $\beta(M)/a/a \xrightarrow{b} N$ iff $M \xrightarrow{a^k b} N$ for some k. Let $M(k)$ run Turing machine k on blank tape, emitting a's as it computes and a b when it finishes. Then $\beta(M)/a/a \xrightarrow{b}$ iff $M(k)$ halts.

Furthermore, Data-SOS calculi are not finitely branching in general; that is, for a given p, v, and κ, there may be infinitely many pairs p', v' with $p \overset{v,v'}{\rightarrow}_\kappa p'$. We have seen this as a feature in Section 5, where $a?x.p$ has a nondeterministic choice for every possible value of x. This is mitigated by the fact that for a given a and v, there are only finitely many p' with $p \overset{a?v}{\rightarrow} p'$. In general, there is no such mitigating fact. We may have rules like

$$\frac{\mathbf{pick}\, i : \mathsf{int}}{o(p) \overset{a}{\rightarrow} o'(p, i)}$$

which make hidden countably nondeterministic choices.

The Special-SOS rule format gives sufficient conditions for recursiveness and finite branching. There are three ways that these properties might fail: infinitary language definition, picking formulas as in (13), and two-step antecedents $x \overset{a}{\rightarrow} y \overset{b}{\rightarrow} z$ as in (14). Definition 3.3 forbids these features, and we obtain:

Theorem 5.1 *Let $\mathcal{L}$ be a Special-SOS language. Then, for any closed p and value v, the set $\left\{\langle \kappa, p', v' \rangle : p \overset{v,v'}{\rightarrow}_\kappa p'\right\}$ is finite and computable in canonical index.*

5.1 Expressiveness

The restriction to Special-SOS format is substantial, and excludes such things as the early semantics of value-CCS of Section 2.2.1 — which are not finitely branching, though they do enjoy some computability properties. The examples of Sections 2.1 and 2.2.2 are Special-SOS.

Corollary 5.2 *The Special-SOS format is less expressive than the Data-SOS format. In particular, the infinitely-branching early semantics of $a?x.p$, (5), is not definable in any Special-SOS language.*

By contrast, we have the following:

Theorem 5.3 *Let $\mathcal{L}$ be a Super-SOS language. Then there is a Data-SOS language $\mathcal{L}'$ with the same signature, such that each operation has identical behavior.*

The language $\mathcal{L}$ may be more compact than $\mathcal{L}'$. In the worst case, $\mathcal{L}'$ may be infinite, even if $\mathcal{L}$ is finite. As a less extreme example, suppose that we want to have an operation $o(x_1, x_2, x_3)$ which behaves just like $(x_1 + x_2)\|x_3$, in the process-algebraic setting. This can be said quite simply in Super-SOS:

$$\frac{(x_1 + x_2)\|x_3 \overset{\alpha}{\rightarrow} y}{o(x_1, x_2, x_3) \overset{\alpha}{\rightarrow} y}$$

In the Data-SOS setting, we need to have a set of rules for o, one for each way that $(x_1 + x_2)\|x_3$ can behave: a collection of rules including

$$\frac{x_1 \overset{\alpha}{\rightarrow} y}{o(x_1, x_2, x_3) \overset{\alpha}{\rightarrow} y\|x_3} \qquad \frac{x_1 \overset{a?v}{\rightarrow} y_1, \quad x_3 \overset{a!v}{\rightarrow} y_3}{o(x_1, x_2, x_3) \overset{\tau}{\rightarrow} y_1\|y_3}$$

6 Conclusion

The rule formats we have proposed seem to cover most first-order process algebras and operational semantics for programming languages that have appeared in the literature, covering most standard process calculi, from theoretical ones like CCS, CSP, and ACP to more applied ones like LOTOS and μCRL, with little more than notational changes. Indeed, the basic theory, through strong bisimulation, of all of these languages is a corollary of the theory in this paper.

These methods apply to descriptions of other languages as well; *e.g.*, to most Plotkin-style descriptions of imperative languages [19] as in Section 2.1. The basic results are of some interest for these languages; this is the first formalization of Plotkin's techniques. The bisimulation results apply to such languages, but are less useful: **skip** **skip** ; **skip**.

This will be remedied by treatment of weaker process equivalences, those which declare some aspects of process behavior invisible. This theory has been worked out to a significant extent for systems without data: [8] covers CCS and CSP without values. It seems likely that a straightforward variation of this theory would cover the weak bisimulation theory of CCS, ACP, LOTOS, and μCRL at the very least.

A theory of weak linear-time equivalences would be helpful as well; such a theory would ideally give corollaries like "failures equivalence is adequate for CSP" (or, more importantly, for a new CSP-like language). Other possible work include automatically generating axiom systems from rules, in the style of [1].

References

[1] L. Aceto, B. Bloom, and F. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, May 1994.

[2] J. Baeten and J. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing Science*, 3(2):142–188, 1991.

[3] J. Baeten, J. Bergstra, and J. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, IX(2):127–168, 1986.

[4] J. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *Proceedings CONCUR 93,* Hildesheim, Germany, volume 715 of *Lecture Notes in Computer Science*, pages 477–492. Springer-Verlag, 1993.

[5] J. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol 4.* Oxford University Press, 1994. To appear.

[6] J. d. Bakker, J. Kok, J.-J. Meyer, E.-R. Olderog, and J. Zucker. Contrasting themes in the semantics of imperative concurrency. In J. de Bakker, W. d. Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 51–121. Springer-Verlag, 1986.

[7] B. Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-like Languages.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Aug. 1989.

[8] B. Bloom. Ready, set, go: Structural operational semantics for linear-time process algebras. Technical Report TR 93-1372, Department of Computer Science, Cornell University, Ithaca, New York, Aug. 1993.

[9] B. Bloom. CHOCOLATE: Calculi of Higher Order COmmunication and LAmbda TErms. (To appear in POPL '94), November 1994.

[10] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced: Preliminary report. In *Conference Record of the* 15^{th} *ACM Symposium on Principles of Programming Languages,* San Diego, California, pages 229–239, 1988. Full version available as Technical Report 90-1150, Department of Computer Science, Cornell University, Ithaca, New York, August 1990. Accepted to appear in *Journal of the ACM.*

[11] J. Camilleri and G. Winskel. CCS with priority choice. In *Proceedings* 6^{th} *Annual Symposium on Logic in Computer Science,* Amsterdam, pages 246–255. IEEE Computer Society Press, 1991.

[12] R. Cleaveland. An operational semantics of value passing. In B. Bloom, editor, *Proceedings* 2^{nd} *North American Process Algebra Workshop,* Ithaca, NY. Cornell Technical Report, 1993.

[13] R. Cleaveland and M. Hennessy. Priorities in process algebra. In *Proceedings* 3^{th} *Annual Symposium on Logic in Computer Science,* Edinburgh, pages 193–202. IEEE Computer Society Press, 1988.

[14] R. v. Glabbeek. Full abstraction in structural operational semantics. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 3^{rd} AMAST Conference,* Twente, The Netherlands, June l993, Workshops in Computing, pages 77–84. Springer-Verlag, 1993.

[15] J. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D. Andrews, J. Groote, and C. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computer Science, pages 231–250. Springer-Verlag, 1993.

[16] J. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, Oct. 1992.

[17] M. Hennessy. A fully abstract denotational model for higher-order processes (extended abstract). In *Proceedings 8^{th} Annual Symposium on Logic in Computer Science*, pages 397–408. IEEE Computer Society Press, 1993. Full version in Sussex TR 6/92.

[18] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[19] G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[20] G. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II,* Garmisch, pages 199–225, Amsterdam, 1983. North-Holland.

[21] R. d. Simone. Higher-level synchronising devices in MEIJE–SCCS. *Theoretical Comput. Sci.*, 37:245–267, 1985.

[22] S. Smolka and B. Steffen. Priority as extremal probability. In J. Baeten and J. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 456–466. Springer-Verlag, 1990.

[23] B. Thomsen. A calculus of higher order communicating systems. In *Conference Record of the 16^{th} ACM Symposium on Principles of Programming Languages,* Austin, Texas, pages 143–154, 1989.

[24] F. Vaandrager. Expressiveness results for process algebras. In J. de Bakker, W. d. Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Semantics: Foundations and Applications,* Beekbergen, The Netherlands, June 1992, volume 666 of *Lecture Notes in Computer Science*, pages 609–638. Springer-Verlag, 1993.

;	(2)	Sequencing operator
$\mathcal{V}$	Section 3.1	One of the data types
$\mathbf{eval}(e)$	Section 3.1	Fully evaluate a data expression
$\mathbf{U}$	Section 3.1	Unit type
$\star$	Section 3.1	The one and only expression of type $\mathbf{U}$
$\mathbf{B}$	Section 3.1	Boolean type, with values true and false
$\mathbf{P}$	Section 3.1	Type of Processes
$\mathbf{p}_\star$	Section 3.1	Unit Process, introduced for uniformity of tests and transitions
$\mathcal{P}$	Section 3.1	Any parametrized process type $\mathcal{V} \to \mathbf{P}$
$\mathcal{T}$	Section 3.1	Any type
o	Section 3.1	Operation symbol.
κ	Section 3.1	Judgement symbol, of type $\mathbf{P} \times \mathcal{V}_\kappa^{\text{in}} \times \mathcal{V}_\kappa^{\text{out}} \times \mathcal{P}_\kappa^{\text{out}}$
$p \overset{v,v'}{\rightarrow}_\kappa p'$	Section 3.1	Judgement: process p given conditions v produces value v' and process p'
$\mathbf{pick}\ x : \mathcal{V}$	Section 3.2	Formula, choosing a value x of type $\mathcal{V}$.
$\mathcal{H}$	(9)	Ordinal-length sequence of formulas in the antecedent of a rule
$\cdot$		Sequence concatenation
$\mathsf{IV}(\varphi), \mathsf{OV}(\varphi)$	(8)	Input and Output Variables of a formula φ
$\spadesuit, \clubsuit$		Binary operation symbols with no particular meaning, used in examples.
$[\![\mathcal{V}]\!], [\![\mathcal{P}]\!], [\![\mathcal{V} \to \mathcal{P}]\!]$	Definition 4.1	Semantic domains for data types, processes and parametrized processes
$\Rightarrow$	Section 4.2	λ-calculus evaluation.
I	Section 4.2	Instantiation of a rule ρ: mapping from variables of ρ to closed terms of the proper type.
e		Data expressions
p, q		Terms of parameterized process type
t		Terms of arbitrary type
x, y		Variables of parametrized process type
z		Variables of arbitrary type
$\smile$	Definition 4.4	Candidate strong bisimulation relation
$\leftrightarrow$	Definition 4.4	Strong bisimulation

Figure 1: Notation and Conventions