Eindhoven University of Technology
Department of Mathematics and Computing Science

Axiomatizing Early and Late Input by Variable Elimination

by

Arie van Deursen

95/21

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. M. Rem

Computing Science Report 95/21
Eindhoven, June 1995

Axiomatizing Early and Late Input by Variable Elimination

Arie van Deursen

Formal Methods Group, Dept. of Computing Science

TU Eindhoven, 5600 MB P.O. Box 513, The Netherlands

arie@win.tue.nl, http://www.win.tue.nl/win/cs/fm/arie/

15 January, 1995

Abstract

Variable binding input actions in process algebra expressions can be characterized by *early* as well as by *late* bisimulation, where the distinction is concerned with whether or not variables are instantiated when considering process equivalence. Baeten and Bergstra have given an axiomatization of late and early bisimulation for finite data sets. We illustrate their method by an example, provide the necessary intuition, formulate correctness properties, list errata, and discuss possibilities for future research.

Note: Supported by NWO, the Netherlands Organization for Scientific Research, project 612-16-433, HOOP: Higher-Order and Object-Oriented Processes.

1991 Mathematics Subject Classification: 68Q60, 68Q10, 68Q40

1991 CR Categories: F.1.2, D.3.1, F.3.1, D.1.3.

Additional keywords and phrases: Process algebra, value passing, term rewriting, ACP, ASF+SDF.

Note: An extended abstract appeared in *Algebra of Communicating Processes, ACP'95*, A. Ponse, C. Verhoef, and B. van Vlijmen (eds).

1 Introduction

In process algebra, input actions can have a variable-binding effect, i.e., some value is read and "assigned" to a variable, which later on in the process-expression can be used. A distinction can then be made between "late" and "early" instantiation of variables, a distinction which has consequences for the notion of equality and bisimulation of processes. Baeten and Bergstra have proposed [BB94] a general framework, *Functional Prefix Algebra*, which they use to come up with various algebraic specifications of late and early

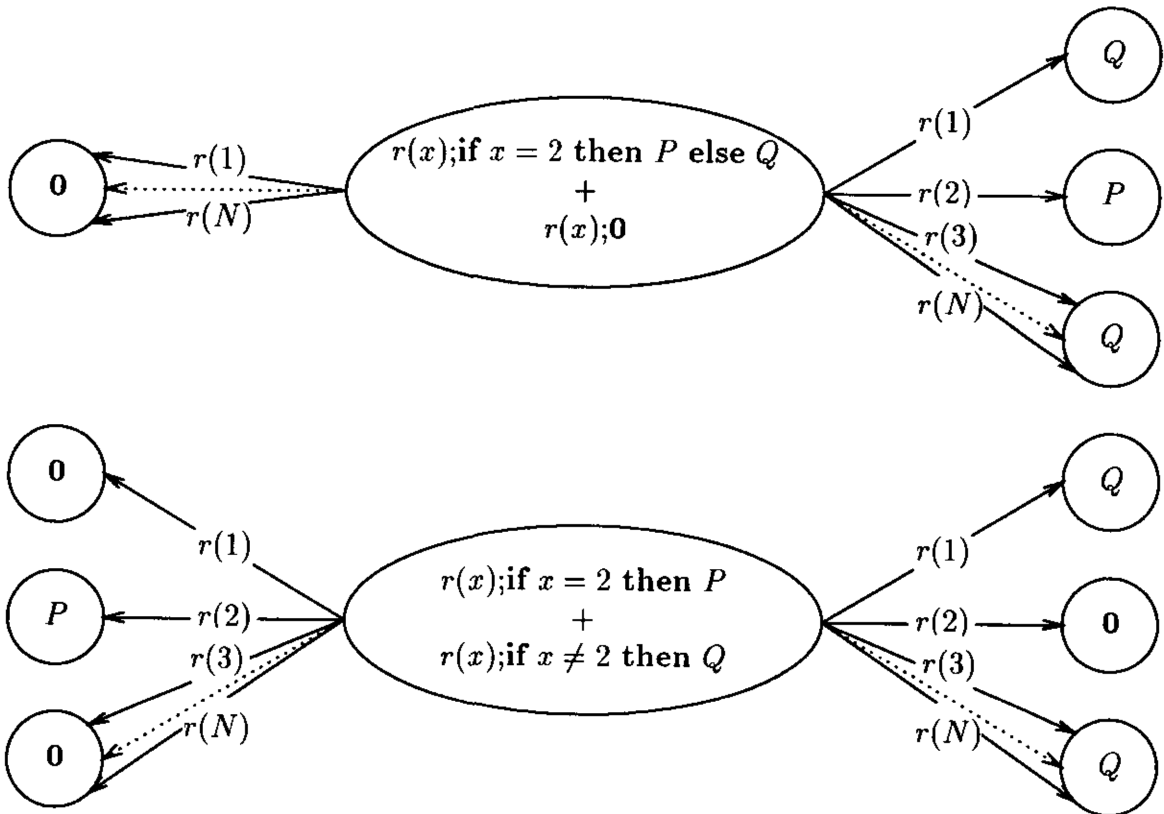

Figure 1: Early Input Transitions

input actions for finite data sets. In these specifications, they cleverly circumvent the need for explicit manipulation of free and bound variables.

Their method is simple and elegant, but the emphasis of their paper is on listing the signatures and axioms. In order to make their article more accessible, we discuss one running example for all specifications given, show how the algebras work by executing the specifications using term rewriting, formulate the claims that should be proven in order to assess the correctness of all specifications, and propose directions for further research.

This paper is to be read in combination with [BB94]. Before studying that paper, you might find it helpful to make the corrections we give in Appendix A.

1.1 Motivating Example

The following example, taken from [MPW91, p.46], illustrates the differences between early and late input (here we are using CCS notation):

$$\begin{aligned} R &= r(x).(\mathbf{if}\ x = 2\ \mathbf{then}\ P\ \mathbf{else}\ Q) + r(x).\mathbf{0} \\ S &= r(x).(\mathbf{if}\ x = 2\ \mathbf{then}\ P) + r(x).(\mathbf{if}\ x \neq 2\ \mathbf{then}\ Q) \end{aligned}$$

In [Mil89] the early approach is taken, and R and S are equal (i.e., they are strongly bisimilar). An input action $r(x)$ is interpreted as an abbreviation for a sum of individual

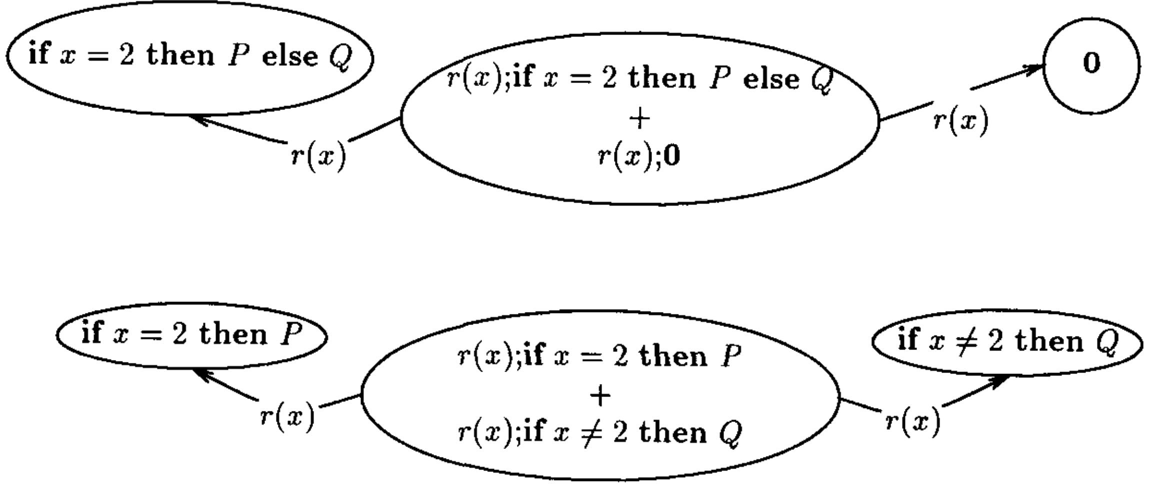

Figure 2: Late Input Transitions

read actions for every possible data value for x, say $\{1, \ldots, N\}$, thus:

$$r(x).Y \quad = \quad \sum_{n=1}^{N} r_n.Y[n/x] \tag{1}$$

Hence, $2 * N$ steps are possible from both R and S, as shown in Figure 1. The steps lead to P, Q or $\mathbf{0}$ nodes, and for both R and S the same transitions lead to the same nodes (although they are given in different order). In other words, they can mimic each other, and hence are bisimilar. This interpretation is referred to as *early*, because the variable x is instantiated as early as possible.

In the alternative, *late*, approach, we do not want to say anything about x's value in advance. We allow ourselves to make $r(x)$ steps, and we see *non-instantiated* variables in the transition labels, as shown in Figure 2 (see [HL95] for a thorough treatment of these "symbolic bisimulations"). This leads to a process equivalence where fewer processes are equal. In this interpretation R and S cannot simulate each other, since once one of the alternatives is chosen for, e.g., R, without having fixed the value of x, it is impossible to find a transition for S which has the same effects for all x's.

The difference between these two interpretations of processes is the theme of this paper. We will distinguish two read-operators, one performing an *early* read, and the other a *late* one. We will axiomatize these read operations in such a way that for

$$R_E \quad = \quad er(x).(\textbf{if } x = 2 \textbf{ then } P \textbf{ else } Q) + er(x).\mathbf{0} \tag{2}$$

$$S_E \quad = \quad er(x).(\textbf{if } x = 2 \textbf{ then } P) + er(x).(\textbf{if } x \neq 2 \textbf{ then } Q) \tag{3}$$

we have $R_E = S_E$, but for

$$R_L \quad = \quad lr(x).(\textbf{if } x = 2 \textbf{ then } P \textbf{ else } Q) + lr(x).\mathbf{0} \tag{4}$$

$$S_L \quad = \quad lr(x).(\textbf{if } x = 2 \textbf{ then } P) + lr(x).(\textbf{if } x \neq 2 \textbf{ then } Q) \tag{5}$$

we will not be able to show $R_L = S_L$. The axiomatization is characterized by bound-variable elimination: any expression with only bound (by early or late input actions) variables is equal to a term without any variable occurrence (be it bound or free).

One might wonder what the use of *late* input is, as it seems so natural to consider agents R and S to be equivalent. Parrow and Sangiorgi [PS93] provide the following explanation: The late bisimulation equivalence builds on a more refined operational intuition: a process can decide to receive input on a port, and by doing so it becomes a function from values to processes. There are thus two atomic "events" corresponding to an input transition, namely (i) committing on a port and then (ii) instantiating the function with the received value. These atomic events should be exactly matched between processes equivalent under late bisimulation. Late bisimulation is particularly relevant to *mobile* process calculi, such as the π-calculus [MPW92] (we will not yet consider mobility here – this is deferred to a forthcoming paper).

1.2 FPA$_{\text{ECA}}$ and its Subalgebras

Figure 3 gives an overview of the various algebraic specifications we consider, illustrating which symbols occur in each of them. The first few signatures are more or less standard ACP; the special operations are in FPA ("Functional Prefix Algebra") and Prefixing. All these together constitute a large specification called FPA$_{\text{ECA}}$, and we will first try to understand its initial algebra. Next, we will study four subalgebras of FPA$_{\text{ECA}}$, referred to as BVMA, VMC, VPC, and VPA (the names will be explained in Section 3). BVMA amounts to standard ACP with Read/Send communication (as in [BK86]); VMC is early read with free and bound variables, while VPC is its late counterpart; VPA, finally, is an intriguing setting where we have (late) input actions but no variables.

1.3 Reduced Model Specifications

When considering a sub-signature $M \in \{\text{BVMA}, \text{VMC}, \text{VPC}, \text{VPA}\}$ of FPA$_{\text{ECA}}$, we change the operations not in M into *hidden* functions in FPA$_{\text{ECA}}$. This amounts to taking the initial model of FPA$_{\text{ECA}}$, written $I(\text{FPA}_{\text{ECA}})$, and eliminating those operations corresponding to functions to be hidden, a construction called *taking the M-reduct of* FPA$_{\text{ECA}}$, written $I(\text{FPA}_{\text{ECA}})_{|M}$. This does *not* change the equalities that are valid between the remaining operations; the hidden functions are just invisible, not deleted.

After eliminating operations from $I(\text{FPA}_{\text{ECA}})$, it is necessary to throw out carrier elements, as for some values (those corresponding to *hidden constructor terms*) there is no notation anymore. We can make our algebra smaller by reducing the carrier sets, until the *minimal subalgebra* remains, which is written $\langle I(\text{FPA}_{\text{ECA}})\rangle_M$.

In this way, one can characterize the intended algebras for the signatures of BVMA, VMC, VPC, and VPA. In a sense, however, this is unsatisfactory, as the specifications contain too many hidden functions and equations. Therefore, as a last step, we will look for shorter, *direct*, axiomatizations for the reduced models of BVMA, VMC, VPC, and VPA.

$$
\begin{array}{rcl}
\text{ACP} & = & \{\ \delta,\ +,\ \cdot,\ \|,\ \mid,\ \lfloor\!\lfloor,\ \partial_H\ \} \\
\text{Val} & = & \{\ 1,\ldots,N\ \} \\
\text{Var} & = & \{\ v_1,\ldots,v_m\ \} \\
\text{D} & = & \text{Val} \cup \text{Var} \\
\text{Booleans} & = & \{\ \neg,\ \wedge,\ \vee,\ eq_D(p,q),\ :\!\rightarrow,\ \ \triangleleft\ \triangleright\ \} \\
\text{Substitution} & = & \{\ P[p/v],\ \beta[p/v],\ q[p/v]\ \} \\
\text{RSC} & = & \{\ r_m(i),\ s_m(i),\ c_m(i)\ \} \\
\text{FPA} & = & \{\ \langle P_1,\ldots,P_N\rangle,\ \bullet_N,\ \circ_N,\ \wedge\ \} \\
\text{Prefixing} & = & \{\ er_m(v);_,\ lr_m(v);_,\ s_m(i);_,\ c_m(i);_,\ r_m,\ \lambda v._\ \} \\
\text{Restriction} & = & \{\ \backslash_\delta\ \} \\
 & & \\
\text{FPA}_{\text{ECA}} & = & \text{ACP} \cup \text{D} \cup \text{Substitution} \cup \text{Booleans} \\
 & & \quad \cup\ \text{RSC} \cup \text{FPA} \cup \text{Prefixing} \cup \text{Restriction} \\
 & & \\
\text{BVMA} & = & \text{ACP} \cup \text{Val} \cup \text{Booleans} \cup \text{RSC} \\
\text{VMC} & = & (\text{FPA}_{\text{ECA}} - \text{FPA}) - \{\ \cdot,\ \partial_H,\ lr_m(v);_,\ r_m(i),\ \lambda v._\ \} \\
\text{VPC} & = & (\text{VMC} - \{\ er_m(v);_\ \}) \cup \{\ lr_m(v);_\ \} \\
\text{VPA} & = & \text{ACP} \cup \text{Val} \cup (\text{RSC} - \{r_m(i)\}) \cup \{r_m\} \cup (\text{FPA} - \{\circ_N, \wedge\})
\end{array}
$$

Figure 3: Signatures for various algebras.

Naturally, the initial models of these direct axiomatizations should be equivalent to the minimal sub-algebras of the reduced models.

Reduced models are well-known in algebraic specification, and discussed, e.g., in [EM85, Section 6.8] or [Wir90, Section 2.2]. The difference between hiding and deleting is elaborated on in [BH93].

1.4 Understanding by Experiment

In order to gain some intuition concerning the specifications of FPA_{ECA}, BVMA, VMC, VPC, and VPA as presented in [BB94], we will sometimes explain them by showing what the equations do when they are interpreted as rewrite rules. For instance, when studying FPA_{ECA}, we reduce terms R_E and S_E (see Section 1.1) to their normal forms, and see that they are the same (as they should with early input).

We have used the ASF+SDF Meta-environment[BHK89, Kli93] to perform such experiments.

ASF+SDF supports execution of specifications based on term rewriting. Moreover, its literate specification facilities (in the sense of [Knu92]) translate ASCII to LaTeX, allowing one to incorporate machine-checked specifications directly as texts in documentation or

technical reports. The full specifications are given in the appendices [1]

Whenever we mention "normal forms" in this document, we refer to normal forms obtained by orienting the equations from left to right, and executing them as rewrite rules. To avoid non-termination, we assume rewriting to take place modulo commutativity and associativity of certain operators. To emphasize that we obtained a certain result using rewriting, we will sometimes use an arrow instead of an equality symbol.

2 Functional Prefix Algebra

We first discuss FPA_{ECA} step by step, following the signatures of Figure 3. The definitions of ACP are very similar to the standard ones and not discussed any further (see, e.g., [BV95]). We start with a setting without atoms, which is gradually extended with atoms (e.g., in the signature of RSC) which all deal with communication. Missing is the communication function γ: communication between atoms is axiomatized directly. A noteworthy point, finally, is that we restrict ourselves to *finite* processes, and *finite* data.

2.1 Booleans, Data Values and Data Variables

For input actions, we need a notion of data values. We restrict ourselves to a *finite* set *Val* of data values, which are assumed to be the numbers $\{1, \ldots, N\}$. We introduce a sort *Bool* with constants T and F, as well as operators like $\wedge, \vee, \neg$. Equality over data values is a function $\text{eq}_{Val} : Val \times Val \rightarrow Bool$ which is either T or F for any value $i, j \in Val$.

We moreover assume a (countably) infinite collection $Var = \{\mathtt{v}, \mathtt{v1}, \mathtt{v2}, \ldots\}$. The union of *Val* and *Var* is the full data sort D. When writing equations involving data, i, j denote elements from *Val*, v, w from *Var*, and p, q from D.

Over this full sort D, we need an equality function. As we do not know whether two variables v and w are equal (we are interested in equality over their instantiated values), we can only give the following equations:

$$\begin{aligned} \text{eq}_D(i, j) &= \text{eq}_{Val}(i, j) \\ \text{eq}_D(p, p) &= T \\ \text{eq}_D(p, q) &= \text{eq}_D(q, p) \end{aligned}$$

Thus, a term like $\text{eq}_D(\mathtt{v1}, \mathtt{v2})$ is neither equal to T, nor to F: The definition of eq_D is not *sufficiently complete*.

This has consequences for the axiomatization of other operators, such as $\vee, \wedge, \neg$, etc. Just specifying these operators for the T and F cases is not sufficient. For example, when we will try to prove the equality between R_E and S_E (our example from Section 1.1), we will also need (β a variable over *Bool*):

$$\beta \wedge \beta = \beta \tag{6}$$

$$\beta \wedge \neg\beta = F \tag{7}$$

[1] The full sources are also available from ftp://ftp.win.tue.nl/pub/techreports/arie/.

$$
\begin{array}{lll}
\textbf{sorts:} & \mathit{Port} & \\
\textbf{functions:} & r_{_}(_), s_{_}(_), c_{_}(_) : \mathit{Port} \times \mathit{Val} \to A^c & \\
\textbf{equations:} & r_m(i) \mid s_m(i) = c_m(i) & \\
 & r_m(i) \mid s_k(j) = \delta & \textbf{when } k \neq m \textbf{ or } i \neq j \\
 & r_m(i) \mid r_k(j) = \delta & \\
 & s_m(i) \mid s_k(j) = \delta & \\
 & c_m(i) \mid a = \delta &
\end{array}
$$

Figure 4: RSC Communication Primitives

In [Sio64] some equational bases for Boolean algebras are discussed in more detail.

In [BB94] the if-then and if-then-else operators (written $:\!\to$ and $\triangleleft \quad \triangleright$) over processes are introduced as well. The if-then operator is equal to δ if the condition is false. Again, the non-standard elements force us to give several extra equations indicating how certain combinations of operators can be eliminated. Two equations not mentioned in [BB94, Section 3.2] (but again needed to show $R_E = S_E$) are:

$$\beta_1 :\!\to (\beta_2 :\!\to X) = (\beta_1 \wedge \beta_2) :\!\to X \tag{8}$$

$$\beta_1 :\!\to X + \beta_2 :\!\to X = (\beta_1 \vee \beta_2) :\!\to X \tag{9}$$

These equations are provided in [BB92].

In addition to Booleans and Data, we need a substitution operator, which changes a variable occurring in a process, Boolean expression, or data expression into a new data element (either a value or a variable).

2.2 Read/Send Communication

In ACP, communication involving data typically comes with the Read/Send communication primitives shown in Figure 4, introduced in [BK86]. The axioms are standard (although here not formulated using the γ notation). We explicitly mention port names as a sort, where k, m are variables ranging over this sort[2].

It is important to realize that these primitives only deal with values, not with variables! In the next sections, we will see how input actions (or at least the early ones) involving variables can be translated into the Read/Send primitives.

The standard CCS restriction operation [Mil89] can be defined on top of these Read/Send primitives (translating to δ):

$$
\begin{array}{rcl}
_ \setminus_\delta _ & : & P \times \mathit{Port} \to P \\
X \setminus_\delta m & = & \partial_{\{r_m(i), s_m(i) \mid i \in \mathit{Val}\}}(X)
\end{array}
$$

[2] In such a setting there is no need to regard equations as "axiom schemas".

2.3 Sequences and Prefixing

The functions introduced so far were probably more or less familiar: from now on we deal with operations especially invented for functional prefix algebra. First, we need a new sort P^N, with the following constructor:

$$\langle _, \dots, _ \rangle : P \times \cdots \times P \rightarrow P^N$$

which builds a sequence of N processes (recall that N is the number of data values). It will be used to represent a family of processes indexed by the value of some variable. It is best understood in combination with an *expansion* operator written $\lambda v.X$, which takes a variable v and a process X, and produces a sequence of N variants of X, one for each instantiation of X:

$$\lambda v.X = \langle X[1/v], \dots, X[N/v] \rangle \tag{10}$$

When modelling input operations, we will prefix such sequences with atomic read actions. To that end, we introduce

$$r_{_} : Port \rightarrow A^c$$

an atomic action which just models "I am going to do some kind of read". It can be put in front of process sequences by means of the following two prefixing operators:

$$_ \circ_N _, _ \bullet_N _ \quad : \quad A \times P^N \rightarrow P$$

The $\circ_N$ operator is called *early functional prefix*, the $\bullet_N$ *late functional prefix*. The following axiom takes care that an early prefix can always be eliminated:

$$a \circ_N \langle X_1, \dots, X_N \rangle \quad = \quad (a \hat{\ } 1) \cdot X_1 + \cdots + (a \hat{\ } N) \cdot X_N \tag{11}$$

where an extra operator

$$_\hat{\ }_ : A \times Val \rightarrow A$$

is used. This operator can be paraphrased as "confront an action with its ith data value". It can be defined for various atomic actions. The crucial equation is for the r_m action from above:

$$r_m \hat{\ } i \quad = \quad r_m(i) \tag{12}$$

The remaining equations defining ^ take care that it is equal to δ for other atoms. With the $r_m(i)$ occurring in the right-hand side, we return to our simple Read/Send communication primitives, and combining (11) and (12) we have, e.g.:

$$r_m \circ_N \langle X_1, \dots, X_N \rangle \rightarrow r_m(1) \cdot X_1 + \cdots + r_m(N) \cdot X_N$$

Such an elimination is in general not possible for the late prefix.[3] Therefore, the $\bullet_N$ operator acts like a "normal form" or equivalence class representative, which is not equal to a simpler form. Only under suitable circumstances, i.e., when we know that we are indeed communicating, the $\bullet_N$ can be translated to an early prefix. Axioms expressing this are, e.g.:

$$\begin{aligned}(a \bullet_N F) \mid b &= (a \circ_N F) \mid b \\ (a \bullet_N F) \mid b \cdot X &= (a \circ_N F) \mid b \cdot X\end{aligned}$$

Moreover, certain combinations can be transformed, axiomatized, e.g., by:

$$(a \bullet_N \langle X_1, \ldots, X_N \rangle) \,\|\!_\, Y = a \bullet_N \langle X_1 \parallel Y, \ldots, X_N \parallel Y \rangle$$

Finally, one type of communication is guaranteed to be non-effective:

$$(a_1 \bullet_N F_1) \mid (a_2 \bullet_N F_2) = \delta$$

This axiom is baptized the *Early Communication Axiom* (ECA), which states that two late read actions have no communication possibility.

The above should explain the intuition of the various prefixing and sequencing operators. The complete set of axioms is given in [BB94, Table 4], except for r_m and λ which are specified in [BB94, Table 8].

2.4 Early and Late Read

2.4.1 Axiomatization

With the above preliminaries, defining early and late input actions is straightforward. We introduce them as variable binding prefix operators, with the following signature:

$$er_(_);_,\ lr_(_);_ \;:\; Port \times Var \times P \rightarrow P$$

The axioms now simply are:

$$er_m(v); X = r_m \circ_N \lambda v.X \tag{13}$$

$$lr_m(v); X = r_m \bullet_N \lambda v.X \tag{14}$$

For the early read, we can prove from (13), (10), (11), and (12) the following identity:

$$er_m(v); Y = \sum_{n=1}^{N} r_m(n) \cdot Y[n/v]$$

which we encountered before as an informal characterization of the early read.

[3]Observe that [BB94, Section 3.8] indeed does not list $\bullet_N$ as an operator that can be eliminated (it does not list λ either, but that one can be eliminated).

2.4.2 The if-then-else Example Revisited

Let us keep our promises, and see how the early and late versions of R and S behave:

$$\begin{array}{rcl} R_E & = & er_m(\mathtt{v}); \{P \lhd \mathrm{eq}_D(\mathtt{v},2) \rhd Q\} \ + \ er_m(\mathtt{v}); \delta \\ S_E & = & er_m(\mathtt{v}); (\mathrm{eq}_D(\mathtt{v},2){:\!\rightarrow} P) \ + \ er_m(\mathtt{v}); (\neg\mathrm{eq}_D(\mathtt{v},2){:\!\rightarrow} Q) \end{array}$$

With the intuition gained from the previous section, it should be easy to see that:

$$\begin{array}{rcl} R_E & \rightarrow & r_m(1)\cdot Q \ + \ r_m(2)\cdot P \ + \ r_m(3)\cdot Q \ + \ \cdots \ + \ r_m(N)\cdot Q + \\ & & r_m(1)\cdot \delta \ + \ r_m(2)\cdot \delta \ + \ r_m(3)\cdot \delta \ + \ \cdots \ + \ r_m(N)\cdot \delta \end{array}$$

Likewise, we have:

$$\begin{array}{rcl} S_E & \rightarrow & r_m(1)\cdot \delta \ + \ r_m(2)\cdot P \ + \ r_m(3)\cdot \delta \ + \ \cdots \ + \ r_m(N)\cdot \delta \ + \\ & & r_m(1)\cdot Q \ + \ r_m(2)\cdot \delta \ + \ r_m(3)\cdot Q \ + \ \cdots \ + \ r_m(N)\cdot Q \end{array}$$

and by commutativity and associativity of the $+$ we have equality of R_E and S_E.

For the *late* version, we have:

$$\begin{array}{rcl} R_L & = & lr_m(\mathtt{v}); \{P \lhd \mathrm{eq}_D(\mathtt{v},2) \rhd Q\} \ + \ lr_m(\mathtt{v}); \delta \\ S_L & = & lr_m(\mathtt{v}); (\mathrm{eq}_D(\mathtt{v},2){:\!\rightarrow} P) \ + \ lr_m(\mathtt{v}); (\neg\mathrm{eq}_D(\mathtt{v},2){:\!\rightarrow} Q) \\ R_L & \rightarrow & r_m \bullet_N \langle Q, P, Q, \ldots, Q\rangle \ + \ r_m \bullet_N \langle \delta, \ldots, \delta\rangle \\ S_L & \rightarrow & r_m \bullet_N \langle \delta, P, \delta, \ldots, \delta\rangle \ + \ r_m \bullet_N \langle Q, \delta, Q, \ldots, Q\rangle \end{array}$$

Here the "normal forms" are different. Can we conclude that the R_L and S_L must be different as well? This is the case if the rewriting system is terminating and confluent. A demonstration of that requires a careful case distinction of all equations given in [BB94]. The intuition given by the above normalization should convince the reader that it is safe to conclude that $R_L \neq S_L$.

This example illustrates the intended behavior of the equations: the original terms with input actions are reduced to terms without the variable v occurring in it.

2.4.3 Bound Variables and α-Conversion?

The axiomatization of early and late read from Section 2.4.1 does not explicitly refer to the notions of free or bound variables, nor does it include a rule for α-conversion. Nevertheless, the $er_m(v); X$ and $lr_m(v); X$ expressions do have a binding effect in the process X, and the name v can be α-converted. Let us briefly look how this is achieved.

In [BB94] the are no equations specifying the effect of substituting over $er_m(v); X, lr_m(v); X$, and $\lambda v.X$. Nevertheless, the substitution operator can always be eliminated [BB94, Section 3.8]. This is because the operators for which no substitution equations are given can themselves be eliminated (e.g., the $er_m(v); X$ is equal to a summation for every data element, and substitution is defined for the choice operator). In other words: "normalization" of terms involving bound variables can only be achieved by first eliminating (i.e., expanding) all operations involving bindings. As a result, substitutions cannot be used to change bound variables.

Secondly, let us study the counterpart of the α-conversion rule in this algebra. If we define free variables in the usual way, we can use structural induction to prove:

$$w \notin FV(X) \;\; \Rightarrow \;\; er_m(v); X = er_m(w); (X[w/v]) \tag{15}$$

This is fairly obvious, as (provided $w \notin FV(X)$)

$$\begin{aligned} \lambda v.X &= \langle X[1/v], \ldots, X[N/v] \rangle \\ &= \langle X[w/v][1/w], \ldots, X[w/v][N/w] \rangle \\ &= \lambda w.(X[w/v]) \end{aligned}$$

Naturally the same holds for the late read operation.

2.5 Process Prefix and Further Extensions

The $\bullet_N$ and $\circ_N$ operators are "action prefixes" in the sense that their first argument is just a single atomic action. To illustrate the generality of the FPA$_{\text{ECA}}$ setting, we show how it can be used to arrive at a "process prefix" situation, where an arbitrary process can be used to read variables which are used in a subsequent process. The signature we need includes two new core atoms:

$$er_{_}(_)\,,\; lr_{_}(_) \;:\; Port \times Var \rightarrow A^c$$

and a process combinator which should have the binding effect:

$$_;_ \;:\; P \times P \rightarrow P$$

With this signature we can write all terms we could express before, but some extra terms as well. We therefore need a few extra equations:

$$er_m(v) \mid a \;=\; \delta \tag{16}$$

$$lr_m(v) \mid a \;=\; \delta \tag{17}$$

which state that read actions in isolation cannot communicate. For "$er_m(v); X$" we have the same equations (13) and (14), but these are constructed from the new operators now (they are parsed differently). For the process prefix combinator we further have the following equations:

$$(X + Y); Z \;=\; X; Z \,+\, Y; Z \tag{18}$$

$$(X \cdot Y); Z \;=\; X; (Y; Z) \tag{19}$$

$$(a \bullet_N \langle X_1, \ldots, X_N \rangle); Y \;=\; a \bullet_N \langle X_1; Y, \ldots, X_N; Y \rangle \tag{20}$$

For other cases, the ";" operator can be translated directly to the "$\cdot$", the normal ACP sequential composition.

An example term, which could not be expressed with only action prefix, is the following, where $P(\mathtt{v},\mathtt{w})$ is some term containing variables v, w.

$$(er_1(\mathtt{v}) \parallel er_2(\mathtt{w})); P(\mathtt{v},\mathtt{w})$$

With the equations from above, we can translate this, using [CM1], (16), (18), (19) to:

$$\begin{aligned}
&= (er_1(\mathtt{v}) \cdot er_2(\mathtt{w}) + er_2(\mathtt{w}) \cdot er_1(\mathtt{v}) + er_1(\mathtt{v}) \mid er_2(\mathtt{w})); P(\mathtt{v},\mathtt{w}) \\
&= (er_1(\mathtt{v}) \cdot er_2(\mathtt{w})); P(\mathtt{v},\mathtt{w}) + (er_2(\mathtt{w}) \cdot er_1(\mathtt{v})); P(\mathtt{v},\mathtt{w}) \\
&= er_1(\mathtt{v}); (er_2(\mathtt{w}); P(\mathtt{v},\mathtt{w})) + er_2(\mathtt{w}); (er_1(\mathtt{v}); P(\mathtt{v},\mathtt{w}))
\end{aligned}$$

From here on, we are back at notation from Section 2.4, and the terms behave as before. The new notation allows one to mix early and late inputs, and to merge them in parallel.

In [BB94], the action prefix setting is also being used to axiomatize *restricted input*, Hoare's input action, *prefix iteration*, *exits*, and CSP *synchronization merge*. Once the reader grasps the early and late input actions these extensions are straightforward, so there is no need for us to dwell on these issues in this document.

3 Reduced Models of FPA

Now that we have seen the full FPA_{ECA} (in Sections 2.1 to 2.4), we can study four interesting subalgebras, as indicated by the signature overview of Figure 3.

3.1 BVMA

The signature of *Basic Value Matching Algebra* (BVMA) gives us ACP with existing Read/Send communication, as discussed, e.g., by [BK86]. The direct axiomatization is simply obtained by taking the axioms from ACP, Val, Booleans, and RSC.

BVMA should *not* include variables nor substitutions (as suggested in [BB94]) if it is to be the algebra for Read/Send Communication from [BK86], as these will introduce non-standard elements in the Booleans.

3.2 VMC

The signature of *Value Matching Calculus* gives us an algebra which is very close to CCS under early bisimulation (i.e., CCS as discussed in [Mil89]). ACP's sequential composition is dropped, and replaced by an early read prefix operator.

We obtain a direct axiomatization by taking some of the theorems we could prove (in Section 2.4) as our new axioms. For example, we now adopt

$$w \notin FV(X) \Rightarrow er_m(v); X = er_m(w); (X[w/v])$$

which is exactly equation (15) from Section 2.4. To make this possible, we have to distinguish between bound and free variable occurrences, so we introduce a function

$$FV(_) : P \to \mathcal{P}(Var)$$

with straightforward axiomatization. Conditional equations then are used to express equalities like the α-conversion above.

In the full FPA_{ECA} specification, early read was axiomatized by "exploding" it to a summation for all possible inputs. In the setting of VMC, the same effect is achieved in smaller steps, by means of the *early input axiom* EIA:

$$er_m(v); X \; + \; er_m(v); Y \;\; = \;\; er_m(v); X \; + \; er_m(v); Y + er_m(v); \{X \; \triangleleft \; eq_D(v,i) \; \triangleright Y\}$$

This axiom states that if we have a choice between two read actions on v, we can add a third summand, in which we test for equality with a particular value i. Intuitively, it can be used to add a summand for any data value[4] i, until we have the full summation again.

Let us see how this works for our R, S example. The point is that we can add R_E to S_E and vice versa, and hence they are equal. For example, we can add $(\neg \text{eq}_D(\text{v},2) :\rightarrow Q)$ to R_E as follows:

$$\begin{aligned} R_E \;&=\; er_m(\text{v}); \{P \triangleleft \text{eq}_D(\text{v},2) \triangleright Q\} \;+\; er_m(\text{v}); \delta \\ &=\; R_E \;+\; er_m(\text{v}); \{\delta \triangleleft \text{eq}_D(\text{v},2) \triangleright \{P \triangleleft \text{eq}_D(\text{v},2) \triangleright Q\}\} \\ &=\; R_E \;+\; er_m(\text{v}); \{(\text{eq}_D(\text{v},2) :\rightarrow \delta) \;+\; (\neg\text{eq}_D(\text{v},2) :\rightarrow \{P \triangleleft \text{eq}_D(\text{v},2) \triangleright Q\})\} \\ &=\; R_E \;+\; (\neg\text{eq}_D(\text{v},2) :\rightarrow Q) \end{aligned}$$

were we have been using EIA (filling in 2 for i) and several equations over the Booleans such as (6) and (8).

3.3 VPC

The *Value Passing Calculus* (VPC) models CCS under late bisimulation. Its signature is that of VMC, but with the early read operation replaced by the late input. Its direct axiomatization is exactly the same as the one for VMC, with early reads replaced by late ones, and, most importantly, without the Early Input Axiom.

3.4 VPA

Recall from Section 2.4.2 that the normal forms of R_L and S_L did not contain variable occurrences of v. This observation is used in the last algebra we consider, *Value Passing Algebra*. Its signature only contains the functions needed to write the normal forms of R_L and S_L, i.e., we do not have variables, conditionals, nor the $lr_m; (X)$ operation: instead we only have a single read atom r_m which can be put as prefix before a sequence. The direct axiomatization merely indicates which communications are known to be δ, and which ones can be successful.

[4]The EIA as formulated in [BB94] uses a p instead of an i in the third summand, thus allowing tests involving arbitrary free variables. This is incorrect as (i) it does not follow from the equations over FPA_{ECA}, and (ii) it conflicts with the aim of equating a term with only bound variables to one without any variable. Also observe that the EIA characterizations of [MPW91], [PS93, Law **SP**] are based on the π-calculus, and therefore do not distinguish variables and values.

4 Assessment

The following propositions, which have not (yet) been formally proved, formulate four properties needed to asses the correctness of the various specifications discussed.

Definition 4.1 *Let M be any of* {FPA$_{\text{ECA}}$, BVMA, VPC, VMC, VPA}. *Define $\mathcal{R}_M$ as the term rewriting system (TRS) obtained by orienting all equations of the axiomatization of M from left to right. Rewriting of the $+$ takes place modulo ACI (Associativity, Commutativity, and Idempotency), $|$ modulo AC, and $\cdot$ modulo associativity.*

Proposition 4.2 *The following TRSs are strongly normalizing and confluent:*

1. $\mathcal{R}_{\text{FPA}_{\text{ECA}}}, \mathcal{R}_{\text{BVMA}}$, *and* $\mathcal{R}_{\text{VPA}}$.
2. $\mathcal{R}_{\text{VPC}}$ *and* $\mathcal{R}_{\text{VMC}}$ *modulo α-conversion of bound variables.*

Proposition 4.3 *Rewriting a term with only bound variables over* $\mathcal{R}_{\text{FPA}_{\text{ECA}}}, \mathcal{R}_{\text{BVMA}}$ *or* $\mathcal{R}_{\text{VPA}}$ *yields a normal form without variables occurring in it.*

Proposition 4.4 *Let M be one of the signatures in* {FPA$_{\text{ECA}}$, BVMA, VMC, VPC, VPA}, *let $I(M)$ be the initial model of the direct axiomatization of M, and let t_1, t_2 be terms over M. Then*

$$\langle I(\text{FPA}_{\text{ECA}})\rangle_M \models t_1 = t_2 \;\Leftrightarrow\; I(M) \models t_1 = t_2$$

Let $\sim_E$ and $\sim_L$ be the early and late bisimulations defined over labeled transitions derived from an operational semantics for VPC and VMC in the style of [HL95, Fig.11, Fig.10]. Then:

Proposition 4.5 *Equality over terms from* VMC *coincides with the early bisimulation $\sim_E$.*

Proposition 4.6 *Equality over terms from* VPC *coincides with the late bisimulation $\sim_L$.*

5 Concluding Remarks

In this paper, we studied *Functional Prefix Algebra*, as proposed by Baeten and Bergstra [BB94]. Their method is simple and elegant, but their article pays more attention to listing the signatures and axioms than to explaining the underlying intuitions. We remedied this deficiency by presenting an appropriate example, by illustrating the operational behavior of several operators, by emphasizing that variable elimination is the aim of these operators, and by listing the properties that can be used to judge the axiomatizations chosen.

An area for future work might include a generalization to recursively defined (infinite) processes, and to infinite data sorts.

An other challenging extension of the work presented here is in the area of *mobility*. The notions of early and late bisimilarity naturally occur in the π-calculus, and it would be intriguing to study whether a similar approach can be used to arrive at a relatively simple axiomatization of the π-calculus. A first step could be to allow for *internal* mobility only, as in the πI-calculus of [San95]. It might be wise to start with finite processes only, which in the π-calculus amounts to omitting the replication operator.

Acknowledgments

I would like to thank Jos Baeten, Jan Bergstra, Gerard Kok, Alban Ponse, and Lars Ake Fredlund for their helpful comments.

References

[BB92] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Method, Marktoberdorf 1990*, number F 88 in NATO ASI Series, pages 273–323. Springer-Verlag, 1992.

[BB94] J.C.M. Baeten and J.A. Bergstra. On sequential composition, action prefixes and process prefix. *Formal Aspects of Computing*, 6:250–268, 1994.

[BH93] J.A. Bergstra and J. Heering. Homomorphism preserving algebraic specifications require hidden sorts. Technical Report CS-R9344, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1993. To appear in Information & Computation.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[BK86] J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. In W. Bibel and K.P. Jantke, editors, *Math. Methods of Specification and Synthesis of Software Systems*, volume 215 of *LNCS*, pages 9–23. Springer-Verlag, 1986.

[BV95] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 4.* Oxford University Press, 1995.

[BW90] J.C.M. Baeten and W.P Weijland. *Process Algebra.* Cambridge University Press, 1990.

[Deu94] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications, Vol. I, Equations and Initial Semantics.* Springer-Verlag, 1985.

[HL95] M. Hennesy and H. Lin. Symbolic bisimulation. *Theoretical Computer Science*, 138:353–389, 1995.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Knu92] D.E. Knuth. *Literate Programming.* Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MPW91] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings CONCUR'91*, volume 527 of *LNCS*, pages 45–60. Springer-Verlag, 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, pages 1–77, 1992.

[PS93] J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *REX – A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *LNCS*. Springer-Verlag, 1993.

[San95] D. Sangiorgi. π-Calculus, internal mobility, and agent-passing calculi. Technical report, University of Edinburgh, 1995. To appear in TAPSOFT'95.

[Sio64] F.M. Sioson. Equational bases of boolean algebras. *The Journal of Symbolic Logic*, 29(3):115–124, 1964.

[Vis93] E. Visser. Combinatory logic & compilation of list matching. Master's thesis, University of Amsterdam, Programming Research Group, 1993.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 675–789. Elsevier Science Publishers, 1990.

A Errata to [BB94]

Before studying [BB94], you should make the following small corrections to it:

- p. 255, Table 4, 6th equation, and p. 266, Table 18, last but one equation: variable a should be a d (encapsulation works on core atoms).
- p. 256, Section 3.1, line above Table 5: Delete "C0, " (there is no such axiom).
- p. 257, Table 7, Booleans. Add the following equations (see [BB92] and our Section 2.1):

$$\beta :\rightarrow \gamma :\rightarrow X = (\beta \wedge \gamma) :\rightarrow X \tag{21}$$
$$\beta :\rightarrow X + \gamma :\rightarrow X = (\beta \vee \gamma) :\rightarrow X \tag{22}$$

- p. 259, Section 3.8, line 5: replace first | by || in list of operators that can be eliminated. Add λ to it, and replace the ";" by $er_m(v);_, lr_m(v);_, s_m(i);_, c_m(i);_$.
- p. 260, Section 4, line 4: Table 7 should be Table 8.
- p. 261, line 3: Table 9 should be Table 11.
- p. 261, Section 4.3, last + in definition of $s_m(v)$ should be a "=".
- p. 263, Section 5, line 3: through 3.5 should be "through 3.4" (section 3.5 only contains an example).
- p. 263, Section 5.1. Substitution and Variables should not be included. See Section 3.1 of this document.
- p. 263, Section 5.2, last line. Axiom A7 should not be included.
- p. 264, Table 15 (VMC). p in right-hand side of Early Input Axiom should be an i (see our Section 3.2).
- p. 264, line 1: Table 5 (Booleans) should be Table 6 (Booleans) except for equations

$$
\begin{aligned}
(\beta :\to X) \cdot Y &= \beta :\to X \cdot Y \\
\partial_{\{d\}}(\beta :\to X) &= \beta :\to \partial_{\{d\}}(X) \\
\rho_f(\beta :\to X) &= \beta :\to \rho_f(X)
\end{aligned}
$$

- p. 264 (VMC), line 2: "except for the sixth and the seventh" should be "except for the fourth and fifth" (the equations for $\cdot$ and $\bullet_N$).
- p. 264 (VMC), Sorts part of signature: include A, N. Constants part of signature: $\delta \in P$ should be "$\delta \in A$" (for consistency with p.252, Section 2.1).

  An alternative interpretation could be that VMC and VPC do not have atomic actions (as δ would be the only one). The sort N should be included in any case.
- p.265 (VMC), top: the signature should also include $_ :\to _ : B \times P \to P$
- p.266 (VPA), bottom: $r_m, s_m(i), c_m(i)$ should all be $\in A^c$, rather than $\in A$.

B ACP and FPA in ASF+SDF

In the remaining appendices the full algebraic specifications of FPA and the axiomatizations of the reduced modesls are given, written in the ASF+SDF specifcation formalism [BHK89, Deu94]. We have used the ASF+SDF Meta-environemnt [Kli93] for executing and documenting the specification. Using ASF+SDF has several advantages: (1) the system automatically checks the correct use of symbols introduced; (2) the user has to be explicit about operator priorities and module structure; (3) the specifier can obtain intuition about the functions by executing them using rewriting (4) the LaTeX generator can be used for full documentation.

An important disadvantage is that no commutative rewriting is supported. Our solution is to ignore this problem and to add commutativity axioms as "comment."

Most of the specifications given here will be easily readable for those familiar with algebraic specifications. Some details of SDF as occurring in the signatures might be unfamiliar; A tutorial explaining such details is given in [Deu94, Chapter 2].

The full specifcation sources are available from ftp://ftp.win.tue.nl/pub/techreports/arie/.

B.1 Basic Process Algebra

B.1.1 Atoms

At this stage, we merely state the existence of atomic actions (and the distinction between *core* and general atoms), we do not provide any particular atoms yet.

imports Layout$^{(F.1)}$
exports
  sorts CORE-ATOM ATOM
  context-free syntax
    CORE-ATOM → ATOM
  variables
    [*ab*][*0-9′*]* → ATOM
    [*de*][*0-9′*]* → CORE-ATOM

B.1.2 Choice

imports Atoms$^{(B.1.1)}$
exports
  sorts PROCESS
  context-free syntax
    ATOM → PROCESS
    PROCESS "+" PROCESS → PROCESS {**left**}
    "(" PROCESS ")" → PROCESS {**bracket**}
  variables
    [*XYZ*][*0-9′*]* → PROCESS
equations

$$X + (Y + Z) = X + Y + Z \qquad \text{[A2]}$$
$$X + X = X \qquad \text{[A3]}$$

Remark We omitted commutative equtaion [A1]: $X + Y = Y + X$, since we wish to use the rewriting system for testing purposes (ASF+SDF does not support rewriting modulo AC). Equation [A2] transforms "+" nodes to their left-associative form (for which, due to the **left** attribute, no brackets are necessary).

B.1.3 BPA

imports Choice$^{(B.1.2)}$
exports
  context-free syntax
    PROCESS "·" PROCESS → PROCESS {left}
  priorities
    "+" < "·"

equations

$$(X + Y) \cdot Z = X \cdot Z + Y \cdot Z \quad \text{[A4]}$$
$$X \cdot (Y \cdot Z) = X \cdot Y \cdot Z \quad \text{[A5]}$$

B.1.4 BPAdelta

imports BPA$^{(B.1.3)}$
exports
  context-free syntax
    δ → ATOM
equations

$$X + \delta = X \quad \text{[A6]}$$
$$\delta + X = X \quad \text{[A6']}$$
$$\delta \cdot X = \delta \quad \text{[A7]}$$

Remark In order to have delta's at the lhs disappear in spite of non-commutativity, we added equation [A6']

B.2 Process Algebra

In PA we extend BPA with concurrency, described using the left-merge operator.

B.2.1 LeftMerge

imports BPA$^{(B.1.3)}$
exports
  context-free syntax
    PROCESS "⌊⌊" PROCESS → PROCESS {left}
    PROCESS "||" PROCESS → PROCESS {left}
  priorities
    "+" < {left: "⌊⌊", "||"} < "·"

equations

$$\mathrm{a} \mathbin{⌊\!⌊} X = \mathrm{a} \cdot X \qquad \text{[CM2]}$$

$$\mathrm{a} \cdot X \mathbin{⌊\!⌊} Y = \mathrm{a} \cdot (X \parallel Y) \qquad \text{[CM3]}$$

$$(X + Y) \mathbin{⌊\!⌊} Z = X \mathbin{⌊\!⌊} Z + Y \mathbin{⌊\!⌊} Z \qquad \text{[CM4]}$$

Remark Equation [CM1] is given in Module ACP$^{(B.3.2)}$.

B.2.2 PA

In PA, we define the concurrency operator "merge" without the existence of communication. In module ACP, we will redefine this operator such that it deals with communication as well.
imports BPA$^{(\mathrm{B.1.3})}$ LeftMerge$^{(\mathrm{B.2.1})}$
equations

$$X \parallel Y = X \mathbin{⌊\!⌊} Y + Y \mathbin{⌊\!⌊} X \qquad \text{[M1]}$$

B.2.3 PAdelta

imports PA$^{(\mathrm{B.2.2})}$ BPAdelta$^{(\mathrm{B.1.4})}$

B.3 The Algebra of Communicating Processes

In ACP we add communcation to PA by means of the communication merge |.

B.3.1 Encapsulation

Define how sets of atoms (the sort A-SET, written as $\{a_1, \ldots, a_n\}$) occurring in a process can be changed into δ.
imports BPAdelta$^{(\mathrm{B.1.4})}$ Atoms$^{(\mathrm{B.1.1})}$
exports
  sorts A-SET
  context-free syntax
    "{" {CORE-ATOM ","}* "}" → A-SET
    ∂ "_" A-SET "(" PROCESS ")" → PROCESS
  variables
    *as[0-9]** → {CORE-ATOM ","}+
equations

Encapsulating Single atoms

$$\partial_{\{d\}}(\delta) = \delta \qquad \text{[D0]}$$

$$\partial_{\{d\}}(e) = e \qquad \textbf{when } d \neq e \qquad \text{[D1]}$$

$$\partial_{\{d\}}(d) = \delta \qquad \text{[D2]}$$

$$\partial_{\{d\}}(X+Y) = \partial_{\{d\}}(X) + \partial_{\{d\}}(Y) \qquad \text{[D3]}$$

$$\partial_{\{d\}}(X \cdot Y) = \partial_{\{d\}}(X) \cdot \partial_{\{d\}}(Y) \qquad \text{[D4]}$$

Encapsulating Sets

$$\partial_{\{\}}(X) = X \qquad \text{[D5]}$$

$$\partial_{\{as_1,\, as_2\}}(X) = \partial_{\{as_1\}}(\partial_{\{as_2\}}(X)) \qquad \text{[D6]}$$

B.3.2 ACP

Introduce the communication merge | and its interaction with the other operators.
imports BPAdelta(B.1.4) Encapsulation(B.3.1) LeftMerge(B.2.1)
exports
  context-free syntax
    PROCESS "|" PROCESS → PROCESS {**left**}
  priorities
    "+" < {**left:** "||", "⌊⌊", "|"} < "."

equations

Communication

$$a_1 \mid a_2 \mid a_3 = a_1 \mid (a_2 \mid a_3) \qquad \text{[C2]}$$

$$\delta \mid a = \delta \qquad \text{[C3]}$$

$$a \mid \delta = \delta \qquad \text{[C3']}$$

Concurrency

$$X \parallel Y = X ⌊⌊ Y + Y ⌊⌊ X + X \mid Y \qquad \text{[CM1]}$$

$$a \cdot X \mid b = (a \mid b) \cdot X \qquad \text{[CM5]}$$

$$a \mid b \cdot X = (a \mid b) \cdot X \qquad \text{[CM6]}$$

$$a \cdot X \mid b \cdot Y = (a \mid b) \cdot (X \parallel Y) \qquad \text{[CM7]}$$

$$(X+Y) \mid Z = X \mid Z + Y \mid Z \qquad \text{[CM8]}$$

$$X \mid (Y+Z) = X \mid Y + X \mid Z \qquad \text{[CM9]}$$

Remark Note that again, as for the "+" in BPA, we omitted the commutativity axiom for "|"; consequently, we added [C3'] to ensure removal of communication with δ.

C Functional Prefixes

C.1 Sequences

Sequences of process are used with the following intention: The full sequence is of length $\mathcal{N}$, which is also the exact number of data elements. A later "instantiation" operation can fill in one data element at each position.

imports ACP$^{(B.3.2)}$
exports
  sorts PROC-SEQ
  context-free syntax
    "(" {PROCESS ","}+ ")" → PROC-SEQ
  variables
    F [*0-9*]∗ → PROC-SEQ

C.2 SeqMap

In this module we define operators not occurring in [BB94]. They immitate the effect of the map, fold, and [1..N] functions as occurring in functional languages (clearly an extension of ASF+SDF with some form of second-order functions is desirable).

Note that the sorts MAP-FUN are left open here; new constants are introduced for each kind of map application.

imports Sequences$^{(C.1)}$ Numbers$^{(F.2)}$
exports
  sorts MAP-FUN FOLD-FUN
  context-free syntax

map(MAP-FUN, PROC-SEQ)	→ PROC-SEQ
"(" MAP-FUN ")" "@" PROCESS	→ PROCESS
succ(MAP-FUN)	→ MAP-FUN
deltaN(NAT)	→ PROC-SEQ
fold(FOLD-FUN, PROC-SEQ, PROCESS)	→ PROCESS
"(" FOLD-FUN ")" "@" PROCESS "," PROCESS	→ PROCESS
succ(FOLD-FUN)	→ FOLD-FUN
nth(PROC-SEQ, NAT)	→ PROCESS

hiddens
  variables

[*XYZ*]*s*[*0-9*]∗	→ {PROCESS ","}+
f	→ MAP-FUN
g	→ FOLD-FUN
[*XYZ*][*0-9*]∗"∗"	→ {PROCESS ","}∗

equations

Mapping over sequences

The succ on map functions can be used to change the function f at each iteration, using, e.g., one of the arguments of f. It can be used by specifying, e.g., succ(f(x,N,y)) = f(x,succ(N),y). If no equation is given, the default case applies, and f is not changed.

$$map(f, \langle X \rangle) = \langle (f) \mathbin{@} X \rangle \qquad [1]$$

$$\frac{(f) \mathbin{@} X = Y, \quad map(succ(f), \langle Xs \rangle) = \langle Ys \rangle}{map(f, \langle X, Xs \rangle) = \langle Y, Ys \rangle} \qquad [2]$$

$$succ(f) = f \quad \textbf{otherwise} \qquad [3]$$

Building up a sequence

$$deltaN(zero) = \langle \delta \rangle \qquad [4]$$

$$\frac{deltaN(n) = \langle Xs \rangle}{deltaN(succ(n)) = \langle \delta, Xs \rangle} \qquad [5]$$

Folding a sequence

$$fold(g, \langle X \rangle, Z) = (g) \mathbin{@} X, Z \qquad [6]$$

$$fold(g, \langle X, Xs \rangle, Z) = fold(succ(g), \langle Xs \rangle, (g) \mathbin{@} X, Z) \qquad [7]$$

$$succ(g) = g \quad \textbf{otherwise} \qquad [8]$$

Nth element

$$nth(\langle X, X^* \rangle, zero) = X \qquad [9]$$

$$nth(\langle X, X^* \rangle, succ(n)) = nth(\langle X^* \rangle, n) \qquad [10]$$

$$nth(F, n) = \delta \quad \textbf{otherwise} \qquad [11]$$

C.3 LatePrefix

The late prefix operator is just notation for an atom followed by a sequence, which distributes over composition, left-merge, and encapsulation.
imports Sequences(C.1) SeqMap(C.2)
exports
  context-free syntax
    ATOM $\bullet_N$ PROC-SEQ → PROCESS
hiddens
  context-free syntax
    "·" PROCESS → MAP-FUN
    "$\lfloor\!\lfloor$" PROCESS → MAP-FUN
    ∂ "_" A-SET → MAP-FUN
equations

$$\delta \bullet_N F = \delta \quad \text{[lat1]}$$
$$a \bullet_N F \cdot Y = a \bullet_N map(\cdot\, Y, F) \quad \text{[lat2]}$$
$$a \bullet_N F \lfloor\!\lfloor Y = a \bullet_N map(\lfloor\!\lfloor\, Y, F) \quad \text{[lat3]}$$

The following equation differs slightly from the one given in [BB94]: encapsulation is only defined on *core* atoms, so we replaced their "a" variable by a "d". Moreover, the result is a *process*, not an action, so we first have to be sure that in this case it also is an action (expressed in the condition). Note that in OBJ's order-sorted framework the automatic retract would allow for the equation as given in [BB94].

$$\frac{a = \partial_{\{d\}}(b)}{\partial_{\{d\}}(b \bullet_N F) = a \bullet_N map(\partial_{\{d\}}, F)} \quad \text{[lat4]}$$

$$(\cdot\, Y) @ X = X \cdot Y \quad \text{[map1]}$$
$$(\lfloor\!\lfloor\, Y) @ X = X \lfloor\!\lfloor Y \quad \text{[map2]}$$
$$(\partial_{\{d\}}) @ X = \partial_{\{d\}}(X) \quad \text{[map3]}$$

C.4 EarlyPrefix

Here we introduce an early prefix operator. In contrast to the late prefixing, the early one can be eliminated. It is best characterized by

$$a \circ \langle X_0, \cdots, X_n \rangle = a\hat{\ }0.X_0 + \cdots + a\hat{\ }n.X_n$$

Here the ˆ operator, left open in this module, can be used to make an action a do something with the ith data element.
imports Sequences(C.1) SeqMap(C.2) Data(C.5)
exports
  context-free syntax

ATOM ˆ DATA-FULL → ATOM
ATOM $\circ_N$ PROC-SEQ → PROCESS

hiddens

context-free syntax

plusconvert(NAT, ATOM) → FOLD-FUN

equations

$$a \circ_N F = fold(plusconvert(zero, a), F, \delta) \quad \text{[pre0]}$$

$$\delta \,\hat{}\, p = \delta \quad \text{[pre1]}$$

Needed for folding

$$(plusconvert(n, a)) \,@\, X, Y = Y + a \,\hat{}\, n \cdot X \quad \text{[fold1]}$$

$$succ(plusconvert(n, a)) = plusconvert(succ(n), a) \quad \text{[fold2]}$$

C.5 Data

The data values we will consider are just natural numbers. Moreover, we will allow an infinite number of *variables* over the data, which we represent by identifiers starting with a "$" sign.

imports Numbers(F.2)

exports

sorts DATA-FULL DATA-VAR DATA-VALUE

lexical syntax

*"$"[A-Z][A-Z0-9']** → DATA-VAR

context-free syntax

NAT → DATA-VALUE
DATA-VALUE → DATA-FULL
DATA-VAR → DATA-FULL
eq "_" "*D*" "(" DATA-FULL "," DATA-FULL ")" → BOOL

exports

variables

[*p*][*0-9*]* → DATA-FULL
[*vw*][*0-9*]* → DATA-VAR
[*ij*][*0-9*]* → DATA-VALUE

equations

$$eq_D(n_1, n_2) = eq_N(n_1, n_2) \quad \text{[dat1]}$$

$$eq_D(v, v) = T \quad \text{[dat2]}$$

Remark We can also specify (but not execute) that $eq_D(p,q) = eq_D(q,p)$ and $eq_D(p,q) = true, eq_D(q,r) = true \Rightarrow eq_D(p,r) = true$.

C.6 FPA

Here we combine late and early prefixing. A late prefix communicating with an atomic action has the same effect as an early prefix.

The combination of early and late prefixing is called the Functional Prefix Algebra. In subsequent modules we will see how an early read and a late read atomic action can be made to fit in the same FPA framework.

imports EarlyPrefix(C.4) LatePrefix(C.3)

equations

$$\mathrm{a} \bullet_N F \mid b = \mathrm{a} \circ_N F \mid b \quad \text{[lco1]}$$

$$\mathrm{a} \mid b \bullet_N F = \mathrm{a} \mid b \circ_N F \quad \text{[lco2]}$$

$$\mathrm{a} \bullet_N F \mid b \cdot X = \mathrm{a} \circ_N F \mid b \cdot X \quad \text{[lco3]}$$

$$\mathrm{a} \cdot X \mid b \bullet_N F = \mathrm{a} \cdot X \mid b \circ_N F \quad \text{[lco4]}$$

C.7 ECA

ECA The early communication axiom is states that late communication is impossible. When dealing with the π-calculus, we should drop this axiom.

imports LatePrefix(C.3)

equations

$$a \bullet_N F_1 \mid b \bullet_N F_2 = \delta \quad \text{[ECA]}$$

D Forms of Communication

D.1 Preliminaries

D.1.1 Ports

Ports are just names for communication channels. We don't need any other properties than an equality check. Here we choose to represent ports by *port(i)*, where i is the number of the port.

imports Booleans(F.3)

exports

sorts PORT DIGITS

lexical syntax

[0-9]+ → DIGITS

context-free syntax

DIGITS → PORT

eq "_" "*P*" "(" PORT "," PORT ")" → BOOL

variables

[*mk*][*0-9*]∗ → PORT

equations

$$eq_P\,(m, m) \;=\; T \tag{req1}$$

$$eq_P\,(m, k) \;=\; F \quad \textbf{otherwise} \tag{req2}$$

D.1.2 RSCprimitives

Here we add the ACP atomic actions for Read-Send Communication, as introduced in [BK86]. Note that only values can be transmitted.

imports ACP(B.3.2) Data(C.5) Ports(D.1.1)

exports

context-free syntax

r "_" PORT "(" DATA-VALUE ")" → CORE-ATOM
s "_" PORT "(" DATA-VALUE ")" → CORE-ATOM
c "_" PORT "(" DATA-VALUE ")" → CORE-ATOM

equations

$$r_m\,(i) \mid s_m\,(i) \;=\; c_m\,(i) \tag{rsc1}$$

$$\frac{\neg\; eq_D\,(i, j) \vee \neg\; eq_P\,(m, k) = T}{r_m\,(i) \mid s_k\,(j) = \delta} \tag{rsc2}$$

$$r_m\,(i) \mid r_k\,(j) \;=\; \delta \tag{rsc3}$$

$$s_m\,(i) \mid s_k\,(j) \;=\; \delta \tag{rsc4}$$

$$c_m\,(i) \mid \mathrm{a} \;=\; \delta \tag{rsc5}$$

D.1.3 RSC

Here we add the late to early conversions for the RSC primitives such that RSC fits in the FPA framework.

imports FPA(C.6) RSCprimitives(D.1.2) ECA(C.7)

equations

$$r_m\,(i)\,\hat{}\;p \;=\; \delta \tag{rsc6}$$

$$s_m\,(i)\,\hat{}\;p \;=\; \delta \tag{rsc7}$$

$$c_m\,(i)\,\hat{}\;p \;=\; \delta \tag{rsc8}$$

D.1.4 Conditional

The if-then-else on processes is used in full CCS of Milner [Mil89] as well. It is important as it can be used, in combination with the equality predicate over DATA-FULL, to test equality over data values, the only essential operation over data elements in our setting. Also notice that it can be used as the $[x = y]P$ "value-matching" operator from the π-calculus.
imports ACP$^{(\text{B.3.2})}$ Booleans$^{(\text{F.3})}$
exports
  context-free syntax
    BOOL ":→" PROCESS → PROCESS
    PROCESS "◁" BOOL "▷" PROCESS → PROCESS
  priorities
    "+" < {":→", "◁""▷"} < {**left:** "·", "|", "||", "⌊⌊"}

equations

If then

$$T :\rightarrow X = X \quad \text{[cb1]}$$
$$F :\rightarrow X = \delta \quad \text{[cb2]}$$

If then else

$$X \triangleleft \beta \triangleright Y = \beta :\rightarrow X + \neg\beta :\rightarrow Y \quad \text{[cb3]}$$

Distributing if-then

If the Boolean specification were sufficiently complete then the following equations would not be necessary (they follow from the equations above and those from module Booleans). However, with data variables, equality over data elements is not well-defined; an equality test over two different variables can only be answered once these variables are bound to an actual value.

$$\beta :\rightarrow \delta = \delta \quad \text{[bi0]}$$
$$\beta :\rightarrow (X + Y) = \beta :\rightarrow X + \beta :\rightarrow Y \quad \text{[bi1]}$$
$$(\beta :\rightarrow X) \cdot Y = \beta :\rightarrow X \cdot Y \quad \text{[bi2]}$$
$$(\beta :\rightarrow X) ⌊⌊ Y = \beta :\rightarrow X ⌊⌊ Y \quad \text{[bi3]}$$
$$(\beta :\rightarrow X) \mid Y = \beta :\rightarrow X \mid Y \quad \text{[bi4]}$$
$$X \mid (\beta :\rightarrow Y) = \beta :\rightarrow X \mid Y \quad \text{[bi5]}$$
$$\partial_{\{d\}}(\beta :\rightarrow X) = \beta :\rightarrow \partial_{\{d\}}(X) \quad \text{[bi6]}$$

The following equations come from [BB92, Table 4], and do not follow from other equations. (also note that equation [CG1] from [BB92, Table 4] follows from [GC9] and [GC2] there, and that for us it is not necessary to add equation [CO5] as it follows from [bi2] and [cb3]).

$$\beta_1 :\rightarrow \beta_2 :\rightarrow X = \beta_1 \wedge \beta_2 :\rightarrow X \quad \text{[bi8]}$$
$$\beta_1 :\rightarrow X + \beta_2 :\rightarrow X = \beta_1 \vee \beta_2 :\rightarrow X \quad \text{[bi9]}$$

Remark For || we first translate to +, and then use [bi1]. (note that $\beta :\rightarrow (X||Y) = (\beta :\rightarrow X)||Y$ does not hold).

D.1.5 Substitution

In this module we define straightforward subsitution. Variables occurring in processes, booleans, or data are just replaced. Note that variable occurrences in processes, booleans or data cannot be bound (here), so we do not need α-conversions here.

imports Data$^{(C.5)}$ Conditional$^{(D.1.4)}$ FPA$^{(C.6)}$ SeqMap$^{(C.2)}$

exports

context-free syntax

PROCESS "[" DATA-FULL "/" DATA-VAR "]" → PROCESS
BOOL "[" DATA-FULL "/" DATA-VAR "]" → BOOL
DATA-FULL "[" DATA-FULL "/" DATA-VAR "]" → DATA-FULL

priorities

"." < {PROCESS "["DATA-FULL "/"DATA-VAR "]" → PROCESS}, "¬" < {BOOL "["DATA-FULL "/"DATA-VAR "]" → BOOL}

hiddens

context-free syntax

"[" DATA-FULL "/" DATA-VAR "]" → MAP-FUN

equations

Processes

$$\delta[p\ /\ v] = \delta \quad \text{[su1]}$$

$$d[p\ /\ v] = d \quad \text{[su2]}$$

$$(X+Y)[p\ /\ v] = X[p\ /\ v] + Y[p\ /\ v] \quad \text{[su3]}$$

$$(X \cdot Y)[p\ /\ v] = X[p\ /\ v] \cdot Y[p\ /\ v] \quad \text{[su4]}$$

$$\frac{a_0[p\ /\ v] = a_1}{a_0 \bullet_N F[p\ /\ v] = a_1 \bullet_N map([p\ /\ v], F)} \quad \text{[su5]}$$

Booleans

Here we sligtly deviate from [BB94] since we do not define subsitution over ◁...▷, but over :→ instead, since it has a nicer rewriting behavior together with the elimination rule [cb4] of module Conditional.

$$(\beta :\rightarrow X)[p\ /\ v] = \beta[p\ /\ v] :\rightarrow X[p\ /\ v] \quad \text{[su6a]}$$

$$T[p\ /\ v] = T \quad \text{[su7]}$$

$$F[p\ /\ v] = F \quad \text{[su8]}$$

$$(\neg\ \beta)[p\ /\ v] = \neg\ \beta[p\ /\ v] \quad \text{[su9]}$$

$$(\beta_1 \vee \beta_2)[p\ /\ v] = \beta_1[p\ /\ v] \vee \beta_2[p\ /\ v] \quad \text{[su10]}$$

$$(\beta_1 \wedge \beta_2)[p\ /\ v] = \beta_1[p\ /\ v] \wedge \beta_2[p\ /\ v] \quad \text{[su11]}$$

$$eq_D\,(p_1, p_2)[p\ /\ v] \;=\; eq_D\,(p_1[p\ /\ v], p_2[p\ /\ v]) \qquad \text{[su12]}$$

Data

$$i[p\ /\ v] \;=\; i \qquad \text{[su13]}$$
$$v[p\ /\ v] \;=\; p \qquad \text{[su14]}$$
$$w[p\ /\ v] \;=\; w \quad \textbf{when} \quad v \neq w \qquad \text{[su15]}$$

Mapping Sequences

$$([p\ /\ v])\ @\ X \;=\; X[p\ /\ v] \qquad \text{[map1]}$$

D.1.6 Functional

We introduce an operator which takes a variable v and a process X, and creates a list with all possible instantiations of X.

imports Substitution[(D.1.5)]
exports
  context-free syntax
    "λ" DATA-VAR "$\cdot$" PROCESS $\rightarrow$ PROC-SEQ
hiddens
  context-free syntax
    subsnext(PROCESS, DATA-VAR, NAT) $\rightarrow$ MAP-FUN
equations

$$\lambda\ v \cdot X \;=\; map(subsnext(X,\ v,\ zero),\ deltaN(\mathcal{N})) \qquad \text{[ap1]}$$
$$(subsnext(X,\ v,\ n))\ @\ Y \;=\; X[n\ /\ v] \qquad \text{[ap1a]}$$
$$succ(subsnext(X,\ v,\ n)) \;=\; subsnext(X,\ v,\ succ(n)) \qquad \text{[ap1b]}$$

Here we assume that $\mathcal{N}$ is the maximum number of the data set. Note that the $\lambda v.X$ construct has the effect of removing all occurrences of v from X.

Some Examples Let us take some reduction examples to illustrate the rewriting effect of the various operators introduced: The term λ \$V $\cdot\ \delta$ rewrites to $\langle \delta, \delta, \delta \rangle$ (note that $\mathcal{N}$ is set to 2 in Numbers [(F.2)], so we only have three data elements):

An if-then-else

$$s_2\,(zero) \triangleleft eq_D\,(\$V,\ zero) \triangleright s_3\,(succ(zero))$$

is rewritten to the following:

$$eq_D\,(\$V,\ zero) :\rightarrow s_2\,(zero) + \neg\ eq_D\,(\$V,\ zero) :\rightarrow s_3\,(succ(zero))$$

Understanding that, it is easy to see that

$$\lambda\ \$V \cdot s_2\ (zero) \triangleleft eq_D\ (\$V,\ zero) \triangleright s_3\ (succ(zero))$$

results in

$$\langle s_2\ (zero),\ s_3\ (succ(zero)),\ s_3\ (succ(zero))\rangle$$

Notice that the three positions in the sequence correspond to the possible values of $V, and that given each value of $V the expression at that position has been computed.

D.1.7 InputAction

The single read operation will be the atomic action to be placed in front of a process sequence. It cannot communicate, but converted to an early version for i it reads in data value i.
imports RSC(D.1.3)
exports
context-free syntax
"r" "_" PORT → CORE-ATOM
equations

$$r_m\ \hat{}\ i = r_m\ (i) \quad \text{[ap6]}$$

$$r_m\ |\ a = \delta \quad \text{[ap7]}$$

D.2 Binding Actions

D.2.1 ActionPrefixing

Add a number of unary operators to represent early and late action prefixing. Note that we can only read variables, and that we can send only values.
imports Functional(D.1.6) InputAction(D.1.7)
exports
context-free syntax
er "_" PORT "(" DATA-VAR ")" ";" PROCESS → PROCESS
lr "_" PORT "(" DATA-VAR ")" ";" PROCESS → PROCESS
s "_" PORT "(" DATA-VALUE ")" ";" PROCESS → PROCESS
c "_" PORT "(" DATA-VALUE ")" ";" PROCESS → PROCESS
priorities

{er"_" "(" ")" ";", lr"_" "(" ")" ";", s"_" "(" ")" ";"} > {"◁" "▷"}

equations

The $[scr]_m$ atoms occuring in right-hand sides are the communication primitives from RSC. The first two equations state that s and c do not have a binding effect. (only values are transmitted.

$$s_m(i); X = s_m(i) \cdot X \quad \text{[ap1]}$$

$$c_m(i); X = c_m(i) \cdot X \quad \text{[ap2]}$$

The read operator is translated to the functional expansion from Functional (D.1.6)

$$er_m(v); X = r_m \circ_N \lambda v \cdot X \quad \text{[ap3]}$$

$$lr_m(v); X = r_m \bullet_N \lambda v \cdot X \quad \text{[ap5]}$$

Example The difference between early and late read only lies in the possibility to eliminate the prefixing operator before the sequence. Thus,

$$er_2(\$V);\ eq_D(\$V, succ(zero)) :\rightarrow s_3(succ(zero))$$

has the effect that every value is actually filled in:

$$r_2(zero) \cdot \delta + r_2(succ(zero)) \cdot s_3(succ(zero)) + r_2(succ(succ(zero))) \cdot \delta$$

Whereas the late counterpart contains a sequence with all possible values, but it cannot decide yet which one to pick (since the value of $V is not yet known):

$$lr_2(\$V);\ eq_D(\$V, succ(zero)) :\rightarrow s_3(succ(zero))$$

has the effect of:

$$r_2 \bullet_N \langle \delta, s_3(succ(zero)), \delta \rangle$$

The equation responsible for this difference between late and early read is equation [pre0] from Module FPA(C.6), which states that an early read prefix followed by a sequence is equal to the summation of all elements in the sequence.

D.2.2 ProcessPrefix

As a last case, we study how we can generalize the unary prefix read actions to a binary sequential composition operator ";". Its effect is a translation to the notation of module Functional (D.1.6).
imports Functional(D.1.6) InputAction(D.1.7)
exports
context-free syntax
PROCESS ";" PROCESS → PROCESS {left}
er "_" PORT "(" DATA-VAR ")" → CORE-ATOM
lr "_" PORT "(" DATA-VAR ")" → CORE-ATOM
priorities
"+" < {":→", "◁" "▷"} < {**left:** "|", "||", "⌊⌊"} < {**left:** "·", PROCESS ";" PROCESS → PROCESS}

hiddens
context-free syntax
";" PROCESS → MAP-FUN
equations

Sequential Composition The following four axioms also occur in module ActionPrefixing$^{(D.2.1)}$, but there they are parsed differently of course (as *prefix* operations)!

$$s_m(i); X = s_m(i) \cdot X \quad \text{[ap1]}$$

$$c_m(i); X = c_m(i) \cdot X \quad \text{[ap2]}$$

$$er_m(v); X = r_m \circ_N \lambda\, v \cdot X \quad \text{[ap3]}$$

$$lr_m(v); X = r_m \bullet_N \lambda\, v \cdot X \quad \text{[ap5]}$$

Distributing ;

$$\delta; X = \delta \quad \text{[se1]}$$

$$r_m(i); X = r_m(i) \cdot X \quad \text{[se2]}$$

$$r_m; X = r_m \cdot X \quad \text{[se3]}$$

$$(X + Y); Z = X; Z + Y; Z \quad \text{[se4]}$$

$$X \cdot Y; Z = X; (Y; Z) \quad \text{[se5]}$$

$$\mathrm{a} \bullet_N F; Y = \mathrm{a} \bullet_N map(; Y, F) \quad \text{[se6]}$$

$$(; Y) @ X = X; Y \quad \text{[se7]}$$

New atom cases

$$er_m(v) \mid \mathrm{a} = \delta \quad \text{[el1]}$$

$$lr_m(v) \mid \mathrm{a} = \delta \quad \text{[el2]}$$

$$er_m(v) \,\hat{}\, p = \delta \quad \text{[el3]}$$

$$lr_m(v) \,\hat{}\, p = \delta \quad \text{[el4]}$$

D.2.3 SendVariable

The plain send only works with values; following [BB94, Example 4.3] we can also send variables in the following way. Note that sending a variable does not have a *binding* effect, so we do not need to use the sequence operator here.
imports RSC$^{(D.1.3)}$ Conditional$^{(D.1.4)}$ SeqMap$^{(C.2)}$

exports
  context-free syntax
    "*s*" "_" PORT "(" DATA-VAR ")" → PROCESS
hiddens
  context-free syntax
    plus-if-eq(NAT, DATA-VAR, PORT) → FOLD-FUN
equations

$$s_m\ (v) = fold(plus\text{-}if\text{-}eq(zero,\ v,\ m),\ deltaN(\mathcal{N}),\ \delta) \quad \text{[se1]}$$

$$(plus\text{-}if\text{-}eq(n,\ v,\ m))\ @\ X,\ Y = eq_D\ (v, n) :\rightarrow s_m\ (n) + Y \quad \text{[se2]}$$

$$succ(plus\text{-}if\text{-}eq(n,\ v,\ m)) = plus\text{-}if\text{-}eq(succ(n),\ v,\ m) \quad \text{[se3]}$$

D.2.4 Example

Sending a variable using s_1 (\$V) results in an if-then series for each possible value:

$$eq_D\ (\$V,\ succ(succ(zero))) :\rightarrow s_1\ (succ(succ(zero)))$$
$$+\ eq_D\ (\$V,\ succ(zero)) :\rightarrow s_1\ (succ(zero))$$
$$+\ eq_D\ (\$V,\ zero) :\rightarrow s_1\ (zero)$$

Thus, testing equality is the single operation allowed on variables.

E Reduced Model Specifications

E.1 Value Matching Calculus

Now we will see how we can obtain a theory which is very similar to finitary CCS under early bisimulation. The relevant signature of VMC is given below:

E.1.1 VMCsyntax

Syntax for the Value Matching Calculus. The auxiliary operators for prefixing from FPA (e.g., the $\circ_N, \bullet_N$), the sequences, and the $\lambda V.$ operator are deleted). In this module VMCsyntax, we only copied the existing relevant equations from the ActionPrefixing modules. In the next module VMC, we will give the new equations.
imports Choice(B.1.2) Booleans(F.3) Data(C.5) Ports(D.1.1)
exports
  context-free syntax
    δ → ATOM

    er "_" PORT "(" DATA-VAR ")" ";" PROCESS → PROCESS
    s "_" PORT "(" DATA-VALUE ")" ";" PROCESS → PROCESS
    c "_" PORT "(" DATA-VALUE ")" ";" PROCESS → PROCESS

```
PROCESS "||" PROCESS                          → PROCESS    {left}
PROCESS "⌊⌊" PROCESS                          → PROCESS
PROCESS "|" PROCESS                           → PROCESS
PROCESS "\" "_" "δ" PORT                      → PROCESS

BOOL ":→" PROCESS                             → PROCESS
PROCESS "◁" BOOL "▷" PROCESS                  → PROCESS

PROCESS "[" DATA-FULL "/" DATA-VAR "]"        → PROCESS
BOOL "[" DATA-FULL "/" DATA-VAR "]"           → BOOL
DATA-FULL "[" DATA-FULL "/" DATA-VAR "]"      → DATA-FULL
```

priorities

```
"+" < {":→", "◁" "▷"} < {left: "||", "⌊⌊", "|", "\" "_" "δ"} < {er "_" "(" ")" ";", s "_" "(" ")" ";",
c "_" "(" ")" ";"} < {PROCESS "[" DATA-FULL "/" DATA-VAR "]" → PROCESS}
```

priorities

```
{"∨", "∧"} < {BOOL "[" DATA-FULL "/" DATA-VAR "]" → BOOL} < "¬"
```

equations

ACP equations Equations A1, A2, A3 are given in module Choice. Axiom A7 deals with $\delta \cdot X$, which should not be included.

$$X + \delta = X \qquad \text{[A6]}$$

$$\delta + X = X \qquad \text{[A6']}$$

$$X \,\|\, Y = X \mathbin{|\!\lfloor} Y + Y \mathbin{|\!\lfloor} X + X \mid Y \qquad \text{[CM1]}$$

$$(X + Y) \mathbin{|\!\lfloor} Z = X \mathbin{|\!\lfloor} Z + Y \mathbin{|\!\lfloor} Z \qquad \text{[CM4]}$$

$$(X + Y) \mid Z = X \mid Z + Y \mid Z \qquad \text{[CM8]}$$

$$X \mid (Y + Z) = X \mid Y + X \mid Z \qquad \text{[CM9]}$$

Booleans See module Booleans for the plain equations. Equality over Data elements is defined in module Data.

$$T :\to X = X \qquad \text{[cb1]}$$

$$F :\to X = \delta \qquad \text{[cb2]}$$

$$X \triangleleft \beta \triangleright Y = \beta :\to X + \neg\beta :\to Y \qquad \text{[cb3]}$$

$$\beta :\to \delta = \delta \qquad \text{[bi0]}$$

$$\beta :\to (X + Y) = \beta :\to X + \beta :\to Y \qquad \text{[bi1]}$$

$$(\beta :\to X) \mathbin{|\!\lfloor} Y = \beta :\to X \mathbin{|\!\lfloor} Y \qquad \text{[bi3]}$$

$$(\beta :\to X) \mid Y = \beta :\to X \mid Y \qquad \text{[bi4]}$$

$$X \mid (\beta :\to Y) = \beta :\to X \mid Y \qquad \text{[bi5]}$$

Substitution

$$\delta[p / v] = \delta \quad \text{[su1]}$$
$$d[p / v] = d \quad \text{[su2]}$$
$$(X + Y)[p / v] = X[p / v] + Y[p / v] \quad \text{[su3]}$$
$$(\beta :\rightarrow X)[p / v] = \beta[p / v] :\rightarrow X[p / v] \quad \text{[su6a]}$$
$$T[p / v] = T \quad \text{[su7]}$$
$$F[p / v] = F \quad \text{[su8]}$$
$$\neg\, \beta[p / v] = \neg\, \beta[p / v] \quad \text{[su9]}$$
$$(\beta_1 \vee \beta_2)[p / v] = \beta_1[p / v] \vee \beta_2[p / v] \quad \text{[su10]}$$
$$(\beta_1 \wedge \beta_2)[p / v] = \beta_1[p / v] \wedge \beta_2[p / v] \quad \text{[su11]}$$
$$eq_D\,(p_1, p_2)[p / v] = eq_D\,(p_1[p / v], p_2[p / v]) \quad \text{[su12]}$$
$$i[p / v] = i \quad \text{[su13]}$$
$$v[p / v] = p \quad \text{[su14]}$$
$$w[p / v] = w \qquad \textbf{when } v \neq w \quad \text{[su15]}$$

E.1.2 FreeVars

In VMC, we drop the process sequence operator. The price to pay is that we have to distinguish between free and bound occurrences of variables

imports VMCsyntax(E.1.1)

exports

sorts VAR-SET

context-free syntax

"{" {DATA-VAR ","}* "}" → VAR-SET
VAR-SET "∪" VAR-SET → VAR-SET {**left**}
DATA-VAR "∈" VAR-SET → BOOL
VAR-SET "−" DATA-VAR → VAR-SET

"FV"(PROCESS) → VAR-SET
"FV"(BOOL) → VAR-SET
"FV"(DATA-FULL) → VAR-SET

hiddens

variables

[vw][0-9]"*"* → {DATA-VAR ","}*
Vars → VAR-SET

equations

Set Operations

$$\{v^*\} \cup \{w^*\} = \{v^*, w^*\} \quad \text{[dv0]}$$
$$v \in \{v_1^*, v, v_2^*\} = T \quad \text{[dv1]}$$

$$v \in \{w^*\} = F \quad \textbf{otherwise} \quad \text{[dv2]}$$
$$\{v_1^*, v, v_2^*\} - v = \{v_1^*, v_2^*\} - v \quad \text{[dv3]}$$
$$\{v^*\} - v = \{\} \quad \textbf{otherwise} \quad \text{[dv4]}$$

Processes

$$\begin{aligned}
\mathrm{FV}(\delta) &= \{\} && \text{[fv1]}\\
\mathrm{FV}(er_m(v); X) &= \mathrm{FV}(X) - v && \text{[fv2]}\\
\mathrm{FV}(s_m(i); X) &= \mathrm{FV}(X) && \text{[fv3]}\\
\mathrm{FV}(c_m(i); X) &= \mathrm{FV}(X) && \text{[fv4]}\\
\mathrm{FV}(X + Y) &= \mathrm{FV}(X) \cup \mathrm{FV}(Y) && \text{[fv5]}\\
\mathrm{FV}(\beta :\rightarrow X) &= \mathrm{FV}(\beta) \cup \mathrm{FV}(X) && \text{[fv6]}
\end{aligned}$$

Booleans

$$\begin{aligned}
\mathrm{FV}(T) &= \{\} && \text{[fv7]}\\
\mathrm{FV}(F) &= \{\} && \text{[fv8]}\\
\mathrm{FV}(\neg \beta) &= \mathrm{FV}(\beta) && \text{[fv9]}\\
\mathrm{FV}(\beta_1 \wedge \beta_2) &= \mathrm{FV}(\beta_1) \cup \mathrm{FV}(\beta_2) && \text{[fv10]}\\
\mathrm{FV}(\beta_1 \vee \beta_2) &= \mathrm{FV}(\beta_1) \cup \mathrm{FV}(\beta_2) && \text{[fv11]}\\
\mathrm{FV}(eq_D(p_1, p_2)) &= \mathrm{FV}(p_1) \cup \mathrm{FV}(p_2) && \text{[fv12]}
\end{aligned}$$

Data

$$\begin{aligned}
\mathrm{FV}(i) &= \{\} && \text{[fv13]}\\
\mathrm{FV}(v) &= \{v\} && \text{[fv14]}
\end{aligned}$$

E.1.3 VMC

The equations needed in for VMC given here are thos not explicitly given (though derivable from) those of ActionPrefixing (D.2.1).
imports FreeVars(E.1.2) Booleans(F.3)
equations

Early Input Axiom The EIA equation given in [BB94] concisely expresses the exact nature of early input,

$$\mathrm{er}_m(v); X + \mathrm{er}(v); Y = \mathrm{er}_m(v); X + \mathrm{er}_m(v); Y + \mathrm{er}_m(v); (X \lhd v = i \rhd Y)$$

It is, however, not usable in a rewriting system.

Alpha Conversion The equation expressing renaming of the parameter is not usable in a term rewriting system either.

$$er_m(v); X = er_m(w); X[w/v]$$

The remaining equations define the operations introduced in VMCsyntax (E.1.1).

Left Merge

$$\delta \,|\!\lfloor\, X = \delta \qquad \text{[vmc3]}$$

$$er_m(v); X \,|\!\lfloor\, Y = er_m(v); (X \,\|\, Y) \quad \textbf{when } v \in \mathrm{FV}(Y) = F \qquad \text{[vmc4]}$$

$$s_m(i); X \,|\!\lfloor\, Y = s_m(i); (X \,\|\, Y) \qquad \text{[vmc5]}$$

$$c_m(i); X \,|\!\lfloor\, Y = c_m(i); (X \,\|\, Y) \qquad \text{[vmc6]}$$

Succesful Communication

$$er_m(v); X \mid s_m(i); Y = c_m(i); (X[i\,/\,v] \,\|\, Y) \qquad \text{[vmc7a]}$$

$$s_m(i); X \mid er_m(v); Y = c_m(i); (X \,\|\, Y[i\,/\,v]) \qquad \text{[vmc7b]}$$

$$er_m(v); X \mid s_k(i); Y = \delta \quad \textbf{when } m \neq k \qquad \text{[vmc8a]}$$

$$s_m(i); X \mid er_k(v); Y = \delta \quad \textbf{when } m \neq k \qquad \text{[vmc8b]}$$

Impossible Communications

$$er_m(v); X \mid er_k(w); Y = \delta \qquad \text{[vmc9]}$$

$$s_m(i); X \mid s_k(j); Y = \delta \qquad \text{[vmc10]}$$

$$c_m(i); X \mid Y = \delta \qquad \text{[vmc11a]}$$

$$X \mid c_m(i); Y = \delta \qquad \text{[vmc11b]}$$

$$\delta \mid X = \delta \qquad \text{[vmc12a]}$$

$$X \mid \delta = \delta \qquad \text{[vmc12b]}$$

Hiding

$$\delta \setminus_\delta m = \delta \qquad \text{[vmc13]}$$

$$er_m(v); X \setminus_\delta m = \delta \qquad \text{[vmc14]}$$

$$er_m(v); X \setminus_\delta k = er_m(v); (X \setminus_\delta k) \quad \textbf{when } m \neq k \qquad \text{[vmc15]}$$

$$s_m(i); X \setminus_\delta m = \delta \qquad \text{[vmc16]}$$

$$s_m(i); X \setminus_\delta k = s_m(i); (X \setminus_\delta k) \quad \textbf{when } m \neq k \qquad \text{[vmc17]}$$

$$c_m(i); X \setminus_\delta k = c_m(i); (X \setminus_\delta k) \qquad \text{[vmc18]}$$

$$(X + Y) \setminus_\delta m = X \setminus_\delta m + Y \setminus_\delta m \qquad \text{[vmc19]}$$

Extra Substitution Axioms

$$\frac{v \neq w, \ w \in \mathrm{FV}(p) = F}{(er_m(w); X)[p / v] = er_m(w); X[p / v]} \qquad \text{[su1]}$$

$$(s_m(i); X)[p / v] = s_m(i); X[p / v] \qquad \text{[su2]}$$

$$(c_m(i); X)[p / v] = s_m(i); X[p / v] \qquad \text{[su3]}$$

E.2 Value Passing Calculus

In the Value Passing Calculus we replace the $er_m(v)$ operation by the late version, $lr_m(v)$. The axiomatization is the same as for the Value Matching Calculus, where the *only* difference is that the Early Input Axiom EIA is dropped. In [BB94, Table 17] no new axioms are introduced; only the ones involving the $er_m(v)$ operation are replaced by their $lr_m(v)$ counterparts.

E.3 Value Passing Algebra

E.3.1 VPA

Value Passing Algebra corresponds to VPC, but without the explicit manipulation of bound and free variables. Many of the equations given below just indicate which communications are δ. In ASF+SDF this could also have been expressed more consily using a default/otherwise equation

imports ACP(B.3.2) LatePrefix(C.3) ECA(C.7) Numbers(F.2) Ports(D.1.1)
exports
  context-free syntax
    r "_" PORT → CORE-ATOM
    s "_" PORT "(" NAT ")" → CORE-ATOM
    c "_" PORT "(" NAT ")" → CORE-ATOM
  variables
    [*ij*][*0-9*]* → NAT
equations

Atomic Failing Communication

$$r_m \mid a = \delta \qquad \text{[vp1]}$$

$$s_m(i) \mid a = \delta \qquad \text{[vp2]}$$

$$c_m(i) \mid a = \delta \qquad \text{[vp3]}$$

Sequence Communication

$$r_m \bullet_N F \mid s_m(i) = c_m(i) \cdot nth(F, i) \quad \text{[vp4a]}$$

$$r_m \bullet_N F \mid s_k(i) = \delta \quad \textbf{when } k \neq m \quad \text{[vp5a]}$$

$$s_m(i) \mid r_m \bullet_N F = c_m(i) \cdot nth(F, i) \quad \text{[vp4b]}$$

$$s_m(i) \mid r_k \bullet_N F = \delta \quad \textbf{when } k \neq m \quad \text{[vp5b]}$$

$$r_m \bullet_N F \mid s_m(i) \cdot Y = c_m(i) \cdot (nth(F, i) \parallel Y) \quad \text{[vp6a]}$$

$$r_m \bullet_N F \mid s_k(i) \cdot X = \delta \quad \textbf{when } k \neq m \quad \text{[vp7a]}$$

$$s_m(i) \cdot Y \mid r_m \bullet_N F = c_m(i) \cdot (Y \parallel nth(F, i)) \quad \text{[vp6b]}$$

$$s_m(i) \cdot X \mid r_k \bullet_N F = \delta \quad \textbf{when } k \neq m \quad \text{[vp7b]}$$

Impossible Sequence Communication

$$r_m \bullet_N F \mid r_k = \delta \quad \text{[vp8a]}$$

$$r_m \mid r_k \bullet_N F = \delta \quad \text{[vp8b]}$$

$$r_m \bullet_N F \mid r_k \cdot X = \delta \quad \text{[vp9a]}$$

$$r_m \cdot X \mid r_k \bullet_N F = \delta \quad \text{[vp9b]}$$

$$r_m \bullet_N F \mid c_k(i) = \delta \quad \text{[vp10a]}$$

$$c_m(i) \mid r_k \bullet_N F = \delta \quad \text{[vp10b]}$$

$$r_m \bullet_N F \mid c_k(i) \cdot X = \delta \quad \text{[vp11a]}$$

$$c_m(i) \cdot X \mid r_k \bullet_N F = \delta \quad \text{[vp11b]}$$

$$\delta \mid X = \delta \quad \text{[vp12a]}$$

$$X \mid \delta = \delta \quad \text{[vp12b]}$$

$$s_m(i) \bullet_N F = \delta \quad \text{[vp13]}$$

$$c_m(i) \bullet_N F = \delta \quad \text{[vp14]}$$

E.3.2 The Example Revisited

Looking at the signature of this specification, one sees that the if-then constructs are eliminated, as well as all variables and a read operation involving bound variables. With the signatures given here, we will have to express our R and S as follows:

$$
\begin{aligned}
S &= er_m \bullet \langle \delta, \delta, P, \delta, \cdots, \delta \rangle + er_m \bullet \langle Q, Q, \delta, Q, \cdots, Q \rangle \\
R &= er_m \bullet \langle Q, Q, P, Q, \cdots, Q \rangle + er_m \bullet \langle \delta, \cdots, \delta \rangle
\end{aligned}
$$

Comparing these expressions with the late versions of S and R from Section ??, immediately explains the way the VPA specification works: The normal forms of the ProcessPrefix specification are the only terms we are dealing with. The equations list some equalities when ACP operators are combined with the $\bullet_N$ operator, and indicates which communications are impossible (resulting in δ).

F Library Modules

F.1 Layout

exports
  lexical syntax
    $[\sqcup \backslash t \backslash n]$ → LAYOUT
    "%%" ~[\n]* "\n" → LAYOUT

F.2 Numbers

Define natural numbers using the *zero* and *succ* operations. The constant $\mathcal{N}$ plays the role of a maximally interesting number, and is here for testing purposes made equal to 2.
imports Booleans(F.3)
exports
  sorts NAT
  context-free syntax
    zero → NAT
    succ(NAT) → NAT
    eq "_" "N" "(" NAT "," NAT ")" → BOOL
    "$\mathcal{N}$" → NAT
  variables
    [n][0-9']* → NAT
equations

$$eq_N(n, n) = T \qquad \text{[nat1]}$$

$$eq_N(n_1, n_2) = F \quad \textbf{otherwise} \qquad \text{[nat2]}$$

$$\mathcal{N} = succ(succ(zero)) \qquad \text{[nat3]}$$

F.3 Booleans

imports Layout[(F.1)]
exports
  sorts BOOL
  context-free syntax

T	$\rightarrow$ BOOL	
F	$\rightarrow$ BOOL	
BOOL "$\wedge$" BOOL	$\rightarrow$ BOOL	**{left}**
BOOL "$\vee$" BOOL	$\rightarrow$ BOOL	**{left}**
"(" BOOL ")"	$\rightarrow$ BOOL	**{bracket}**
"$\neg$" BOOL	$\rightarrow$ BOOL	

  priorities
    "$\vee$" < "$\wedge$" < "$\neg$"
  variables
    "β" [0-9]* $\rightarrow$ BOOL
equations

$$T \wedge \beta = \beta \quad \text{[bb1]}$$
$$F \wedge \beta = F \quad \text{[bb2]}$$
$$T \vee \beta = T \quad \text{[bb3]}$$
$$F \vee \beta = \beta \quad \text{[bb4]}$$
$$\neg T = F \quad \text{[bb5]}$$
$$\neg F = T \quad \text{[bb6]}$$

Remark As long as true and false act as constructors of the Booleans, the following equations follow from the ones given above. However, we will later on deal with Booleans involving an equality predicate over variables, which does not reduce to one of these constructors as long as the value for the variable is not filled in. In that case, we do want the following equations to hold:

$$\beta \wedge \beta = \beta \quad \text{[bb7]}$$
$$\beta \wedge \neg \beta = F \quad \text{[bb8]}$$
$$\beta \vee \beta = \beta \quad \text{[bb9]}$$
$$\beta \vee \neg \beta = T \quad \text{[bb10]}$$

$$\neg \neg \beta = \beta \quad \text{[bb11]}$$
$$\beta_0 \wedge (\beta_1 \vee \beta_2) = \beta_0 \wedge \beta_1 \vee \beta_0 \wedge \beta_2 \quad \text{[bb12]}$$
$$\beta_0 \vee \beta_1 \wedge \beta_2 = (\beta_0 \vee \beta_1) \wedge (\beta_0 \vee \beta_2) \quad \text{[bb13]}$$
$$\neg (\beta_1 \wedge \beta_2) = \neg \beta_1 \vee \neg \beta_2 \quad \text{[bb14]}$$
$$\neg (\beta_1 \vee \beta_2) = \neg \beta_1 \wedge \neg \beta_2 \quad \text{[bb15]}$$

Remark We should also add commutativity axioms for $\wedge$ and $\vee$.

Contents

1 Introduction 1
- 1.1 Motivating Example . . . 2
- 1.2 FPA_{ECA} and its Subalgebras . . . 4
- 1.3 Reduced Model Specifications . . . 4
- 1.4 Understanding by Experiment . . . 5

2 Functional Prefix Algebra 6
- 2.1 Booleans, Data Values and Data Variables . . . 6
- 2.2 Read/Send Communication . . . 7
- 2.3 Sequences and Prefixing . . . 8
- 2.4 Early and Late Read . . . 9
  - 2.4.1 Axiomatization . . . 9
  - 2.4.2 The if-then-else Example Revisited . . . 10
  - 2.4.3 Bound Variables and α-Conversion? . . . 10
- 2.5 Process Prefix and Further Extensions . . . 11

3 Reduced Models of FPA 12
- 3.1 BVMA . . . 12
- 3.2 VMC . . . 12
- 3.3 VPC . . . 13
- 3.4 VPA . . . 13

4 Assessment 14

5 Concluding Remarks 14

A Errata to [BB94] 16

B ACP and FPA in ASF+SDF 17
- B.1 Basic Process Algebra . . . 18
  - B.1.1 Atoms . . . 18
  - B.1.2 Choice . . . 18
  - B.1.3 BPA . . . 19
  - B.1.4 BPAdelta . . . 19
- B.2 Process Algebra . . . 19
  - B.2.1 LeftMerge . . . 19
  - B.2.2 PA . . . 20
  - B.2.3 PAdelta . . . 20
- B.3 The Algebra of Communicating Processes . . . 20
  - B.3.1 Encapsulation . . . 20
  - B.3.2 ACP . . . 21

C Functional Prefixes **22**
C.1 Sequences 22
C.2 SeqMap 22
C.3 LatePrefix 24
C.4 EarlyPrefix 24
C.5 Data 25
C.6 FPA 26
C.7 ECA 26

D Forms of Communication **26**
D.1 Preliminaries 26
D.1.1 Ports 26
D.1.2 RSCprimitives 27
D.1.3 RSC 27
D.1.4 Conditional 28
D.1.5 Substitution 29
D.1.6 Functional 30
D.1.7 InputAction 31
D.2 Binding Actions 31
D.2.1 ActionPrefixing 31
D.2.2 ProcessPrefix 32
D.2.3 SendVariable 33
D.2.4 Example 34

E Reduced Model Specifications **34**
E.1 Value Matching Calculus 34
E.1.1 VMCsyntax 34
E.1.2 FreeVars 36
E.1.3 VMC 37
E.2 Value Passing Calculus 39
E.3 Value Passing Algebra 39
E.3.1 VPA 39
E.3.2 The Example Revisited 41

F Library Modules **41**
F.1 Layout 41
F.2 Numbers 41
F.3 Booleans 42

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness, p. 24
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Komatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma, J. Davies R. Gerth, S. Graf W. Janssen, B. Jonsson S. Katz, G. Lowe M. Poel, A. Pnueli C. Rump, J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef	Concrete process algebra, p. 134.
95/04	J. Hidders	An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra	Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber	Math/pad: A System for On-Line Prepararation of Mathematical Documents, p. 15

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.