Causality and True Concurrency: A Dataflow Analysis of the Pi-Calculus

Lalita Jategaonkar Jagadeesan
Radha Jagadeesan

July 1995

In *Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology* ,Volume 936 of the Lecture Notes in Computer Science, July 1995.

Causality and True Concurrency: A Data-flow Analysis of the Pi-Calculus (Extended Abstract)

(Appeared in the Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, July 1995 Lecture Notes in Computer Science, Volume 936)

Lalita Jategaonkar Jagadeesan
Software Production Research Dept.
AT&T Bell Laboratories
Naperville, IL 60566 (USA)
lalita@research.att.com

Radha Jagadeesan*
Math. Sciences
Loyola University
Chicago, IL 60626 (USA)
radha@math.luc.edu

1 Introduction

The PI–CALCULUS [18, 17] is a process algebra for describing networks of processes with dynamically evolving communication structure. The key idea underlying the PI–CALCULUS is the notion of *naming*: names are used to refer to channels — the links between processes, and can be dynamically created or hidden. Names together with a rich algebra of process combinators that includes parallel composition, allow the PI–CALCULUS to encode asynchronous networks of processes that evolve dynamically. In turn, mobility — this ability to change the network configuration during execution — gives the calculus enormous expressive power; for example, it supports a compositional translation of the lambda calculus [17], supports a compositional translation of the higher order PI–CALCULUS into the first order fragment [23], and has allowed it to be exploited as the target language of semantics of object oriented programming [27].

Most of the reasoning principles proposed for the PI–CALCULUS have been based on *interleaving semantics*, in which only temporal behavior of processes is observed. In particular, all interleaving-based theories satisfy the equation

$$a \parallel b = ab + ba$$

However, a significant difference between these processes is that in the process $a \parallel b$, the actions a and b are *causally independent* and may occur simultaneously. In contrast, the process $ab + ba$ forces either the a to occur first, causing b, or vice-versa. Thus, as is well-known, interleaving-based semantics cannot capture the notion of causality.

* This research was supported by grants from NSF and ONR.

Our approach and results

In this paper, we use Kahn semantics of static determinate data-flow as our starting point for the examination of causality in the PI–CALCULUS. The choice of this starting point is influenced by the elegant encoding of causality in Kahn semantics: an event a is a necessary cause for event b if the minimal input required to produce b includes a. However, there are two complications that arise when one attempts to use such ideas in analyzing the PI–CALCULUS.

- Kahn semantics applies to static networks, whereas the PI–CALCULUS can encode dynamic networks. The solution to this problem is motivated by tools used in the study of computation in Linear Logic [8], in particular the Geometry of Interaction program [9, 1]. For the purposes of this paper, the key relevant idea is to mimic dynamic networks (of the lambda calculus) by flow of *structured* tokens in a static network.
- The second complication that arises is the non-determinism in PI–CALCULUS processes. The approach we adopt here is to use a variant of the generalization of the determinate data-flow semantics to a semantics for indeterminate dataflow networks [22, 2].

By changing the structure of the tokens, the resulting semantics applies uniformly to the Calculus of Communicating Systems [19] and the PI–CALCULUS. This change of structure of tokens is suggested by the nature of computation in the PI–CALCULUS. The definitions of the process combinators essentially stand unchanged. In particular, the definition of parallel composition is unchanged. Thus our treatment of interaction is "generic", *an essential criterion for any good description of parallel composition.*

Furthermore, for all of these calculi, our semantics distinguishes the processes $a \parallel b$ and $ab + ba$. More generally, for CCS, we show that our semantics induces the same process equivalence as a pomset-based semantics.

Related work

Many "true concurrency" semantics [3, 5, 25, 11, 13, 21, 26, 28], which capture causality to varying degrees, have been proposed for other models of concurrency, including process algebras such as CSP and CCS, Petri Nets, and event structures. These semantics have typically generalized interleaving semantics to encode some degree of concurrency, such as "steps" of concurrent actions rather than single actions, and "pomsets" of partially-ordered multisets of actions rather than linear temporal sequences. However, the investigation of true concurrency semantics for the PI–CALCULUS has not achieved this level of maturity. As far as the authors are aware, the semantics developed in this paper is one of the first true concurrency semantics to generalize smoothly from CCS to the PI–CALCULUS.

The only exceptions of which we are aware is the work of Boreale and Sangiorgi on location bisimulation [7, 16] in the PI–CALCULUS [24] and causality bisimulation [5] in the PI–CALCULUS [4]. Since location bisimulation is incomparable with causal equivalences [4, 5], the former does not address the problem that we are aiming to solve.

On the other hand, [4] is quite relevant to our aims. The primary contribution of [4] is to show that their definition of causal bisimulation can be reduced to observational equivalence in the PI–CALCULUS. As a special case this shows that causal bisimulation on CCS can be reduced to observation equivalence on the monadic PI–CALCULUS. This *beautiful* theorem justifies the observing of some forms of "true concurrency" in CCS.

However, Boreale and Sangiorgi's extension of causal bisimulation to the PI–CALCULUS does not have a uniform notion of causality. Hence, as they point out, it distinguishes the processes

$$P = \nu b(\bar{a} < b > .0 | \bar{b} < y > .0), \quad Q = \nu b(\bar{a} < b > .\bar{b} < y > .0)$$

for purely technical reasons. Indeed, the problem of describing a homogeneous causality-based semantics, that would perforce equate these processes, is left as an open problem there. We note that our semantics does equate these processes.

Organization

We start off with a description of the model for CCS and its full abstraction properties. This section brings out many of the key intuitions underlying our interpretation in a simpler setting. We follow with a description of the model for the PI–CALCULUS.

Notation and background

Domain Theory We assume some familiarity with domain theory and stable domain theory. A Scott domain is a bounded complete algebraic poset with a countable basis. We will use $\sqsubseteq$ for the ordering, $\sqsubset$ for the strict ordering, $\sqcup$ for least upper bounds, and $\sqcap$ for greatest lower bounds. We will use $B(D)$ to stand for the basis elements of a Scott domain D. Two elements of a Scott domain are consistent if they have an upper bound. A dI-domain is a Scott domain that is distributive, in which there are no infinite descending chains of compact elements. Given a set of labels $\mathcal{A} = \{ai\}$, we will use $\Pi_{\mathcal{A}} D$ for the A–indexed product of copies of the domain D. We will use $\mathbf{x}, \mathbf{y}$ etc for elements of $\Pi_{\mathcal{A}} D$. The ai'th projection will be denoted by π_{ai}. We will sometimes write x_{aj} for $\pi_{aj}(\mathbf{x})$.

A function between Scott domains is continuous if it preserves limits of omega chains. A function is *stable* if it is continuous and in addition preserves meets of consistent elements. If f is stable, and $y \sqsubseteq f(x)$, the least element $z \sqsubseteq x$ such that $y \sqsubseteq f(z)$ exists and is denoted $\mathtt{Min}(f, x, y)$. We will consider stable functions between dI-domains ordered by the extensional ordering, $f \sqsubseteq g$ if $(\forall x)\ [f(x) \sqsubseteq g(x)]$.

We will also use the relational powerdomain construction on Scott domains. The relational powerdomain of a Scott domain D will be denoted as $P_R(D)$.

$B(P_R(D))$ is the non-empty finite subsets of $B(D)$, ordered by $(\forall d_i)\ (\exists e_j) d_i \sqsubseteq d_j \Rightarrow \{d_1, \ldots, d_m\} \sqsubseteq \{e_1, \ldots, e_n\}$. $B(D)$ is a lattice, and hence $P_R(D)$ is a complete algebraic lattice. The top element of $P_R(D)$ is denoted $\top$. For any domain D, $P_R(D)$ supports a continuous operation $\uplus : P_R(D) \times P_R(D) \rightarrow P_R(D)$ that is idempotent, commutative and associative and satisfies $x \sqsubseteq x \uplus y$. There is a natural embedding of D into $P_R(D)$, denoted $\{\cdot\}$; and any continuous function f from a Scott domain E into $P_R(D)$ extends uniquely to a $\uplus$ preserving function $f^\dagger$ from $P_R(E)$ to $P_R(D)$.

We will also use the following constructions on dI–domains. Let D be a dI-domain. Then, the domain $\mathtt{Seq}(D)$ ("sequences on D") is the dI-domain with basis elements $\{\langle d_1, \ldots, d_n\rangle \mid d_i \in B(D) - \{\perp\}\}$, with ordering $\langle d_1, \ldots, d_m\rangle \sqsubseteq \langle e_1, \ldots, e_n\rangle$ if $m \leq n\ \wedge\ (\forall 1 \leq i \leq m)\ [d_i \sqsubseteq e_i]$. Let D be a dI-domain. Then, the domain $D_\perp$ ("lift of D") is the domain with basis elements $\{d_\perp\rangle \mid d \in B(D)\} \cup \{\perp\}$, with ordering $\perp \sqsubseteq d_\perp, d \sqsubseteq e \Rightarrow d_\perp \sqsubseteq e_\perp$. We will use the following functions on $\mathtt{Seq}(D_\perp)$. We define them as monotone functions on $B(\mathtt{Seq}(D_\perp))$.

$$\mathtt{cons}(d, x) = \begin{cases} \langle (d)_\perp \rangle, \text{ if } x = \perp \\ \langle (d)_\perp, x_1, \ldots, x_n \rangle, \text{ if } x = \langle x_1, \ldots, x_n \rangle \end{cases}$$

$$\mathtt{tail}(\langle d_1 \ldots d_n \rangle) = \left\{ \begin{array}{l} \perp, \text{ if } n \leq 1 \\ \langle d_2, \ldots d_n \rangle \text{ otherwise} \end{array} \right\} \quad \mathtt{head}(x) = \left\{ \begin{array}{l} (d),\ x = \mathtt{cons}((d), \_) \\ \perp, \text{ otherwise} \end{array} \right\}$$

Pomsets A *pomset* is a labeled partial order. Formally, a pomset, p, consists of a set Events_p whose elements are called *events*, a set Labels_p whose elements are called *labels*, a function $\mathrm{label}_p : \mathrm{Events}_p \rightarrow \mathrm{Labels}_p$, and a partial order relation $\leq_p$ on Events_p.

We say that event e *causes* event e' in a pomset p iff $e <_p e'$. We write $\mathrm{initial}(p)$ to denote the set of events in p that are minimal with respect to $<_p$, *i.e.*, events that do not have any causes in p. A *cut* of p is any subset C of Events_p such that no two distinct events in C are causally related by $<_p$.

A pomset p is a *prefix* of a pomset q iff p is a restriction of q to a downward-closed subset of Events_q. A pomset p is an *augmentation* of a pomset q simply adds more ordering to q; namely, $\mathrm{Events}_p = \mathrm{Events}_q$, $\mathrm{Labels}_p = \mathrm{Labels}_q$, $\mathrm{label}_p = \mathrm{label}_q$, and for all events $e, e' \in \mathrm{Events}_q$, $e <_q e'$ implies $e <_p e'$.

2 CCS

The Calculus of Communicating Systems [19] is a calculus for describing static networks of processes that synchronize via channels. Let $\mathcal{A}$ be a set of actions, let $\bar{\mathcal{A}} = \{\bar{a} : a \in \mathcal{A}\}$ such that $\mathcal{A}$ and $\bar{\mathcal{A}}$ are disjoint, and let $\Sigma = \mathcal{A} \cup \bar{\mathcal{A}}$. For all actions $a \in \mathcal{A}$, $\bar{\bar{a}} = a$. We will use the names ai to refer to elements of Σ. The syntax of CCS processes is as follows, where a ranges over $\mathcal{A}$, α ranges over Σ, and X ranges over some set of process variables V:

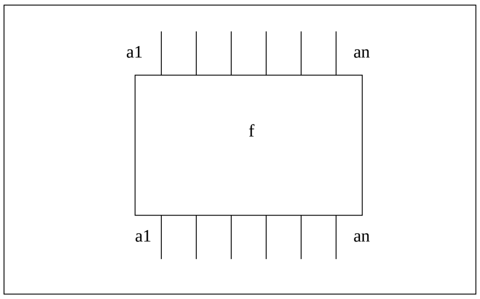
$$p ::= \mathrm{NIL} \mid \alpha.p \mid \nu a.p \mid p + q \mid (p|p) \mid X \mid recX.p$$

where . is the prefixing operator, ν is the restriction operator, $+$ is the non-deterministic choice operator, $|$ is the parallel composition operator, and rec is the recursion operator.

2.1 The semantics

The domain D_{CCS} N^v is the dI-domain of vertical natural numbers. Formally, $|N^v| = \{\infty\} \cup N$ with the ordering relation $n \leq m \Rightarrow n \sqsubseteq m, (\forall n)\ [n \sqsubseteq \infty]$. Note that $N^v \simeq \mathtt{Seq}((\cdot)_\perp)$ where $(\cdot)$ stands for the one point domain. In the dataflow metaphor that we will be using to motivate the definitions, the elements of N^v correspond to the number of tokens in a channel; the `tail` function defined earlier returns the "cdr" of the sequence of tokens in a channel, and the `cons` function defined earlier adds a token to the sequence of tokens in a channel. The domain D_{CCS} is defined as the indexed product $\Pi_{\mathcal{A}} N^v$, *i.e* a product with a copy of N^v for each action name. D_{CCS} is a dI-domain, that is also a lattice.

Basic intuitions We will exploit the intuitions gleaned from the analysis of computation in Linear Logic, in particular, as in the geometry of interaction program. A process will be a set of operators on D_{CCS}— thus, indeterminacy gets modeled by describing a process as a set of functions. The functions in a process can be represented as in the picture. Informally, the outputs on a channel



"a" can be thought of as "willingness" of the process to synchronize on 'a". Inputs on a channel "a" are thought of as the "willingness" of the environment to synchronize on "a".

Pomset operators We first identify the class of functions on D_{CCS} that are of interest. Our definition is motivated by two key properties of pomsets. Firstly, every event has a unique cause; stability is the domain theoretic representation of this property. Secondly, there is an underlying partial order and every event is reachable finitely from the initial events. The following definition identifies a reachability property of stable functions. In the context of the dataflow metaphor, we can read the following strict causality condition intuitively as follows. Consider the minimum input required to cause the output of a finite number n of tokens on channel ai. Then the number of tokens on channel ai in this input is strictly lesser than n. The full abstraction theorem brings out the precise content of this connection between pomset traces and the pomset operators.

Definition 1 *A* pomset operator f *on* D_{CCS} *is a stable function from* D_{CCS} *to* D_{CCS} *that satisfies* strict causality*:*

$$(\forall \mathbf{x} \in D_{\mathsf{CCS}})\ (\forall x^f_{ai} \in B(N^v))\ [x^f_{ai} \sqsubseteq \pi_{ai} f(\mathbf{x}) \Rightarrow\ \pi_{ai}(\mathtt{Min}(\pi_{ai} \circ f, \mathbf{x}, x^f_{ai})) \sqsubset x^f_{ai}]$$

P_{CCS} is the set of pomset operators on D_{CCS}, ordered extensionally. The denotation of a process will be an element of $P_R(P_{\mathsf{CCS}})$; *i.e* the denotation of a process is a set of pomset operators on D_{CCS}.

Nil $[\![\mathtt{nil}]\!]\mathbf{v} = \bot$. The $\mathtt{nil}$ process produces no information.

Prefixing We define a prefixing operation $\left[\!\left[ai.[\cdot] \right]\!\right]$ on pomset operators as follows. The prefixing operation on processes is defined as $\left[\!\left[ai.[\cdot] \right]\!\right]^\dagger$; *i.e.* by lifting the operation to sets of functions pointwise.

Pictorially, prefixing on a pomset operator is done as follows: in this picture, the box is used to indicate synchronization: the function inside the box is only activated if the indicated channel has non-bottom information.

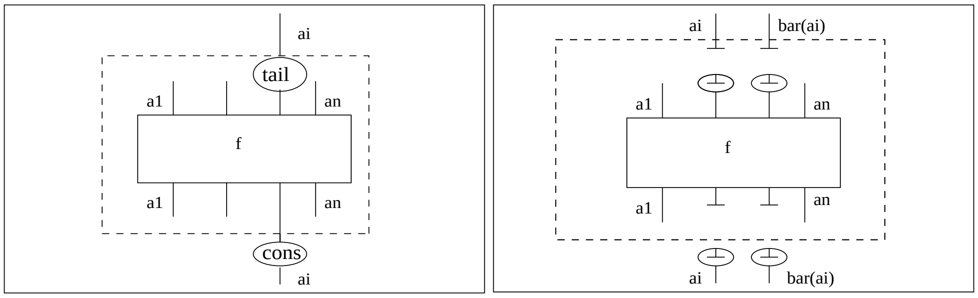


Fig. 1. Prefixing (left) and Restriction (right) in CCS

Formally, we proceed as follows. Define the functions $\mathtt{tail}_i, \mathtt{cons}_i : D_{\mathsf{CCS}} \to D_{\mathsf{CCS}}$ as follows. $\mathtt{tail}_i(\mathbf{x}) = \mathbf{y}$ if $j \neq i$ implies $\pi_{aj}(\mathbf{x}) = \pi_{aj}(\mathbf{y})$ and $j = i$ implies that $\pi_{ai}(\mathbf{y}) = \mathtt{tail}(\pi_{ai}(\mathbf{x}))$. Similarly, $\mathtt{cons}_i(\mathbf{x}) = \mathbf{y}$ if $j \neq i$ implies $\pi_{aj}(\mathbf{x}) = \pi_{aj}(\mathbf{y})$ and $j = i$ implies that $\pi_{ai}(\mathbf{y}) = \mathtt{cons}(\bot, \pi_{ai}(\mathbf{x}))$.

$$[\![\mathtt{ai.f}]\!]\mathbf{v} = \begin{cases} \mathtt{cons}_i(\bot), \text{ if } \pi_{ai}(\mathbf{v}) = \bot \\ \mathtt{cons}_i(f(\mathtt{tail}_i(\mathbf{v}))), \text{ otherwise} \end{cases}$$

We draw the attention of the reader to the fact that the above definition is merely a textual representation of the transformations indicated in the figure.

Restriction We define a restriction operation $[\![(\nu ai)\,[\cdot]]\!]$ on pomset operators as follows. The restriction operation on sets of pomset operators is defined as $[\![(\nu ai)\,[\cdot]]\!]^{\dagger}$; *i.e.* by lifting the operation to sets of functions pointwise.

Pictorially, restriction on a pomset operator f is done in figure 1. Ignore all input on channel $ai, \bar{ai}$ and produce no output on channel $ai, \bar{ai}$; otherwise behave like f.

Formally, we proceed as follows. Define a function $\mathtt{rest}_{ai} : D_{\mathsf{CCS}} \to D_{\mathsf{CCS}}$ as follows: $\mathtt{rest}_{ai}(\mathbf{x}) = \mathbf{y}$ if $\pi_{ai}(\mathbf{y}) = \pi_{\bar{ai}}(\mathbf{y}) = \perp$ and $\pi_{aj}(\mathbf{x}) = \pi_{aj}(\mathbf{y})$ otherwise. Define

$$[\![(\nu ai)[f]]\!]\mathbf{v} = \mathtt{rest}_{ai}(f(\mathtt{rest}_{ai}(\mathbf{v})))$$

Plus Plus is interpreted by the union operation of $P_R(P_{\mathsf{CCS}})$. This is the primary reason for the correspondence of the semantics with a variant of pomset traces (as opposed to a richer notion of pomset failures, for example).

$$[\![P_1 + P_2]\!] = [\![P_1]\!] \uplus [\![P_2]\!]$$

Parallel Composition We define a parallel composition operation $[\![[\cdot]|[\cdot]]\!]$ on pomset operators as follows; this composition will yield a set of pomset operators, *i.e.* an element of $P_R(P_{\mathsf{CCS}})$. The parallel composition operation on sets of pomset operators is defined as $[\![[\cdot]|[\cdot]]\!]^{\dagger}$; *i.e.* by lifting the operation to sets of functions pointwise.

We first expose the intuitions behind the definitions. Pictorially, parallel composition on pomset operators f, g works as follows.

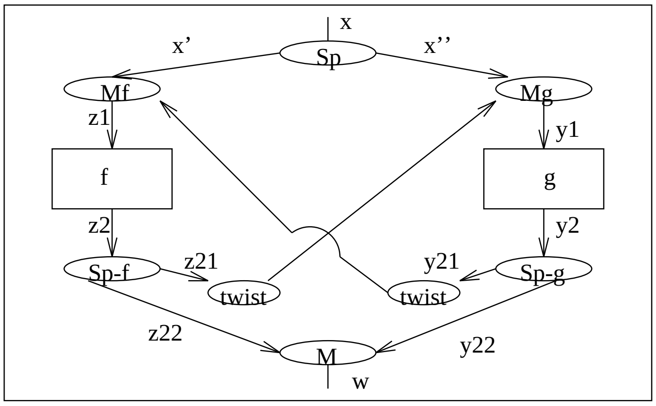


Fig. 2. Parallel composition in CCS

Note that the feedback loop in the above picture meshes with the intuitions that were stated earlier about inputs and outputs. The "willingness" of f to synchronize on channel a gets transmitted to g as the "willingness" of the environment to synchronize on $\bar{a}$, and vice versa. Inputs from the environment are split (Sp in the above figure) amongst f, g. Outputs created by f, g are split (Spf and Spg in the figure) between outputs to the environment and outputs to the dual channel name in the other process. The outputs on any channel name are obtained by merging the outputs to the environment from the individual processes(M in the above figure). The inputs to f, g are created by merging inputs from the environment and outputs from the dual channel name in the other process (Mf and Mg in the above figure). The above construction is repeated for all possible split and merge processes. The formal development follows.

Let $\mathtt{Sp}, \mathtt{Spf}, \mathtt{Spg}$ be determinate splits and $\mathtt{M}, \mathtt{Mf}, \mathtt{Mg}$ be determinate merges such that $(\mathtt{Sp}, \mathtt{M})$, $(\mathtt{Spf}, \mathtt{Mf})$ and $(\mathtt{Spg}, \mathtt{Mg})$ are pairs of functions that are inverses to each other. Let the function $\mathtt{twist}$ that interchanges the data on dual names be defined as follows: $\mathtt{twist}(\mathbf{x}) = \mathbf{y}$, if $\pi_{ai}(\mathbf{x}) = \pi_{\bar{a}i}(\mathbf{y})$.

The parallel composition of f, g for a given choice of determinate split and merge processes is described as as the least solution of the following equations; these equations are solved using the fixpoint iteration in the style of the semantics for determinate dataflow networks [14, 15]. More precisely, the following equations describe a pomset operator $h_{\mathtt{M},\mathtt{Mf},\mathtt{Mg}}$, where $h_{\mathtt{M},\mathtt{Mf},\mathtt{Mg}}(\mathbf{x}) = \mathbf{w}$, if $\mathbf{w}$ arises in the least solution of the equations:

$$\begin{array}{llll}
\mathtt{Sp}(\mathbf{x}) & = \langle \mathbf{x}', \mathbf{x}'' \rangle & \mathtt{Mf}\langle \mathbf{x}', \mathtt{twist}(\mathbf{y}_{21}) \rangle & = \mathbf{z}_1 \\
\mathtt{Mg}\langle \mathbf{x}'', \mathtt{twist}(\mathbf{z}_{21}) \rangle & = \mathbf{y}_1 & f(\mathbf{z}_1) & = \mathbf{z}_2 \\
g(\mathbf{y}_1) & = \mathbf{y}_2 & \mathtt{Spf}(\mathbf{z}_2) & = \langle \mathbf{z}_{21}, \mathbf{z}_{22} \rangle \\
\mathtt{Spg}(\mathbf{y}_2) & = \langle \mathbf{y}_{21}, \mathbf{y}_{22} \rangle & \mathtt{M}\langle \mathbf{z}_{22}, \mathbf{y}_{22} \rangle & = \mathbf{w}
\end{array}$$

Note that each merge (and hence the corresponding split) can be encoded as a (possibly infinite) sequence over 0, 1. Let $\mathtt{Z}$ be the flat domain over the set $\{0, 1\}$. Thus, $\mathtt{Seq}(Z)$ is the domain of finite and infinite binary sequences under the prefix ordering. In this light, the fixpoint equation above describes a continuous function H from $\mathtt{Seq}(\mathtt{Z}) \times \mathtt{Seq}(\mathtt{Z}) \times \mathtt{Seq}(\mathtt{Z})$ (one copy to encode each merge) to pomset operators. Thus, $H^{\dagger}$ is a continuous function preserving $\uplus$ from $P_R(Seq(\mathtt{Z}) \times Seq(\mathtt{Z}) \times Seq(\mathtt{Z}))$ to $P_{\mathtt{CCS}}$.

$$[\![f|g]\!] = H^{\dagger}(\top)$$

Recursion All the process combinators mentioned earlier are monotone and continuous in their process arguments. Recursion is interpreted via the usual fixpoint principles (valid since $P_R(P_{\mathtt{CCS}})$ is a cpo).

Examples Let $\mathtt{onea}$ denote the element of $D_{\mathtt{CCS}}$ that satisfies $\pi_i(\mathtt{onea}) = \perp$ for $i \neq a$ and $\pi_i(\mathbf{y}) = 1$ for $i = a$. Symmetrically define $\mathtt{oneb}$. Then, the denotation of the process $a||b$ has the following maximum element f: f is the constant function returning $\mathtt{onea} \sqcup \mathtt{oneb}$. The denotation of the process $a.b$ has the following maximum element g: $g(\mathbf{x}) = \mathtt{onea}$ when $\pi_a(\mathbf{x}) = \perp$, and $g(\mathbf{x}) = \mathtt{onea} \sqcup \mathtt{oneb}$ when

$\pi_a(\mathbf{x}) \neq \perp$. The denotation of the process $b.a$ has a maximum element h with the roles of a, b reversed in the definition of g. Thus, we have:

$$[\![a||b]\!] \neq [\![ab + ba]\!]$$

2.2 Results

We introduce the following variation on pomsets. Let p be a pomset, let a be a label in p, let C be a cut of p, and let C_a be the restriction of C to events labeled a. Then C has *temporally-ordered autoconcurrency over a* iff there is some total ordering $e_1 \ldots e_n$ of all the events in C_a such that for all $1 \leq i < j \leq n$; for all events $e \in Events_p$, $e <_p e_i$ implies that $e <_p e_j$, and for all events $e' \in Events_p$, $e_j <_p e'$ implies $e_i <_p e'$. Hence, if e_j is enabled, then e_i must be enabled as well; conversely, if e_j is a cause of some event e, then e_i is a cause of event e as well.

Definition 2 *A pomset p is a* TP *(for temporally ordered autoconcurrency) iff for all labels $a \in Labels_p$ and all cuts C of p, C has temporally-ordered autoconcurrency over a.*

We first give a pomset-trace based semantics to CCS by translating CCS terms to 1-safe Petri Nets as in [10, 12], and taking the augmentation-closure of the pomset-traces of these Petri Nets as in [13]. The details are standard and are omitted here due to space restrictions. We then restrict these sets of pomset-traces to TPs, to get the semantics $[\![(\cdot)]\!]_{\mathtt{POM}}$.

Correspondence theorems The space of TPs corresponds precisely to the space of pomset operators. This result in essentially "predicted" in the work on (stable) event structures [20].

Let f be a pomset operator; then, f induces the following (possibly infinite) TP. The nodes of the pomset are $\{ai^n \mid (\exists \mathbf{x})\ [n \sqsubseteq \pi_{ai}(f\mathbf{x})]$. The label on a node ai^n is ai. The ordering relation is induced by the canonical trace of the stable function f, $\mathtt{Tr}(f)$ that associates with each finite output the minimal input required to cause the output; aj^m is a cause for ai^n if $(\exists (\mathbf{y}, \mathtt{cons}_i(n)) \in \mathtt{Tr}(f))\ [m \sqsubseteq \pi_{aj}(\mathbf{y})]$. Stability ensures the unique causal prefix property required for pomsets. Strict causality ensures that we have a partial order and that all nodes of the induced pomset are reachable from the initial nodes. By construction, the pomsets are TPs.

Let C be a TP. Then, C induces the following pomset operator. $(\mathbf{x}, \langle \perp, \ldots, \perp, n_{ai}, \perp, \ldots, \rangle)$ is in the trace of the pomset operator if $\mathbf{x}$ is the minimum causal prefix in the pomset for n occurrences of ai: the TP condition ensures the existence of such a minimum causal prefix. Stability is guaranteed by the unique causal prefix properties of pomsets. The partial order underlying the pomset and the reachability of the pomset ensures the strict causality property.

This close correspondence between the space of pomset operators and the space of TPs is the key tool in proving full abstraction.

Theorem 1 *(FULL ABSTRACTION)* $[\![P]\!] = [\![Q]\!] \Leftrightarrow [\![P]\!]_{\mathtt{POM}} = [\![Q]\!]_{\mathtt{POM}}$.

3 PI–CALCULUS

This section describes the extension of the earlier treatment to the pi-calculus. Most of the theory developed for CCS goes through unchanged. The only changes will be the replacement of the base domain N^v by a domain T.

The pi-calculus [18, 17] extends CCS by allowing the passage of names along channels. Let $\mathcal{A}$ be a set of actions, let $\bar{\mathcal{A}} = \{\bar{a} : a \in \mathcal{A}\}$ such that $\mathcal{A}$ and $\bar{\mathcal{A}}$ are disjoint, and let $\Sigma = \mathcal{A} \cup \bar{\mathcal{A}}$. For all actions $a \in \mathcal{A}$, $\bar{\bar{a}} = a$. The syntax of PI–CALCULUS processes is as follows, where a, x range over $\mathcal{A}$:

$$p ::= \mathrm{NIL} |\ a(x).p |\ \bar{a}(x).p |\ \nu(x).p |\ p + q |\ (p|p) |\ !p$$

where $a(x).(.)$ is the input prefixing operator, and $\bar{a}(x).(.)$ is the output prefixing operator. The input prefixing operator binds the occurrence of x. ν is the restriction operator, $+$ is the non-deterministic choice operator, $|$ is the parallel composition operator, and $!p$ is the copying operator.

3.1 The semantics

The domain $D_{\mathtt{PiC}}$ The domain T (for "tree–like") is given by the solution of the recursive domain equation

$$T \simeq \mathtt{Seq}((T \times T)_\perp)$$

This solution is obtained in the category of dI–domains and rigid embeddings [6]. The domain $D_{\mathtt{PiC}}$ is defined as the indexed product $\Pi_{\mathcal{A}} T$. Thus, T is the domain associated with each channel name, just as N^v was the domain associated with each channel name in CCS. The structure of T is suggested by the information carried along names in the PI–CALCULUS. Intuitively, in the course of the evolution of a process, a name carries many (possibly different) names. The Seq construction above models this. Also, note that when a name is sent along another name, the co-name is also carried along implicitly. This gets modeled by the product in the above construction.

Pomset operators are defined exactly as before, *i.e.* strictly causal and stable operators on $D_{\mathtt{PiC}}$. $P_{\mathtt{PiC}}$ is the set of pomset operators on $D_{\mathtt{PiC}}$, ordered extensionally. Following the earlier recipe, processes are interpreted as elements of $P_R(P_{\mathtt{PiC}})$.

The definitions of restriction, nil and parallel composition are exactly as before. (apart from the changes in the domain and range of definition, of course!!). Furthermore, plus is again interpreted by the $\uplus$ operation on $P_R(P_{\mathtt{PiC}})$. As before, all combinators yield monotone and continuous functions on their process arguments, yielding an immediate interpretation of full recursion (hence including the pi-calculus operation $!P$). We treat prefixing below.

Input prefixing We define an input prefixing operation $\llbracket ai(x).[\cdot] \rrbracket$ on pomset operators as follows. The input prefixing operation on processes is defined as $\llbracket ai.[\cdot] \rrbracket^{\dagger}$; *i.e.* by lifting the operation to sets of functions pointwise.

Pictorially, input prefixing on a pomset operator is done as follows. The box indicates synchronization as before. The modeling of the fact that input prefixing $ai(x)$ binds the name x (and hence $\bar{x}$) is based on the treatment of multiplicatives in the geometry of interaction. In particular, the input information for x is extracted from the input on a, and the output information on x is included in the output on a. Similarly for $\bar{x}$. Also, note the absence of any connection between the external inputs/outputs on x and the internal inputs/outputs on x. Similarly for $\bar{x}$.

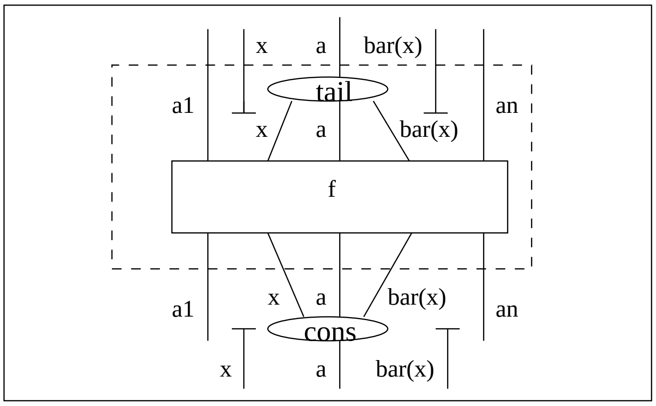


Fig. 3. Input prefixing in the pi-calculus

Formally, we have the following definition. $\llbracket \mathtt{ai(x).f} \rrbracket \mathbf{v} = \mathbf{w}$ where

$$v'_{aj} = \left\{ \begin{array}{l} \pi_{aj}(\mathbf{v}), \text{ if } aj \notin \{ai, x, \bar{x}\} \\ \mathtt{tail}(v_{aj}), \text{ if } aj = ai \\ \pi_1(\mathtt{head})(v_{ai}), \text{ if } aj = x \\ \pi_2(\mathtt{head})(v_{ai}), \text{ if } aj = \bar{x} \end{array} \right\} \quad \mathbf{w}' = \left\{ \begin{array}{l} f(\mathbf{v}'), \text{ if } v_{ai} \neq \perp \\ \perp, \text{ otherwise} \end{array} \right\}$$

$$w_{aj} = \left\{ \begin{array}{l} w'_{aj}, \text{ if } aj \notin \{ai, x, \bar{x}\} \\ \mathtt{cons}((w'_x, w'_{\bar{x}}), v_{aj}), \text{ if } aj = ai \\ \perp, \text{ if } aj \in \{x, \bar{x}\} \end{array} \right\}$$

Output prefixing We define an output prefixing operation $\llbracket \bar{ai}(x).[\cdot] \rrbracket$ on pomset operators; this operation yields an element of $P_R(P_{\mathtt{PiC}})$. The output prefixing operation on processes is defined as $\llbracket \bar{ai}(x).[\cdot] \rrbracket^{\dagger}$; *i.e.* by lifting the operation to sets of functions pointwise.

Pictorially, output prefixing on a pomset operator is done as follows. The synchronization box has been omitted to avoid cluttering up the figure. The important difference from the input prefixing is that the name x is not bound. This leads to two issues:

- The environment can interact independently with the name x. The nodes titled `M1x` and `S1x` in the figure mix the information on x from the environment and from the channel $\bar{a}$. Simlarly for $\bar{x}$.
- There is one further level of complication arising from *scope extrusion*. Output prefixing $\bar{a}(x)$ can export the environment's interaction on x to the output on $\bar{a}$ pertaining to $\bar{x}$. Similarly for $\bar{x}$. The required interaction is modeled by the dotted arrows in the figure.

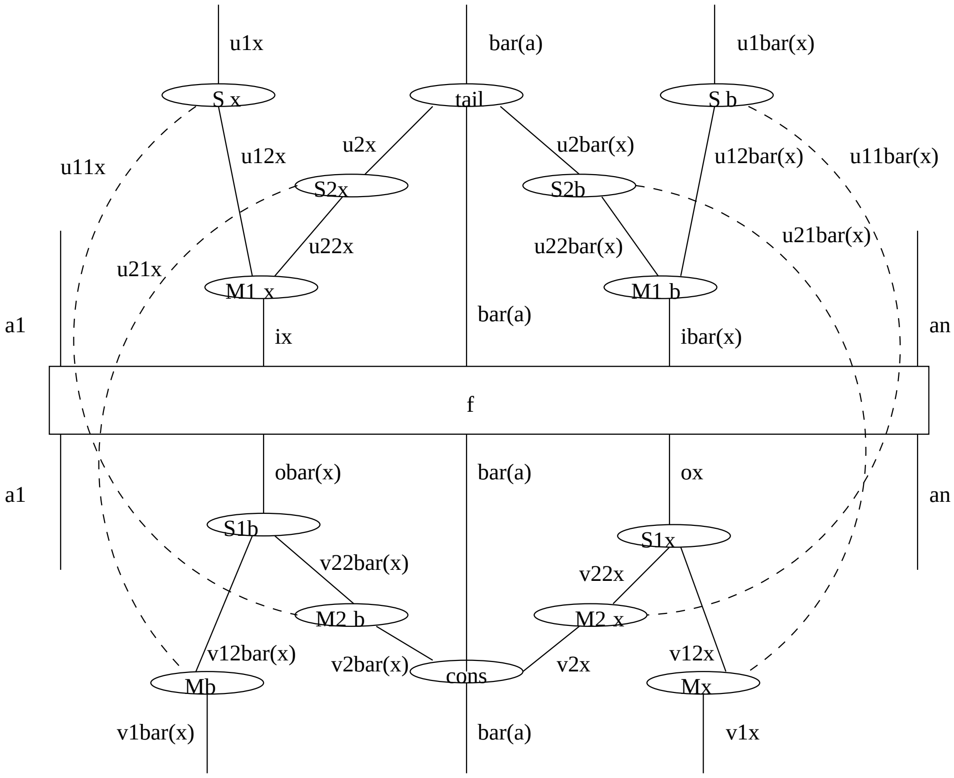


Fig. 4. Output prefixing in the pi-calculus

Formally, we proceed by writing down the equations implicit in the above figure. Let $S_s, S1_s, S2_s$ where $s \in \{x, \bar{x}\}$, be determinate splits. Let $M_s, M1_s, M2_s$, where $s \in \{x, \bar{x}\}$ be determinate merges such that (S_s, M_s), $(S1_s, M1_s)$ and $(S2_s, M2_s)$, where $s \in \{x, \bar{x}\}$ are pairs of functions that are inverses to each other. We are going to describe $\mathbf{z} = \llbracket \bar{ai}(x).f \rrbracket(\mathbf{r})$. There are two cases. If $\pi_{\bar{a}}(\mathbf{r}) =\perp$, $\mathbf{z} = \mathtt{cons}_{\bar{a}}((\perp,\perp),\perp)$.

Otherwise, we proceed as follows. $\mathbf{w}$, the input into f, is defined by the following equations:

$$\begin{array}{l}
u_2^x = \pi_2(\mathtt{head}(\pi_{\bar{a}}(\mathbf{r}))),\ u_2^{\bar{x}} = \pi_1(\mathtt{head}(\pi_{\bar{a}}(\mathbf{r}))) \\
\pi_s(\mathbf{r}) = u_1^s, S_s(u_1^s) = (u_{11}^s, u_{12}^s), \text{ for } s \in \{x, \bar{x}\} \\
S2_s(u_2^s) = (u_{21}^s, u_{22}^s), M1_s(u_{12}^s, u_{22}^s) = i^s, \text{ for } s \in \{x, \bar{x}\} \\
\pi_{\bar{a}}(\mathbf{w}) = \mathtt{tail}(\pi_{\bar{a}}(\mathbf{r})), \pi_s(\mathbf{w}) = i^s, \text{ for } s \in \{x, \bar{x}\} \\
\pi_t(\mathbf{w}) = \pi_t(\mathbf{r}), \text{ otherwise}
\end{array}$$

Let $f(\mathbf{w}) = \mathbf{w}'$. The result $\mathbf{z}$, is defined by the following equations:

$$\begin{array}{l}
\pi_s(\mathbf{w}') = o^s, S1_s(o^s) = (v_{12}^s, v_{22}^s), \text{ for } s \in \{x, \bar{x}\} \\
M2_s(u_{11}^{\bar{s}}, v_{22}^s) = v_s^2, M_s(u_{21}^{\bar{s}}, v_{12}^s) = v_1^s, \text{ for } s \in \{x, \bar{x}\} \\
\pi_{\bar{a}}(\mathbf{z}) = \mathtt{cons}((v_2^{\bar{x}}, v_2^x), \pi_{\bar{a}}(\mathbf{w}')) \\
\pi_s(\mathbf{z}) = v_1^s,\ s \in \{x, \bar{x}\} \\
\pi_t(\mathbf{x}) = \pi_t(\mathbf{w}'), \text{ otherwise}
\end{array}$$

As in the treatment of parallel composition of CCS, the above equations describe a pomset operator for every choice of merge. As before, every determinate merge (and hence the corresponding split) can be encoded as an element of $\mathsf{Seq}(Z)$. In this light, the equations above describes a continuous function $H : \Pi_6 \mathsf{Seq}(Z) \to P_{\mathsf{CCS}}$; as before, we consider $H^\dagger$ and define

$$\llbracket \bar{ai}(x).f \rrbracket = H^\dagger(\top)$$

3.2 Results

Firstly, the semantics is a true concurrency semantics. This calculation is similar to the calculation done earlier for CCS.

Lemma 1. $\llbracket \bar{a}(b) | \bar{c}(d)] \rrbracket \neq \llbracket \bar{a}(b).\bar{c}(d) + \bar{c}(d).\bar{a}(b) \rrbracket$.

Next, we note that the semantics described above does not make distinguish between the "subject" and "object" dependencies of [4]. This is exemplified by the treatment of the example drawn from [4].

Lemma 2. $\llbracket \nu(b)[\bar{a}(b) | \bar{b}(y)] \rrbracket = \llbracket \nu(b)[\bar{a}(b).\bar{b}(y)] \rrbracket$.

At this point, we are not aware of any other work that treats causality in the PI–CALCULUS as a uniform notion. Consequently, we are unable to establish a point of reference for comparison. (This was possible for CCS because of the rich literature that investigates true concurrency.) Here, we restrict ourselves to a basic adequacy theorem for interleaving traces on the standard transition system [18] for the PI–CALCULUS. We follow the notation of [18]; $\equiv$ stands for structural congruence and $\to$ stands for the one-step reduction relation.

Theorem 2

- $P \equiv Q \Rightarrow [\![P]\!] = [\![Q]\!]$
- $P \rightarrow Q \Rightarrow [\![Q]\!] \sqsubseteq [\![P]\!]$
- $[\![P]\!] = [\![Q]\!]$ *implies that for all process contexts* $C[\cdot]$, $\mathtt{Traces}(C[P]) = \mathtt{Traces}(C[Q])$.

The first item above states that the structural congruence is reflected as equality in the semantics. The proof proceeds by a straightforward inductive analysis of rules used in [18] to deduce that $P \equiv Q$. The second item is the analogue of the "one step reduction preserves meaning" property. We note that the inequality (as opposed to the more familiar equality in semantics of deterministic transition systems) is characteristic of non-deterministic transition systems. Intuitively, when P reduces one–step to Q, some potential reductions of P may not be available for Q. The key case in this proof is the reduction $a(x).P||\bar{a}(y).Q \rightarrow P[x/y]||Q$. The essence of the proof — namely that the model is correct with respect to substitution — is handled by an adaptation of the soundness proof of geometry of interaction models (see for example [1]).

Acknowledgments The authors thank the anonymous referees for valuable comments. This paper owes a great deal to the work and ideas of Samson Abramsky — indeed this work is a flushing out (in the setting of the PI–CALCULUS) of the analysis of computation in the geometry of interaction. The authors also wish to acknowledge the inspiration afforded by the work of Simon Gay on the relationship between interaction nets and determinate CCS. The authors are of course responsible for any errors and omissions.

References

1. S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. *Information and Computation*, 111(1):53–119, 1994.
2. Samson Abramsky. A generalized Kahn principle for abstract asynchronous networks. In *Mathematical Foundations of Programming Semantics*, pages 1–21. Springer-Verlag, 1989. Lecture Notes in Computer Science Vol. 442.
3. L. Aceto and M. Hennessy. Towards action-refinement in process algebras. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 138–145. IEEE Computer Society Press, 1989.
4. M. Boreale and S. Sangiorgi. A fully abstract semantics for causality in the π-calculus. Technical report, University of Edinburgh, 1994. Tec. Rep.ECS–LFCS–94–297.
5. G. Boudol, I. Castellani, Matthew C. Hennessy, and A. Kiehn. A theory of processes with localities. In *Proceedings of International Conference on Concurrency Theory, Volume 630 of Lecture Notes in Computer Science*, pages 108–122, 1992.
6. T. Coquand, C. Gunter, and G. Winskel. dI-domains as a model of polymorphism. In *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, pages 344–363. Springer-Verlag, 1987.

7. D. Degano and D. Darondeau. Causal trees. In *Proceedings of ICALP '89, Volume 372 of Lecture Notes in Computer Science*, pages 234–248, 1989.
8. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
9. J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro et al., editor, *Logic Colloquium 88*. North Holland, 1989.
10. Ursula Goltz. CCS and Petri nets. Technical report, GMD, July 1990.
11. Roberto Gorrieri. *Refinement, Atomicity, and Transactions for Process Description Languages*. PhD thesis, University of Pisa, 1991.
12. Lalita Jategaonkar. *Observing "True" Concurrency*. PhD thesis, Massachusetts institute of Technology, September 1993. Supervised by Albert R. Meyer.
13. Lalita Jategaonkar and Albert R. Meyer. Testing equivalence for Petri nets with action refinement. In *Proceedings of International Conference on Concurrency Theory, Volume 630 of the Lecture Notes in Computer Science*, pages 17–31, 1992.
14. G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1977.
15. G. Kahn and D. MacQueen. Coroutines and networks of parallel proceses. In *Proceedings of IFIP Congress*. North-Holland, 1977.
16. A. Kiehn. Local and global causes. Technical report, Technische Universitat Munchen, 1991.
17. R. Milner. Functions as processes. In *Proceedings of ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1990.
18. R. Milner. The polyadic pi-calculus: a tutorial. Technical Report ECS-LFCS-91-180, University of Endinburgh, 1991.
19. Robin Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall, 1989.
20. M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains i. *Theoretical Computer Science*, 13(1):85–108, 1981.
21. Vaughan Pratt. Modeling concurrency with partial orders. *International Journal Of Parallel Programming*, 15(1):33–71, 1986.
22. J. R. Russell. On oraclizable networks and kahn's principle. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1990.
23. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.
24. D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. In *Proceedings of ICALP '94, Lecture Notes in Computer Science*, 1994.
25. Rob van Glabbeek and Frits Vaandrager. Petri net models for algebraic theories of concurrency. In *Proceedings of PARLE, Volume 259 of the Lecture Notes in Computer Science*, pages 224–242, 1987.
26. Walter Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992. 252 pp.
27. David Walker. π-calculus semantics of object-oriented programming languages. In *Theoretical Aspects of Computer Software*, pages 532–547, 1991.
28. Glynn Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Volume 255 of Lecture Notes in Computer Science*, pages 325–392, 1987.

This article was processed using the LaTeX macro package with LLNCS style