Traces, pomsets, fairness and full abstraction for communicating processes

Stephen Brookes
Department of Computer Science
Carnegie Mellon University

Abstract

We provide a trace-based semantics for a language of synchronously communicating processes, assuming weakly fair parallel execution. The semantics is fully abstract: processes have the same trace sets if and only if their communication behaviors, including potential for deadlock, are identical in all program contexts. We avoid the traditional use of book-keeping information such as refusal sets, failures, and divergence traces; instead traces include the relevant information directly. The semantics can easily be adapted to model blocking or non-blocking communication guards, asynchronous communication and shared-memory parallelism. Thus we obtain a flexible model with a simple yet adaptable structure which emphasizes the underlying similarities between parallel paradigms. We also provide a compositional partial-order description of trace sets, adapting ideas from pomset semantics to incorporate fairness and synchronization. The pomset semantics can also be adapted easily to model alternative forms of guard, asynchronous communication and shared memory. In each case the trace set of a process can be recovered from the pomset semantics by taking all fair interleavings consistent with the partial order. We illustrate the utility of our semantics by analyzing the behavior of a number of examples, and by listing some laws of semantic equivalence which rely on fairness and can be used in analyzing process behavior.

1 Introduction

Traces of various kinds have been used widely to model parallel programs, with parallel execution being interpreted as some form of (fair) interleaving. *Transition traces*, finite or infinite sequences of pairs of states, can be used for shared-memory parallel programs [3], for concurrent logic programs and concurrent constraint programs [2][1], and for networks of asynchronously communicating processes [6], assuming weakly fair execution. Transition traces also provide a semantics of a parallel Algol-like language with block structure, procedures, shared-memory parallelism, and asynchronous message-passing [4, 5]. *Communication traces*, sequences of input/output events, were the basis for an early model of CSP [16, 17], later augmented with *refusal sets* to permit deadlock analysis in the *failures* model of CSP [10], and with *divergence traces* in the *failures-divergences* model of CSP [11][2]. *Pomset traces* provide a partial-order based framework for making behavioral distinctions based on "true concurrency" rather than interleaving[25, 26].

Fairness assumptions [14], such as *weak process fairness*, the assumption that each persistently enabled process is eventually scheduled, allow us to abstract away from unknown or unknowable implementation details. Fairness plays a crucial role in proving safety and liveness properties of parallel systems [22]. A safety properties has the typical form that something "bad" never happens. A liveness property typically asserts that something "good" will eventually happen [21].

CSP [16] is a language of synchronously communicating parallel processes; a process attempting to output must wait until another process reaches a matching input, and *vice versa*. The early denotational models of CSP [17, 10, 11, 27] were not designed to take fairness into account, and it seems difficult to adapt them for this purpose. Moreover, there is a plethora of fairness notions, including strong and weak forms of process fairness, channel fairness, and communication fairness [14]; it is not clear which (if any) of these fairness

[1]The term *reactive sequence* is used in [2] and related papers to refer to a sequence of pairs of states. The use of this kind of sequence to model concurrency dates back at least to Park's seminal papers on the subject [22]. Although these models use the same kind of trace they focus on different notions of observable behavior and (correspondingly) incorporate different closure rules for trace sets.

[2]Roscoe's book [27] gives a detailed account of these and related models of CSP. Van Glabbeek's article [15] provides a wide-ranging and detailed survey of process algebras and notions of behavioral equivalence.

notions is a reasonable abstraction for synchronously communicating processes, although strong and weak process fairness can at least be characterized intuitively and mathematically in relatively straightforward fashion and weak process fairness can be guaranteed by a "reasonable" scheduler using a simple round-robin strategy. Costa and Stirling have shown how to provide an operational semantics for a CCS-like language assuming either weak or strong process fairness [12, 13]. Older's thesis [19] and related papers [9, 20] show that one can treat some of these fairness notions denotationally by augmenting failure-style models still further, with book-keeping information concerning processes, communications, and synchronizations which become persistently enabled but not scheduled. However, this results in rather complicated models. Much of the difficulty is caused by the fact that the ability of a process to communicate depends on the availability of another process which is capable of synchronizing. Indeed, in Older's formulation even weak process fairness fails to be *equivalence robust* [1], in that there is a pair of computations, one fair and one unfair, which differ only in the interleaving order of independent actions [19].

In contrast, if we assume instead that communication is asynchronous, so that a process attempting to output is always enabled to do so autonomously, and a process attempting an input must wait if there is no available input, we can use transition traces to obtain a fully abstract model incorporating weak process fairness [6]. Moreover, in the asynchronous setting weak process fairness is equivalence robust. One can also formulate an equivalent semantics using a suitably designed form of communication traces [7, 8].

The disparity between the relatively simple trace semantics for asynchronously communicating processes and the intricate book-keeping semantics for synchronously communicating processes obscures the underlying similarities between the two paradigms. It seems to be widely believed that this disparity is inevitable, since traces are too simple a notion to support the combination of deadlock, fairness, and synchronized communication. This is certainly a valid criticism of traditional trace-based accounts of CSP, which used prefix-closed sets of finite traces (augmented with refusal sets) and allowed infinite traces to be handled implicitly based on their finite prefixes. Although the traditional models of CSP give an adequate account of blocking and deadlock as well as more general safety properties, they do not adequately support liveness analysis since they do not admit fairness: the existence of a fair infinite trace for a process does not follow from the process's ability to perform each of its finite prefixes.

In this paper we show that, if we assume a reasonable (weak and robust) notion of fairness, a satisfactory trace semantics can be designed; the key is to choose the right notion of trace and avoid unnecessary closure assumptions. Indeed, the *same* notion of trace can be used both for synchronously communicating processes and for asynchronously communicating processes. In each case we model a weak form of fairness which is consistent with a simple form of round-robin scheduling, even when communication requires synchronization, so that we obtain a good abstraction of process behavior independent of implementation details[3]. In each case the trace semantics is compositional, and supports safety and liveness analysis. Indeed our semantics is *fully abstract*, in the sense that two processes have the same trace set if and only if they exhibit identical communication behavior (including the potential for deadlock) in all program contexts. We do not need to augment traces with extraneous book-keeping information, or to impose complex closure conditions, in order to achieve these results.

Our achievement is perhaps surprising, given the long history of largely separate development of semantic frameworks for the two communication paradigms: traditional denotational models of asynchronous communicating processes and synchronous communicating processes have frustratingly little in common. There is little family resemblance, for instance, between the failures model of CSP and transition traces. In contrast, we treat both kinds of communication as straightforward variations on a trace-theoretic theme, so that we achieve a semantic unification of parallel communication paradigms. Given our prior results concerning the utility of trace semantics for shared-memory parallelism, the unification goes further still.

We also show how to model fairness and synchronous communication in the pomset framework. We define a partial-order based semantics in which a process denotes a set of partially ordered multisets of actions (pomsets). Each pomset determines a set of traces obtainable by fair interleaving consistent with the partial order. The trace set of a process can be recovered in this way from its pomset semantics. The pomset semantics supports a style of reasoning which avoids dealing explicitly with interleaving, and this may help to tame the combinatorial explosion inherent in analyzing parallel systems.

[3]By this we mean that a family of simple round-robin schedulers can be defined, such that each member of this family ensures weakly fair execution, and every weakly fair execution is allowed by some such scheduler. To handle synchronization we assume that if the process currently scheduled is waiting for communication the scheduler will use a round-robin strategy to see if another process is ready to perform a matching communication.

We can also adapt pomset semantics to model asynchronous communication, with a small number of simple changes to the semantic definitions.

In the rest of this abstract we sketch some of the technical development for synchronously communicating processes. The full paper will contain proofs, more complex examples, and a fuller summary of related work. For concreteness we focus on a CSP-style language in which communication guards are modelled as blocking, as in Hoare's original language [16]. We can adapt our definitions and results to handle alternative language design decisions, for example non-blocking guards, mixed boolean and input/output guards, and general recursive process definitions. We attach a brief Appendix summarizing the adjustments for asynchronous communication. The full paper will also show how to adapt our techniques for shared-variable parallel programs.

2 Syntax

Let P range over processes, G over "guarded" processes, h over the set **Chan** of channel names, x over the set **Ide** of integer-valued variables, e over integer-valued identifiers, and b over boolean-valued expressions, given by the following abstract grammar:

$$\begin{array}{rcl} P & ::= & \mathbf{skip} \mid x{:=}e \mid h?x \mid h!e \mid P_1; P_2 \mid P_1 \| P_2 \mid P_1 \sqcap P_2 \mid \\ & & \mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \mid \mathbf{while}\ b\ \mathbf{do}\ P \mid \mathbf{local}\ h\ \mathbf{in}\ P \\ G & ::= & (h?x \to P) \mid G_1 \square G_2 \end{array}$$

As usual for CSP, we include two forms of choice: $P_1 \sqcap P_2$ is "internal" choice, and $G_1 \square G_2$ is "external" choice. Note that $(h?x \to P)$ will be treated as semantically equivalent to $h?x; P$.

Although we have only included while-loops above, the ensuing definitions and results can be adapted to handle recursion and more general guarded conditional commands **if** G **fi** and loops **do** G **od**. We can also handle mixed guards combining boolean expressions with communication, and we can model both blocking and non-blocking versions of input/output. One can also allow local variable declarations, as in **local** x **in** P.

3 Actions and traces

Let V be the set of integers, with typical member v. An action has form $h?v, h!v, x = v, x{:=}v$ or δ_X, where X is a finite set of "directions" ($h?$ or $h!$).

A *communication action* $h?v$ or $h!v$ represents a *potential* for communication, and can only be completed by a process if and when another (concurrent) process is able to match this action with the corresponding $h!v$ or $h?v$. We use $x{=}v$ to represent an *evaluation action* in which x is discovered to have value v, and $x{:=}v$ to represent an *assignment action* updating the value of x. Each communication action has a corresponding *direction*, and we let $\mathbf{Dir} = \{h?, h! \mid h \in \mathbf{Chan}\}$ be the set of directions. A *blocking action* of form δ_X represents an unrequited attempt to communicate along the directions in the set X. When X is a singleton we write $\delta_{h?}$ or $\delta_{h!}$. When X is empty we write δ instead of $\delta_{\{\}}$; the action δ is also used to represent a "silent" local action, such as a synchronized handshake or reading or writing a local variable. We let $X \backslash h = X - \{h?, h!\}$. Let Σ stand for the set of all actions, Λ for the set of communication actions, and Δ for the set of blocking actions:

$$\Lambda = \{h?v, h!v \mid h \in \mathbf{Chan}\ \&\ v \in V\} \quad \Delta = \{\delta_X \mid X \subseteq \mathbf{Dir}\}.$$

A trace is a finite or infinite sequence of actions representing a potential behavior of a process. We model persistent waiting for communication and divergence (infinite local activity) as an infinite sequence of blocking actions. We assume that unless and until blocking or divergence occurs we only care about the non-silent actions taken by a process[4]. Accordingly, we assume when concatenating that $\delta\lambda = \lambda\delta = \lambda$ for all actions λ, and we suppress waiting actions which lead to successful communication, so that $\delta_{h?}^{*}h?v = h?v$ for example. A trace of the form $\alpha{\delta_X}^{\omega}$ describes an execution in which the process performs α then gets stuck waiting to communicate along the directions in X. For a trace β we define the set of blocked directions in β, written $\mathit{blocks}(\beta)$, as the set of all directions which occur infinitely often in blocking steps of β. For example, $\mathit{blocks}(a!0(\delta_{b?}\delta_{c?})^{\omega}) = \{b?, c?\}$.

4 Denotational semantics

We assume given the semantics of expressions: the trace set denoted by expression e, written $\mathcal{T}(e)$, describes all possible evaluation behaviors of e, and consists of all pairs (ρ, v) where ρ is a sequence of evaluation steps which yield value v for the expression. For example, for an identifier y we have

[4]Other notions of observable behavior, such as the assumption that we can see all actions that occur, including blocking steps, can also be incorporated with appropriate modifications to the semantic definitions.

$\mathcal{T}(y) = \{(y = v, v) \mid v \in V\}$ and for a numeral $\underline{n}$ we have $\mathcal{T}(\underline{n}) = \{(\delta, n)\}$. For a boolean expression b we let $\mathcal{T}(b)_{\mathbf{true}} = \{\rho \mid (\rho, \mathbf{true}) \in \mathcal{T}(b)\}$.

Definition 1 *The trace set denoted by process P, written $\mathcal{T}(P)$, is defined compositionally as follows:*

$$\begin{array}{rcl}
\mathcal{T}(\mathbf{skip}) & = & \{\delta\} \\
\mathcal{T}(x{:=}e) & = & \{\rho\, x{:=}v \mid (\rho, v) \in \mathcal{T}(e)\} \\
\mathcal{T}(h?x) & = & \{h?v\, x{:=}v \mid v \in V\} \cup \{\delta_{h?}{}^{\omega}\} \\
\mathcal{T}(h!e) & = & \{\rho\, h!v,\ \rho\, \delta_{h!}{}^{\omega} \mid (\rho, v) \in \mathcal{T}(e)\} \\
\mathcal{T}(P_1; P_2) & = & \mathcal{T}(P_1)\mathcal{T}(P_2) = \{\alpha_1\alpha_2 \mid \alpha_1 \in \mathcal{T}(P_1)\ \&\ \alpha_2 \in \mathcal{T}(P_2)\} \\
\mathcal{T}(P_1 \| P_2) & = & \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in \mathcal{T}(P_1)\ \&\ \alpha_2 \in \mathcal{T}(P_2)\} \\
\mathcal{T}(\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) & = & \mathcal{T}(b)_{\mathbf{true}}\mathcal{T}(P_1)\ \cup\ \mathcal{T}(b)_{\mathbf{false}}\mathcal{T}(P_2) \\
\mathcal{T}(\mathbf{while}\ b\ \mathbf{do}\ P) & = & (\mathcal{T}(b)_{\mathbf{true}}\mathcal{T}(P))^{*}\mathcal{T}(b)_{\mathbf{false}}\ \cup\ (\mathcal{T}(b)_{\mathbf{true}}\mathcal{T}(P))^{\omega} \\
\mathcal{T}(\mathbf{local}\ h\ \mathbf{in}\ P) & = & \{\alpha \backslash h \mid \alpha \in \mathcal{T}(P)\ \&\ h \notin \mathit{chans}(\alpha)\} \\
\mathcal{T}(G_1 \Box G_2) & = & \{\alpha \in \mathcal{T}(G_1) \cup \mathcal{T}(G_2) \mid \alpha \notin \Delta^{\omega}\}\ \cup \\
 & & \{\delta_{X \cup Y}{}^{\omega} \mid \delta_X{}^{\omega} \in \mathcal{T}(G_1)\ \&\ \delta_Y{}^{\omega} \in \mathcal{T}(G_2)\} \\
\mathcal{T}(P_1 \sqcap P_2) & = & \mathcal{T}(P_1) \cup \mathcal{T}(P_2)
\end{array}$$

Given two traces α_1 and α_2, $\alpha_1 \| \alpha_2$ is the set of all traces formed by merging them fairly, allowing synchronization of matching communication actions. We let $\alpha \| \epsilon = \epsilon \| \alpha = \{\alpha\}$. When α_1 and α_2 are finite and non-empty, say $\alpha_i = \lambda_i \beta_i$, we let

$$\begin{array}{rcl}
(\lambda_1\beta_1) \| (\lambda_2\beta_2) & = & \{\lambda_1\gamma \mid \gamma \in \beta_1 \| (\lambda_2\beta_2)\ \cup\ \{\lambda_2\gamma \mid \gamma \in (\lambda_1\beta_1) \| \beta_2\} \\
 & & \cup\ \{\delta\gamma \mid \gamma \in \beta_1 \| \beta_2\ \&\ \mathit{match}(\lambda_1, \lambda_2)\}
\end{array}$$

When α_1 and α_2 are infinite, we let $\alpha_1 \| \alpha_2 = \{\}$ if some direction in $\mathit{blocks}(\alpha_1)$ matches a direction in $\mathit{blocks}(\alpha_2)$, since it is unfair to avoid synchronizing two processes which are blocked but trying to synchronize on a common channel. Otherwise we let $\alpha_1 \| \alpha_2$ consist of all traces of form $\gamma_1\gamma_2 \ldots$ where α_1 can be written as a sequence of finite chunks $\alpha_{1,1}\alpha_{1,2} \ldots$, and α_2 can be written as $\alpha_{2,1}\alpha_{2,2} \ldots$, and each γ_i is a fair merge of $\alpha_{1,i}$ and $\alpha_{2,i}$.

For example, $\delta_{h!}{}^{\omega} \| \delta_{h?}{}^{\omega} = \{\}$ and $(a!0\delta_{h!}{}^{\omega}) \| (b!1\delta_{h?}{}^{\omega}) = \{\}$. However, $\delta_{a!}{}^{\omega} \| \delta_{b?}{}^{\omega}$ is non-empty and can be written in the form $(\delta_{a!}{}^{*}\delta_{b?}\delta_{b?}{}^{*}\delta_{a!})^{\omega}$.

We write $\mathit{chans}(\alpha)$ for the set of channels occurring in input or output actions along α, and when $h \notin \mathit{chans}(\alpha)$ we let $\alpha \backslash h$ be the trace obtained from α by replacing every δ_X with $\delta_{X \backslash h}$. For instance, the trace $(a!0\, \delta_{h?}{}^{\omega}) \backslash h$ is $a!0\, \delta^{\omega}$.

5 Operational semantics

A state s is a mapping from program variables to values.[5] An action may or may not be enabled in a given state. For instance, the action $x = v$ is only enabled in a state for which the value of x is v.

We assume given an operational semantics for expressions, with transitions of form $e, s \xrightarrow{\mu} e', s$ and $e, s \xrightarrow{\mu} v$, where μ is an evaluation action and v is an integer. The operational semantics for boolean expressions is similar. An assignment action changes the state. We write $[s \mid x : v]$ for the state obtained from s by updating the value of x to v.

The operational semantics for processes is specified by a labelled transition relation $P, s \xrightarrow{\lambda} P', s'$ and a termination predicate P, s **term**. Some of the most relevant rules are listed in Figure 1. (We omit several rules, including those dealing with sub-expression evaluation.)

A *transition sequence* of process P is a sequence of transitions of form

$$\begin{array}{l} P, s_0 \xrightarrow{\lambda_0} P_1, s'_0 \\ P_1, s_1 \xrightarrow{\lambda_1} P_2, s'_1 \\ P_2, s_2 \xrightarrow{\lambda_2} P_3, s'_2 \\ \cdots, \end{array}$$

either infinite or ending in a terminal configuration. A *computation* is a transition sequence in which the state never changes between steps, so that $s'_i = s_{i+1}$.

A transition sequence (or a computation) of P is *fair* if it contains a complete transition sequence for each syntactic sub-process of P, and no pair of sub-processes is permanently blocked yet attempting to synchronize. For example, the computation

$$a?x \| a!0, s \xrightarrow{\delta_{a?}} a?x \| a!0, s \xrightarrow{\delta_{a!}} a?x \| a!0, s \xrightarrow{\delta_{a?}} a?x \| a!0, s \xrightarrow{\delta_{a!}} \cdots$$

is not fair, because the two processes block on matching directions. However,

$$a?x \| a!0, s \xrightarrow{a?1} x{:=}1 \| a!0, s \xrightarrow{x:=1} \textbf{skip} \| a!0, s' \xrightarrow{\delta_{a!}} \textbf{skip} \| a!0, s' \xrightarrow{\delta_{a!}} \cdots$$

where $s' = [s \mid x{:}1]$, is fair because only one process is blocked. Indeed, there is a fair computation of the process $a!1 \| (a?x \| a!0)$ in which the first process performs $a!1$ and the second performs the above transition sequence.

[5]Since channels are only used for synchronized handshaking there is no need to treat channel contents as part of the state.

$$\frac{}{\mathbf{skip}, s\ \mathbf{term}}$$

$$\frac{}{h?x, s \xrightarrow{h?v} x{:=}v, s} \qquad \frac{}{h?x, s \xrightarrow{\delta_{h?}} h?x, s}$$

$$\frac{}{h!v, s \xrightarrow{h!v} \mathbf{skip}, s} \qquad \frac{}{h!v, s \xrightarrow{\delta_{h!}} h!v, s}$$

$$\frac{G_1, s \xrightarrow{\delta_X} G_1, s \quad G_2, s \xrightarrow{\delta_Y} G_2, s}{G_1 \Box G_2, s \xrightarrow{\delta_{X \cup Y}} G_1 \Box G_2, s}$$

$$\frac{G_1, s \xrightarrow{\lambda} P_1, s' \quad \lambda \notin \Delta}{G_1 \Box G_2, s \xrightarrow{\lambda} P_1, s'} \qquad \frac{G_2, s \xrightarrow{\lambda} P_2, s' \quad \lambda \notin \Delta}{G_1 \Box G_2, s \xrightarrow{\lambda} P_2, s'}$$

$$\frac{}{P_1 \sqcap P_2, s \xrightarrow{\delta} P_1, s} \qquad \frac{}{P_1 \sqcap P_2, s \xrightarrow{\delta} P_2, s}$$

$$\frac{P_1, s \xrightarrow{\lambda} P_1', s'}{P_1 \| P_2, s \xrightarrow{\lambda} P_1' \| P_2, s'} \qquad \frac{P_2, s \xrightarrow{\lambda} P_2', s'}{P_1 \| P_2, s \xrightarrow{\lambda} P_1 \| P_2', s'} \qquad \frac{P_1, s\ \mathbf{term} \quad P_2, s\ \mathbf{term}}{P_1 \| P_2, s\ \mathbf{term}}$$

$$\frac{P_1, s \xrightarrow{\lambda_1} P_1', s \quad P_2, s \xrightarrow{\lambda_2} P_2', s \quad match(\lambda_1, \lambda_2)}{P_1 \| P_2, s \xrightarrow{\delta} P_1' \| P_2', s}$$

$$\frac{P, s \xrightarrow{\lambda} P', s' \quad chan(\lambda) \neq h}{\mathbf{local}\ h\ \mathbf{in}\ P, s \xrightarrow{\lambda} \mathbf{local}\ h\ \mathbf{in}\ P', s'}$$

$$\frac{P, s \xrightarrow{\delta_X} P', s}{\mathbf{local}\ h\ \mathbf{in}\ P, s \xrightarrow{\delta_{X \setminus h}} \mathbf{local}\ h\ \mathbf{in}\ P', s'}$$

$$\frac{P, s\ \mathbf{term}}{\mathbf{local}\ h\ \mathbf{in}\ P, s\ \mathbf{term}}$$

Figure 1: Operational semantics for processes

6 Relating denotational and operational

The denotational and operational characterizations of fair traces coincide:

Theorem 1 *For every process P, $\mathcal{T}(P)$ consists of the traces generated by the fair transition sequences of P.*

7 Full abstraction and trace equivalence

Suppose we can observe communication sequences, including blocking steps, and the values of non-local variables, but we cannot backtrack to try alternative runs. This notion of observable behavior suffices to allow safety and liveness analysis. For example, one might want to prove that a parallel system ensures that no process becomes blocked on a given channel. Our trace semantics is fully abstract with respect to this notion of behavior:

Theorem 2 *Two processes P_1 and P_2 have the same trace sets iff they have the same observable behavior in all contexts.*

Although an obvious corollary of compositionality, this result generalizes analogous well known full abstraction results for failures semantics, which hold in a much more limited setting, without fairness [27]. The significance of this result is not full abstraction *per se* but the construction of a *simple* trace-based semantics that incorporates a reasonable form of fair parallelism and synchronized communication while supporting safety and liveness analysis; this kind of simplicity is itself a virtue.

To demonstrate that trace semantics distinguishes between processes with different deadlock capabilities, note that:

$$\delta_X{}^\omega \in \mathcal{T}((a?x \to P) \,\square\, (b?x \to Q)) \iff X = \{a?, b?\}$$

$$\delta_X{}^\omega \in \mathcal{T}((a?x \to P) \sqcap (b?x \to Q)) \iff X = \{a?\} \text{ or } X = \{b?\}.$$

If we run these processes in a context which is only capable of communicating on channel b, such as

$$\textbf{local } a, b \textbf{ in } ([-] \| b!0)$$

the first process would behave like $x{:=}0;$ **local** a, b **in** Q but the second would also have the possibility of behaving like **local** a, b **in** $((a?x \to P)\|b!0)$, which is deadlocked and has the trace set $\{\delta^\omega\}$.

Fair synchronous laws

The following semantic equivalences, to be interpreted as equality of trace sets, illustrate how our model supports reasoning about process behavior.

Theorem 3 *The synchronous trace semantics validates the following laws of equivalence:*

1. $\textbf{local}\ h\ \textbf{in}\ (h?x; P)\|(h!v; Q)\|R \ = \ \textbf{local}\ h\ \textbf{in}\ (x{:=}v; P)\|Q\|R$
   provided $h \not\in \texttt{chans}(R)$.
2. $\textbf{local}\ h\ \textbf{in}\ (h?x; P)\|(Q_1; Q_2) \ = \ Q_1; \textbf{local}\ h\ \textbf{in}\ (h?x; P)\|Q_2$
   provided $h \not\in \texttt{chans}(Q_1)$.
3. $\textbf{local}\ h\ \textbf{in}\ (h!v; P)\|(Q_1; Q_2) \ = \ Q_1; \textbf{local}\ h\ \textbf{in}\ (h!v; P)\|Q_2$
   provided $h \not\in \texttt{chans}(Q_1)$.

These properties, which reflect our assumption of (a weak form of) fairness, can be particularly helpful in proving liveness properties. They are not valid in an unfair semantics. For instance, if execution is unfair there is no guarantee in the first law that the synchronization will eventually occur, and there is no guarantee in the second or third laws that the right-hand process will ever execute its initial (non-local) code.

8 Pomset semantics

A *pomset* $(T, <)$ is a partially ordered multiset of actions: T is a multiset whose elements are drawn from the set Σ of actions, and $<$ is a partial order on T, representing a "precedence" relation on the action occurrences in T. Actually we allow the precedence relation to be a pre-order: when T contains a pair of matching communication occurrences which precede each other this will force a synchronization. We also assume that the ordering has no accumulation points, i.e. that every action dominates finitely many actions, so the precedence relation is well founded. When analyzing examples we usually work with the *kernel* of the ordering relation, i.e. the subset of $<$ consisting of the pairs (μ, μ') such that $\mu < \mu'$ and there is no μ'' such that $\mu < \mu'' < \mu'$. The full ordering relation can be recovered by taking the transitive closure of the kernel. We also elide non-final occurrences of δ, for example replacing $\mu < \delta < \mu'$ by $\mu < \mu'$. (This is analogous to our earlier convention for concatenating δ.)

A process P denotes a set (or "family") $\mathcal{P}(P)$ of pomsets. Each pomset $(T, <)$ determines a set of traces, the traces containing all action occurrences from T in a linear order consistent with the precedence relation, possibly allowing synchronization. A single trace α can be viewed as a special pomset whose elements are the action occurrences from α and whose ordering is linear. A pomset consists of a number of connected components, or *threads*.

Again we assume that the semantics of expressions is given, so that for an expression e, $\mathcal{P}(e)$ is a set of pairs of the form (T, v), where $v \in V$ and T is a pomset of evaluation actions.

We define $T_1; T_2 = T_1$ if $|T_1| = \omega$, otherwise $T_1; T_2$ is the ordering on $T_1 \cup T_2$ obtained by putting T_2 after T_1. $T_1 \| T_2$ is the disjoint union of T_1 and T_2 ordered with the disjoint union of the orderings from T_1 and T_2. We say that a pomset is *fair* iff it does not contain a pair of concurrent threads which eventually block on a pair of matching directions. For example, the pomset $\{a!0\delta_{b!}{}^{\omega}, a?0\delta_{b?}{}^{\omega}\}$ is unfair.

We define $T \preceq_h T'$ to mean that T' arises by choosing for each occurrence of $h?v$ (or $h!v$) in T a unique concurrent matching action occurrence $h!v$ (respectively, $h?v$) in T, and augmenting the ordering accordingly, with an arrow each way between the matched pairs. This can be formalized as a *synchronizing schedule* for channel h. There may be no such T', in which case T does not describe any traces which contribute to the behavior of **local** h **in** P, or there may be multiple such T', each corresponding to a sequence of synchronization choices. Given a pomset T' in which all visible actions on h are matched, we define $T' \backslash h$ to be the result of replacing all matching pairs by δ (i.e. enforcing synchronization), replacing every δ_X by $\delta_{X \backslash h}$, and eliding non-final δ actions.

Definition 2 *The pomset semantics of a process P, written $\mathcal{P}(P)$, is given* *compositionally by:*

$$
\begin{array}{rcl}
\mathcal{P}(\mathbf{skip}) & = & \{\{\delta\}\} \\
\mathcal{P}(x{:=}e) & = & \{T;\{x{:=}v\} \mid (T,v) \in \mathcal{P}(e)\} \\
\mathcal{P}(h?x) & = & \{\{h?v\} \mid v \in V\} \cup \{\{{\delta_{h?}}^{\omega}\}\} \\
\mathcal{P}(h!e) & = & \{T;\{h!v\} \mid (T,v) \in \mathcal{P}(e)\} \cup \{\{{\delta_{h!}}^{\omega}\}\} \\
\mathcal{P}(P_1;P_2) & = & \{T_1;T_2 \mid T_1 \in \mathcal{P}(P_1)\ \&\ T_2 \in \mathcal{P}(P_2)\} \\
\mathcal{P}(\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) & = & \mathcal{P}(b)_{\mathbf{true}};\mathcal{P}(P_1) \cup \mathcal{P}(b)_{\mathbf{false}};\mathcal{P}(P_2) \\
\mathcal{P}(\mathbf{while}\ b\ \mathbf{do}\ P) & = & (\mathcal{P}(b)_{\mathbf{true}};\mathcal{P}(P))^*;\mathcal{P}(b)_{\mathbf{false}}\ \cup\ (\mathcal{P}(b)_{\mathbf{true}};\mathcal{P}(P))^{\omega} \\
\mathcal{P}(G_1 \square G_2) & = & \{T \in \mathcal{P}(G_1) \cup \mathcal{P}(G_2) \mid T \cap \Delta^{\omega} = \{\}\} \cup \\
 & & \{\{{\delta_{X\cup Y}}^{\omega}\} \mid \{{\delta_X}^{\omega}\} \in \mathcal{P}(G_1)\ \&\ \{{\delta_Y}^{\omega}\} \in \mathcal{P}(G_2)\} \\
\mathcal{P}(P_1 \sqcap P_2) & = & \mathcal{P}(P_1)\ \cup\ \mathcal{P}(P_2) \\
\mathcal{P}(P_1 \| P_2) & = & \{T_1 \| T_2 \mid\ T_1 \in \mathcal{P}(P_1)\ \&\ T_2 \in \mathcal{P}(T_2)\ \&\ (T_1 \| T_2)\ \mathit{fair}\} \\
\mathcal{P}(\mathbf{local}\ h\ \mathbf{in}\ P) & = & \{T' \backslash h \mid T \in \mathcal{P}(P)\ \&\ T \preceq_h T'\}
\end{array}
$$

An Example

Let $\mathit{buff}_1(\mathit{in}, \mathit{mid})$ be **while true do** $(\mathit{in}?x; \mathit{mid}!x)$, which behaves like a 1-place buffer. Let $\mathit{buff}_1(\mathit{mid}, \mathit{out})$ be similarly defined. It is easy to prove using pomsets, or directly from the trace semantics, that with synchronized communication

$$\mathit{buff}_2(\mathit{in}, \mathit{out}) =_{\mathrm{def}} \mathbf{local}\ \mathit{mid}\ \mathbf{in}\ \mathit{buff}_1(\mathit{in}, \mathit{mid}) \| \mathit{buff}_1(\mathit{mid}, \mathit{out})$$

behaves like a 2-place buffer. One can also use the semantics to analyze a variety of alternative buffer-like constructs, such as

$$\mathbf{local}\ \mathit{mid}\ \mathbf{in}\ \mathit{buff}_2(\mathit{in}, \mathit{mid}) \| \mathit{buff}_2(\mathit{mid}, \mathit{out})$$

and one can validate a number of buffer laws along similar lines to those developed by Roscoe [27].

9 Recovering traces

The pomset semantics determines the trace semantics in a natural manner.

Definition 3 *The set of synchronous traces consistent with a pomset T, written $\mathcal{L}(T)$, consists of all traces which arise by fair interleaving the threads of T, possibly allowing synchronization.*

Equivalently, $\mathcal{L}(T)$ is the set of all linear orders on the multi-set T which extend the order of T, allowing for the possibility of synchronization.

Theorem 4 *For all processes P, $\mathcal{T}(P) = \bigcup\{\mathcal{L}(T) \mid T \in \mathcal{P}(P)\}$.*

Note the obvious but useful corollary:

Corollary 5 *For all P_1 and P_2, if $\mathcal{P}(P_1) = \mathcal{P}(P_2)$ then $\mathcal{T}(P_1) = \mathcal{T}(P_2)$.*

The converse is not necessarily true: the pomset family for $(a!0\|b!1)$ is

$$\{\{a!0, b!1\}, \{a!0, {\delta_{b!}}^{\omega}\}, \{b!1, {\delta_{a!}}^{\omega}\}, \{{\delta_{a!}}^{\omega}, {\delta_{b!}}^{\omega}\}\}$$

but the family

$$\{\{a!0\, b!1\}, \{b!1\, a!0\}, \{a!0, {\delta_{b!}}^{\omega}\}, \{b!1, {\delta_{a!}}^{\omega}\}, \{{\delta_{a!}}^{\omega}, {\delta_{b!}}^{\omega}\}\}$$

also determines the same trace set. Nevertheless, pomset semantics can serve as an alternative compositional approach to parallel program analysis, a potentially more succinct model of process behavior which might facilitate proofs. The pomset representation of the trace set of a process may allow us to avoid dealing explicitly with all interleavings, thus offering a chance to avoid a combinatorial explosion. Moreover, many laws of process equivalence hold for pomset semantics, and can be proven without dealing with fully expanded trace sets; such laws transfer immediately to trace semantics.

Our pomset semantics and even our trace semantics make certain distinctions which might seem more consistent with the "true concurrency" philosophy, despite the trace-theoretic rationale for our models. Indeed the so-called "interleaving law" does not hold. For example, $\mathcal{P}(a!0\|b!1)$ is the family given above, whereas $\mathcal{P}(a!0; b!1 \sqcap b!1; a!0)$ is the family

$$\{\{a!0\, b!1\}, \{b!1\, a!0\}, \{a!0\, {\delta_{b!}}^{\omega}\}, \{b!1\, {\delta_{a!}}^{\omega}\}, \{{\delta_{a!}}^{\omega}\}, \{{\delta_{b!}}^{\omega}\}\}$$

so that $a!0\|b!1$ is not pomset-equivalent to $(a!0; b!1) \sqcap (b!1; a!0)$. Indeed this distinction also holds in the trace semantics, since

$$\mathcal{T}(a!0\|b!1) \cap \Delta^{\omega} = {\delta_{a!}}^{\omega}\|{\delta_{b!}}^{\omega} = ({\delta_{a!}}^{*}\delta_{b!}{\delta_{b!}}^{*}\delta_{a!})^{\omega}$$

and

$$\mathcal{T}(a!0; b!1 \sqcap b!1; a!0) \cap \Delta^{\omega} = \{{\delta_{a!}}^{\omega}, {\delta_{b!}}^{\omega}\}.$$

This difference in trace sets can be explained intuitively, without appealing to considerations of true concurrency, since $a!0 \| b!1$ can be observed (if placed in a suitable environment) waiting repeatedly for action on one of the two channels, whereas the other process makes a non-deterministic choice and thereafter fixates on one particular channel. Note also that $a!0 \| b!1$ is also not trace- or pomset-equivalent to $(a!0 \to b!1) \square (b!1 \to a!0)$, since

$$\mathcal{T}((a!0 \to b!1) \square (b!1 \to a!0)) \cap \Delta^{\omega} = \{ {\delta_{\{a!, b!\}}}^{\omega} \},$$

so neither form of non-deterministic choice can be used to expand away a parallel composition.

10 Related work

Hoare's early "trace model" of CSP [17] interpreted a process as a non-empty, prefix-closed set of finite communication traces, recording only visible actions such as $h!v$ and $h?v$. Each trace represents a *partial* behavior of the process. Hoare's model did not treat infinite behaviors or fairness, and is mainly suitable for proving safety properties.

The failures semantics of CSP [10] modelled a process as a set of failures, each failure (α, X) consisting of a finite sequence α of communications and a "refusal set" X of directions, representing the potential to perform α then refuse to communicate along any direction in X. Our notion of trace subsumes failures: a process capable of (α, X) would have a trace $\alpha {\delta_Y}^{\omega}$, for some set Y disjoint from X. Our notion of trace is more general, allowing for instance traces of form $\alpha(\delta_A \delta_B)^{\omega}$ which cannot be represented in failure format. The extra generality is needed in order to cope properly with fair parallel composition. Our traces represent *entire* computations, so our trace sets are not prefix-closed. Again this is more than a philosophical difference: one cannot deduce the fair traces of a parallel process by looking at the prefixes of the traces of its constituent processes. The failures semantics and its later more refined extensions, all building on a prefix-closed trace set, were not designed with fairness in mind [27].

Older's Ph.D. thesis [19] provides a general framework capable of being instantiated to model several specialized forms of fairness in the synchronous setting, including weak and strong process fairness. She introduced generalized notions of fairness and blocking *modulo* a set of directions, and her models incorporated extensive book-keeping information to keep track of the sets of directions which were infinitely often enabled but not taken along traces, together with cleverly devised but complex closure conditions designed to achieve full abstraction [20, 9]. As Older comments, it is questionable if these fairness notions are useful and accurate abstractions of realistic schedulers, since their implementation requires meticulous attention to so much enabling information. Moreover, these forms of fairness tend to be sensitive to subtle nuances in the formulation of the operational semantics [1].

In contrast we assume a form of weakly fair execution, suitably adapted to deal reasonably with synchronization to ensure that two processes waiting to perform matching communications will not be ignored forever. This property would be guaranteed for instance by any round-robin scheduler which runs each process for an randomly chosen number of steps, and also uses a round-robin strategy to look for matching communications if the chosen process blocks while attempting input or output. This form of fairness is a simple variant of weak process fairness sensitive to the synchronization needs of processes, and we believe this is a reasonable abstraction from the behavior of realistic implementations. By adopting this fairness notion we avoid the need for excessive book-keeping: the traces themselves can be designed to carry the relevant information in their δ actions. Older's notion of being blocked but fair *modulo* a set X of directions corresponds to a trace β such that $\mathit{blocks}(\beta) \subseteq X$. It would be interesting to see if any of the more complex forms of fairness discussed by Older can be treated within our framework.

Our semantics for asynchronously communicating processes incorporates weak (process) fairness. Hennessy [18] gave an earlier treatment of a CCS-like language with weakly fair execution. Parrow [24] discusses various fairness notions for CCS-like processes.

Partial-order semantics of various kinds, such as Pratt-style pomsets [25, 26], Winskel's event structures [28][6], and Petri nets [23] have been widely used, with parallel composition interpreted as so-called "true concurrency" rather than interleaving. Our motivation in developing a pomset formulation is to obtain a more tractable methodology for dealing with trace sets, rather than having to deal explicitly with the results of interleaving. Pratt typically models a process as a set of *finite* pomsets, and concepts such as fairness, which only really crop up significantly when dealing with infinite behaviors, are not encountered in finite pomset models. Pratt-style pomsets are usually

[6]Event structures can be seen as pomsets equipped with a "conflict relation", although this characterization does not reflect their original development and subsequent usage.

taken to be order-isomorphism classes, and this works well if we care only about actions as abstract entities without data or imperative content. We do not do this: although the pomsets for processes $a!0$ and $b!0$ and even for $c!1$, are order-isomorphic, they do not behave identically in all contexts and we need to distinguish between them in order to reason accurately about safety and liveness properties concerning the values of program variables and the data transmitted during execution.

References

[1] K. R. Apt, N. Francez, and S. Katz, *Appraising fairness in languages for distributed programming*, Distributed Computing, 2(4):226-241 (1988).

[2] F. de Boer, J. Kok, C. Palamidessi, and J. Rutten, *The failure of failures in a paradigm for asynchronous concurrency*, Proc. 2nd International Conference on Concurrency Theory, CONCUR'91, Springer LNCS 527, pp. 111-126 (1991).

[3] S. Brookes, *Full abstraction for a shared-variable parallel language*, 8th IEEE Symposium on Logic in Computer Science, pp. 98-109 (1993). Extended version in: Information and Computation, vol 127, no. 2, Academic Press (June 1996).

[4] S. Brookes, *The Essence of Parallel Algol*, 11th IEEE Symposium on Logic in Computer Science, pp. 164-173 (July 1996).

[5] S. Brookes, *Idealized CSP: Combining Procedures with Communicating Processes*, 13th Conference on Mathematical Foundations of Programming Semantics (MFPS'97), Pittsburgh (March 1997).
Electronic Notes in Theoretical Computer Science 6, Elsevier Science (1997).
`http://www.elsevier.nl/locate/entcs/volume6.html`.

[6] S. Brookes, *On the Kahn Principle and Fair Networks*, 14th Conference on Mathematical Foundations of Programming Semantics, Queen Mary Westfield College, University of London, (May 1998).

[7] S. Brookes, *Communicating Parallel Processes*, Symposium in Celebration of the work of C.A.R. Hoare, Oxford University, September 1999. MacMillan Publishers (2000).

[8] S. Brookes, *Deconstructing CCS and CSP: Asynchronous Communication, Fairness and Full Abstraction*, 16th Conference on Mathematical Foundations of Programming Semantics (2000).

[9] S. Brookes and S. Older, *Full abstraction for strongly fair communicating processes*, 11th Conference on Mathematical Foundations of Programming Semantics, New Orleans, (March 1995).
`http://www.elsevier.nl/locate/entcs/volume1.html`

[10] S. Brookes, C. A. R. Hoare, and A. W. Roscoe, *A Theory of Communicating Sequential Processes*, JACM 31(3):560-599 (July 1984).

[11] S. Brookes, and A. W. Roscoe, *An improved failures model for CSP*, Seminar on concurrency, Springer-Verlag, LNCS 197 (1984).

[12] G. Costa and C. Stirling, *A fair calculus of communicating systems*, ACTA Informatica 21:417-441 (1984).

[13] G. Costa and C. Stirling, *Weak and strong fairness in CCS*, Technical Report CSR-16-85, University of Edinburgh (January 1985).

[14] N. Francez, **Fairness**, Springer-Verlag (1986).

[15] R. van Glabbeek, *The Linear Time – Branching Time Spectrum*, chapter 1 of **Handbook of Process Algebra**, J. A. Bergstra, A. Ponse and S. Smolka (eds), Elsevier (2001).

[16] C. A. R. Hoare, *Communicating Sequential Processes*, CACM 21:8, pp. 666-677 (1978).

[17] C. A. R. Hoare, *A Model for Communicating Sequential Processes*, Technical Monograph PRG-22, Programming Research Group, Oxford University (June 1981).

[18] M. Hennessy, *An algebraic theory of fair asynchronous communicating processes*, Theoretical Computer Science, 49:121-143 (1987).

[19] S. Older, *A Denotational Framework for Fair Communicating Processes*, Ph.D. thesis, Carnegie Mellon University. Technical report CMU-CS-96-204 (December 1996).

[20] S. Older, *A Framework for Fair Communicating Processes*, 13th Conference on Mathematical Foundations of Programming Semantics (March 1997).
`http://www.elsevier.nl/locate/entcs/volume6.html`.

[21] S. Owicki and L. Lamport, *Proving liveness properties of concurrent programs*, ACM TOPLAS, 4(3): 455-495 (July 1982).

[22] D. Park, *On the semantics of fair parallelism.* In D. Bjørner, editor, **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86 (1979), 504–526.

[23] C. A. Petri, *Concepts of Net Theory*, Symposium on Mathematical Foundations of Computer Science (September 1973).

[24] J. Parrow, *Fairness Properties in Process Algebras*, Ph. D. thesis, University of Uppsala (1985).

[25] V. Pratt, *On the Composition of Processes*, 9th ACM Symposium on Principles of Programming Languages, pp. 213-223 (1982).

[26] V. Pratt, *Modeling concurrency with partial orders*, International Journal on Parallel Processing, 15(1): 33–71 (1986).

[27] A. W. Roscoe, **The Theory and Practice of Concurrency**, Prentice-Hall, 1998.

[28] G. Winskel, *Events in Computation*, Ph. D. thesis, Edinburgh University (1980).

11 Appendix: asynchronous communication

Output actions are always enabled, and we assume that channels behave like unbounded queues; a process wishing to perform input from a channel must wait if the queue is empty. We only need δ_X when X is a set of input directions.

The set $\mathcal{AT}(P)$ of *asynchronous traces* of P is defined compositionally, exactly as for the synchronous traces but with modifications in the clauses for output, parallel composition, and local channels, which become:

$$\begin{array}{rcl}\mathcal{AT}(h!e) & = & \{\rho\, h!v \mid (\rho, v) \in \mathcal{T}(e)\} \\ \mathcal{AT}(P_1 \| P_2) & = & \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in \mathcal{AT}(P_1) \ \&\ \alpha_2 \in \mathcal{AT}(P_2)\} \\ \mathcal{AT}(\textbf{local}\ h\ \textbf{in}\ P) & = & \{\alpha \backslash h \mid \alpha \in \mathcal{AT}(P) \ \&\ \alpha\ \textit{local for}\ h\}\end{array}$$

Here we redefine $\alpha_1 \| \alpha_2$ to be the set of fair interleavings of α_1 with α_2, without allowing any synchronization. We say that α is *local for h* if the communications on h along α obey the queue discipline, and we redefine $\alpha \backslash h$ to replace all communications on h by δ and replace δ_X by $\delta_{X \backslash h}$.

The asynchronous operational semantics is obtained by making similar adjustments to the rules for output, parallel composition, and local channels, and including channel contents as part of the state. The operational notion of fair transition sequence is as before, except that the transition relation no longer includes synchronizing steps.

Again the denotationally characterized trace set coincides with the operationally characterized trace set, and again we have full abstraction with respect to communication behavior.

Theorem 6 *For every process P, $\mathcal{AT}(P)$ consists of the traces generated by the fair asynchronous transition sequences of P.*

Theorem 7 *Two processes P_1 and P_2 have the same asynchronous trace sets iff they have the same asynchronous communication behavior in all contexts.*

We can define an asynchronous pomset semantics $\mathcal{AP}(P)$, again adjusting the clauses for output, parallel composition and local channels:

$$\begin{array}{rcl}\mathcal{AP}(h!e) & = & \{T; \{h!v\} \mid (T, v) \in \mathcal{P}(e)\} \\ \mathcal{AP}(P_1 \| P_2) & = & \{T_1 \| T_2 \mid\ T_1 \in \mathcal{AP}(P_1) \ \&\ T_2 \in \mathcal{AP}(P_2)\} \\ \mathcal{AP}(\textbf{local}\ h\ \textbf{in}\ P) & = & \{T' \backslash h \mid T \in \mathcal{AP}(P) \ \&\ T \preceq_h T'\}\end{array}$$

We no longer need a side condition in the clause for $P_1 \| P_2$: every trace consistent with the disjoint union $T_1 \| T_2$ will represent a fair asynchronous behavior of the parallel process.

We redefine $T \preceq_h T'$ to mean that T' arises by choosing, for each *input* occurrence $h?v$ in T an output occurrence $h!v$ in T which *justifies* it, all choices respecting the precedence order of T and the queue discipline of the channel, and augmenting the ordering so that each input is preceded by its justifying output. For a given T and h, there may be no such T', in which case T does not describe any traces which are local for h, or there may be multiple such T', each corresponding to a different sequence of scheduling choices. Given a pomset T' in which all inputs on h are justified in this manner, we define $T' \backslash h$ to replace each communication on h by δ, replace δ_X by $\delta_{X \backslash h}$, and elide non-final δ actions.

Definition 4 *The set of asynchronous traces consistent with a pomset T, written $\mathcal{AL}(T)$, is the set of all traces which arise by fair interleaving the threads of T.*

Again the asynchronous traces of a process can be recovered from its pomset semantics:

Theorem 8 *For all processes P, $\mathcal{AT}(P) = \bigcup\{\mathcal{AL}(T) \mid T \in \mathcal{AP}(P)\}$.*

One can show using the pomset semantics that if we assume asynchronous communication the process

$$\textbf{local } \textit{mid} \textbf{ in } \textit{buff}_1(\textit{in}, \textit{mid}) \| \textit{buff}_1(\textit{mid}, \textit{out})$$

behaves like an *unbounded* buffer, instead of the 2-place buffer which described its behavior under synchronous communication.

The following laws hold for *asynchronous* trace semantics, and are the asynchronous analogues of the first two laws given earlier for synchronous communication. The third law does not hold, because of the assumption that output is always enabled.

Theorem 9 *The asynchronous trace semantics validates the following laws of equivalence:*

1. $\textbf{local } h \textbf{ in } (h?x; P) \| (h!v; Q) \| R \ = \ \textbf{local } h \textbf{ in } (x{:=}v; P) \| Q \| R$ *provided* $h \notin \texttt{chans}(R)$.
2. $\textbf{local } h \textbf{ in } (h?x; P) \| (Q_1; Q_2) \ = \ Q_1; \textbf{local } h \textbf{ in } (h?x; P) \| Q_2$ *provided* $h \notin \texttt{chans}(Q_1)$.