Graphical Models of Separation Logic

C. A. R. HOARE [a], Ian WEHRMAN [b,1] and Peter W. O'HEARN [c]
[a] *Microsoft Research, Cambridge, UK*
[b] *The University of Texas at Austin, USA*
[c] *Queen Mary University London, UK*

Abstract. Labelled graphs are used to model control and data flow among events occurring in the execution of a (possibly concurrent) program. Data flow is a unifying concept that covers both access to memory and communication along channels; it covers many variations including weakly consistent memory, re-ordered execution, and communication channels that are multiplexed, buffered, or otherwise unreliable. Nevertheless, the laws of Hoare and Jones correctness reasoning remain valid when interpreted in this general model. The key is use of the same language and logic for the specification of programs as for description of the behavior of programs. We make no attempt to lift the level of abstraction above that of the atomic events involved in program execution.

Keywords. concurrency, formal semantics, separation logic

1. Introduction

In this paper, we present a trace semantics based on graphs: nodes represent the events of a program's execution, and edges represent dependencies among the events. The style is reminiscent of partially ordered models [12,16], though we do not generally require properties like transitivity or acyclicity. Concurrency and sequentiality are defined in Section 2 using variations on separating conjunctions: whereas the conjunction in the original separation logic partitions addresses in a heap [11,14], the conjunctions here partition events in a trace. We show graphical models of simple programming primitives in Section 3, and of more advanced primitives in Section 7. The model has pleasant algebraic properties, which are shown with surprisingly simple proofs. We present a number of theorems about the generic model, including the soundness of the laws of Hoare logic [4] in Section 5 and the Jones rely/guarantee calculus [6] in Section 6. The paper is liberally illustrated by diagrams.

2. Traces and Separation

A *directed graph* is a pair of sets (EV, AR), where EV is a set of nodes and AR is a set of (directed) arrows linking the nodes. The objects of EV represent occurrences

[1]Corresponding Author: The University of Texas at Austin, Department of Computer Sciences, 1 University Station, C0500, Austin, TX 78712-0233, USA; E-mail: iwehrman@cs.utexas.edu.

of atomic events recorded in a trace of program execution; the objects of AR represent control or data flows that occur between events. We assume that the sets EV and AR contain all possible events in the execution of a program.

A *labelled graph* also has a labelling function on its events and arrows. The theorems in this paper only make use of event labels, but our examples also describe labels on arrows. We write $label(e) = a$ to mean that a is the atomic action of the programming language (e.g., $x := x + 12$) whose execution is recorded as e.

A *trace* is a subset of EV that records the execution of a component of the program. We will identify each command of our programming language with the set of traces of all its possible executions, in any possible environment. An *atomic* command is one whose execution gives rise to only a single event, and this event has label ℓ:

Definition 1. $[\ell] =_{\mathbf{def}} \{tr \mid \exists r \in tr.tr = \{r\} \ \& \ label(r) = \ell\}$.

We can define the semantics of a program as the set of its possible traces, as in the CSP traces model [15]. For example, the $[\ell]$ is the set of traces associated with a label (thought of as a program) ℓ.

If p and q are events, we write $p \longrightarrow q$ to indicate the existence of an arrow between them. As usual, we write $\longleftarrow$ and $\overset{+}{\longrightarrow}$ for the inverse and transitive closure of the $\longrightarrow$ relation, respectively. Additionally, we write $\longleftrightarrow$ for $\longrightarrow \cup \longleftarrow$ and $\overset{+}{\longleftrightarrow}$ for $\overset{+}{\longrightarrow} \cup \overset{+}{\longleftarrow}$. When we wish to restrict a relation to a particular set of events $t \subseteq EV$, we write, for example, $\overset{+}{\longrightarrow}_t$. For trace tq, we write $p \longrightarrow tq$ when, for some event $q \in tq$, $p \longrightarrow q$; and similarly $tp \longrightarrow tq$ when for some $p \in tp$ and $q \in tq$, $p \longrightarrow q$.

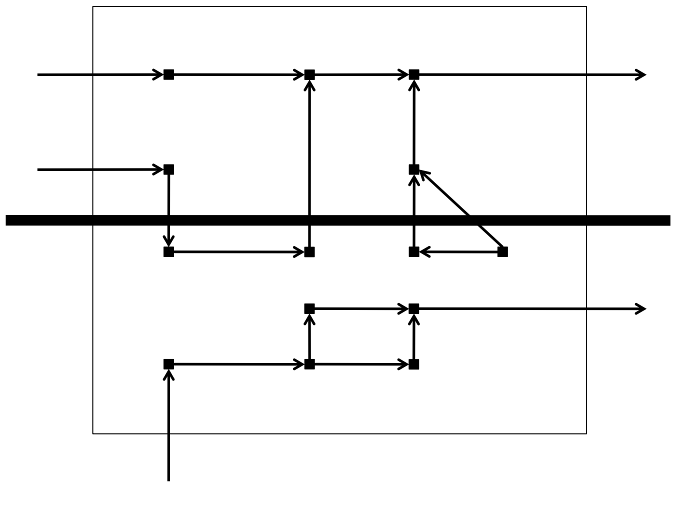

Figure 1. A trace separated with $(*)$

Let $P * Q$ be the structured command that denotes the concurrent execution of components P and Q. Obviously no event is simultaneously a part of the execution of both these commands. Furthermore, every event in the execution of $P * Q$ is in the execution of at least one of P and Q. Therefore, the most general form of a trace tr of $P * Q$ is the disjoint union of some trace tp of P with some trace tq of Q:

$$tr = tp * tq \equiv_{\mathbf{def}} tr = tp \cup tq \ \& \ tp \cap tq = \emptyset.$$

An example of a trace separated with $(*)$ is shown in Figure 1. The trace separator $(*)$ is a partial function on traces; it can be lifted to a total function on sets of traces in the usual way:

Definition 2. $P * Q =_{\mathbf{def}} \{tr \mid tr = tp * tq \ \& \ tp \in P \ \& \ tq \in Q\}$.

We call the $(*)$ function on trace sets the *concurrent separator*. In words, a trace is a model of $P * Q$ exactly when it can be split into two disjoint parts, one of which is a trace of P and the other a trace of Q. This definition can be implemented by running P and Q concurrently, as separate threads or processes in the same or in different computers. There is no restriction on the arrows which communicate between events of P and events of Q. The two threads may communicate freely with each other, e.g. through shared memory or channels.

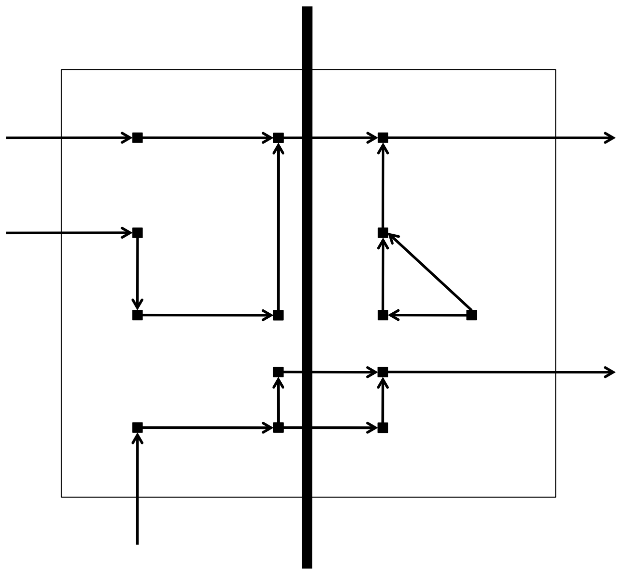

Figure 2. A trace separated with $(;)$

A stronger notion is *sequential separation*. Informally, a trace tr may be split into a sequential separation $tp\,;tq$ when there is no arrow from an event of tq to an event of tp:

$$tr = tp\,;tq \equiv_{\mathbf{def}} tr = (tp * tq) \ \& \ \neg(tp \leftarrow tq).$$

An example of a trace separated with $(;)$ is shown in Figure 2. Again, the trace separator $(;)$ is lifted pointwise to a total function on trace sets:

Definition 3. $P\,;Q =_{\mathbf{def}} \{tr \mid tr = (tp\,;tq) \ \& \ tp \in P \ \& \ tq \in Q\}$.

This definition expresses an essential property of sequential composition. It allows implementations to optimise a program by interleaving events of the first trace with events of the second; the events can even be executed concurrently, if they do not violate the dependency condition in the definition. We have thus defined what is sometimes called a “weak sequential composition” in concurrency theory, and it has the usual anomalous consequence.

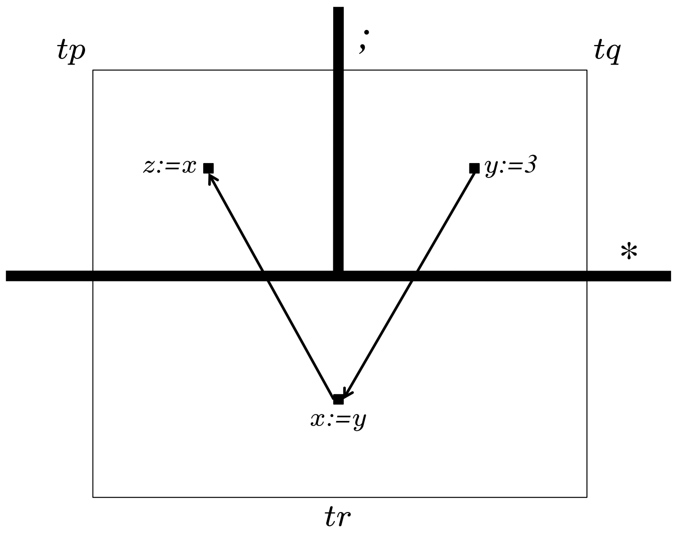

Figure 3. Backward dependency anomaly: finally $z = 3$

Consider the trace $(tp\,;tq) * tr$ in Figure 3. In our definition, an event in tp can depend on an event of tq through a chain of dependencies in the concurrent thread tr. The events in Figure 3 are labeled with assignment statements to show how such a trace might arise. As a result of standard optimisations, this apparently paradoxical data flow—in which z may take the value 3—occurs in standard computers of the present day. Thus, our theory faithfully represents the (problematic) features of the real world. It is therefore surprising and encouraging that the model validates all the familiar proof rules of sequential and concurrent reasoning about programs [4,6], as we show in Sections 5 and 6.

A yet stronger notion is *parallel separation*, in which distinct processes have no interdependencies among them. The definition allows (but does not compel) an implementation to run the whole of each process concurrently with the others, perhaps in its own partition of memory:

$$tr = tp\,||\,tq \;\equiv_{\mathbf{def}}\; tr = (tp\,;tq) \;\&\; tr = (tq\,;tp).$$

Again, we lift the definition of trace sets, and call the resulting operation a *parallel separator*:

Definition 4. $P\,||\,Q \;=_{\mathbf{def}} \{tr \mid tr = (tp\,||\,tq) \;\&\; tp \in P \;\&\; tq \in Q\}.$

In models based on sequential traces [15] this definition is given in terms of arbitrary interleaving. Our definition seems easier to reason with.

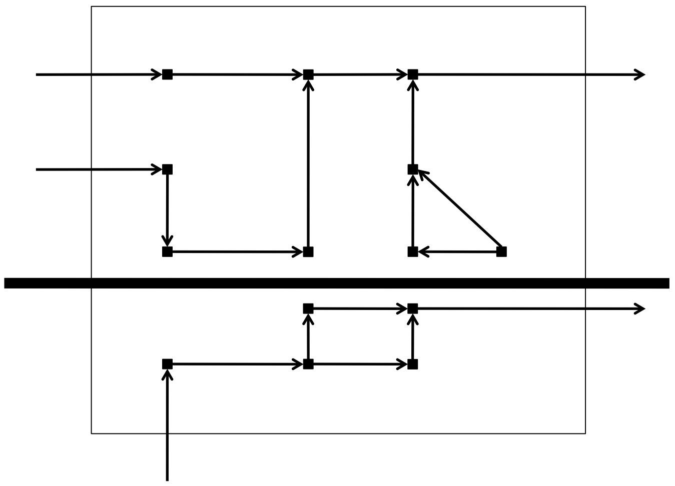

Figure 4. A trace separated with $(||)$

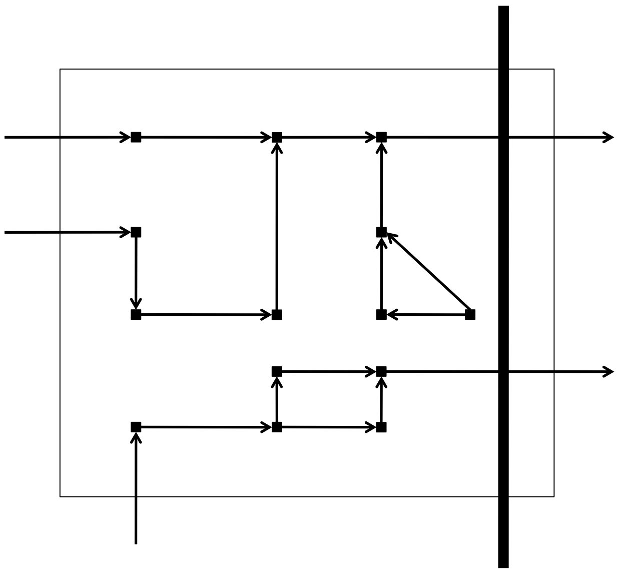

Figure 5. A trace separated with $([])$

The fourth and strongest notion of separation holds between traces when at least one is empty:

$$tr = tp \, [] \, tq \;\; \equiv_{\mathbf{def}} \;\; tr = tp \cup tq \;\; \& \;\; (tp = \emptyset \vee tq = \emptyset).$$

We can represent nondeterministic choice by lifting this operator to sets of traces. The each execution of process $P \, [] \, Q$ behaves either as an execution of P or as an execution of Q:

Definition 5. $P \, [] \, Q \;\; =_{\mathbf{def}} \;\; \{tr \mid tr = tp \, [] \, tq \;\; \& \;\; tp \in P \;\; \& \;\; tq \in Q\}.$

We shall not deal in further detail with the operators $(||)$ and $([])$. It is sufficient to note that they have the same algebraic properties as the concurrent separator $(*)$.

3. Graphical Models of Simple Programming Primitives

This section shows graphical models of some simple primitive operations of some programming languages. Our models are presented informally with pictures of traces. Events in the traces are depicted by boxes that surround their labels. Dependencies are depicted by arrows. A plurality of arrows with identical labelling is indicated by an diagonal slash.

The orientation of the arrows in the following pictures is informally meaningful: vertical arrows indicate dependencies between processes (i.e., across separators), and horizontal arrows indicate dependencies within a process (i.e., only across the sequential separator (;)).

Arrows that represent data dependencies are labeled with this data and the resource across which it flows. For horizontal arrows, the resource is above and the data below; for vertical arrows, the resource is to the left and the data is to the right. We use the letters x, y, z for variables, and a, b, c for constant values (including names).

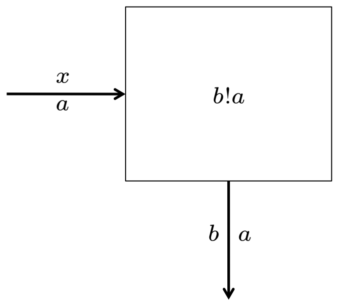

Figure 6. $b!x$: Output on channel b

The first primitive is shown in Figure 6. It shows a trace of the command $b!x$, an output of the value of variable x on communication channel b. The trace contains a single event, depicted as a box. The event has a single incoming arrow, which indicates the incoming flow value a of variable x, and a single outgoing arrow, which indicates the outgoing flow of value a on channel b. The value a flows in across x from within the process, and so is represented by a horizontal arrow. The value a flows on channel b outside of the process, and so is represented by a vertical arrow.

Similarly, Figure 7 shows a trace of the command $b?x$, an input from communication channel b into variable x. The trace contains a single event, with a single input arrow that represents the input of value a from channel b. There may be many output arrows, which represent thread-local accesses of the value a via variable x. The value a flows out across x within the process, and so is represented by a horizontal arrow.

Figure 8 shows a trace of a blocking input command, $b?x.P(x)$, as in CCS. Its first action is to input a value on channel b and give it name x, which is local to the command $P(x)$. We will use normal data flow arrows to carry the value that has been input to

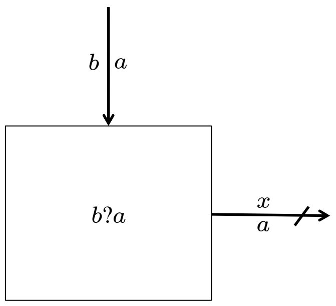

Figure 7. $b?x$: Input from channel b

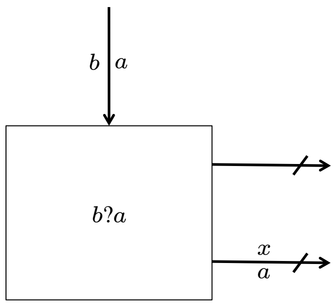

Figure 8. $b?x.P(x)$: Blocking input from channel b

the places in $P(x)$ where it is used. That is the function of the collection of arrows at the bottom right of Figure 8. But CCS also ensures that no action of $P(x)$ is executed before the input. That is the function of the collection of control arrows at the top right of Figure 8.

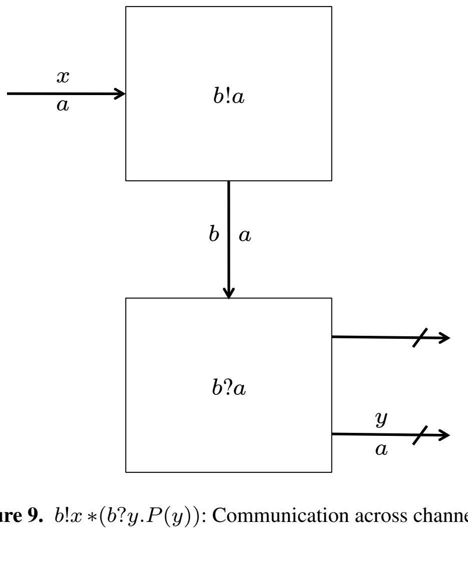

Figure 9. $b!x * (b?y.P(y))$: Communication across channel b

Figure 9 shows communication across channel b, as in CSP [15], by concurrent composition of an input with an output command: $b!x * (b?y.P(y))$. According to the definition of $(*)$, the events of this command include all and only the events of $b!x$ and $b?y.P(y)$. The value a from variable x is communicated across channel b, and then accessed via variable y by process P. Note that the communication between threads via channel b is represented by a vertical arrow.

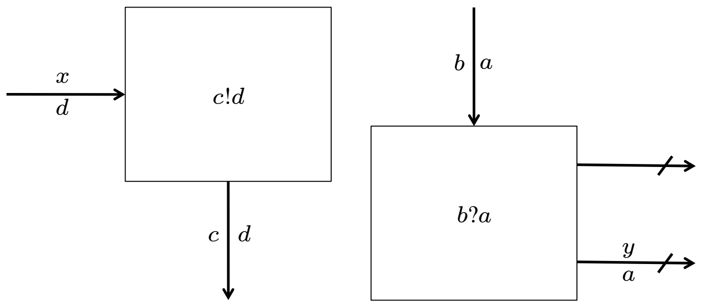

Figure 10. $c!x * (b?y.P(y))$: Interleaved output and input events

Figure 10 also shows a concurrent composition of input and output commands, $c!x * (b?y.P(y))$, with $c \neq b$. Because the input and output commands employ different channels, no direct communication occurs—the data from the output commands flows to the environment, and the data from the input command flows from the environment.

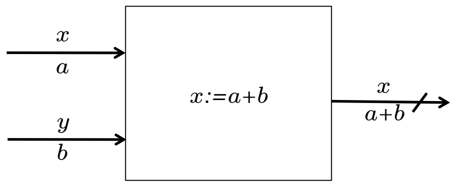

Figure 11. $x := x + y$: Variable assignment

Figure 11 shows an assignment command, $x := x + y$, which for the sake of example we will treat as atomic. The trace has only a single event, labeled with the variable x and the value assigned to it. There is a single input arrow for each variable in the expression on the right-hand-side of the assignment statement; in this case, one for variable x carrying value a, and one for variable y carrying value b. There are output arrows labeled with resource x and data value $a + b$, indicating the value of the variable after the assignment. All arrows are horizontal, indicating that variables should only be accessed locally, within a process.

Finally, Figure 12 shows a program that combines that variable assignment, input, output and blocking input:

$$x := x + y \ ; (c?z \ . \ (d!(x - z) * d?y)) \ .$$

After assigning $x + y$ to variable x, a value for z is read from channel c (from the environment), and $x - z$ is communicated across channel d, which then becomes locally available via variable y.

Graphical models of additional, advanced commands are given in Section 7.

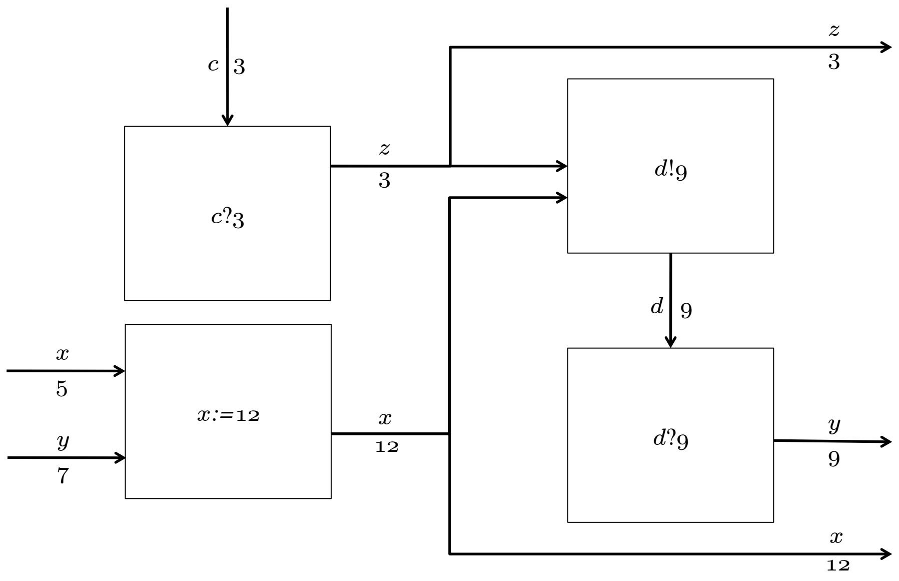

Figure 12. $x := x + y$; $(c?z.(d!(x - z) * d?y))$: A combined example

4. Program algebra

To facilitate comparison with (original) separation logic [11,14,10], in this section we use notations of propositional logic to represent the corresponding operations on sets. For example, $P \vee Q$ represents a nondeterministic choice between P and Q, and can easily be executed by executing either one of them. $P \wedge Q$ specifies a conjunction of two desirable properties of a program; it may be false, and therefore cannot in general be executed. Nevertheless, conjunction is an essential connective for modular specification of programs. We define the special predicate *false* as the empty set of traces (which cannot be implemented), and *skip* as the set that contains only the empty trace (which is easily implemented by doing nothing). For predicate P and trace tp, we write $P(tp)$ to indicate $tp \in P$, and say in this case that tp satisfies P.

The inclusion $P \subseteq Q$ can be interpreted as a refinement ordering:

Definition 6. $P \models Q \equiv_{\mathbf{def}} P \subseteq Q$.

It means that every trace observed of P is also a possible trace of Q; hence we often prove $P \models Q$ by showing, for an arbitrary trace t, that $P(t) \Rightarrow Q(t)$. It follows that if P is implementable, then so is Q; and if P is incorrect, then so is Q. As result, Q can be implemented by running P instead. For example, the following theorem shows that the four separators form a chain of refinements:

Theorem 1 (Refinement). $P \,[]\, Q \models P \,||\, Q \models P \,;\, Q \models P * Q$.

We show that the (;) operator has all the familiar properties: it is associative, distributes across disjunction, and is monotonic. *skip* is its unit and *false* is a zero. The $(*)$ operator satisfies all these laws, as well as commutativity.

In the sequel, note that in reasoning about a partially defined term like $tp\,;tq$, we adopt the convention that mention of a term implies that it is defined. As a result, all our equalities are strong equalities.

Lemma 1. $t = tp\,;(tq\,;tr) \;\Leftrightarrow\; t = (tp\,;tq)\,;tr$

Proof.

$$\begin{array}{ll}
 & t = tp\,;(tq\,;tr) \\
\Leftrightarrow & \{\text{definition of } (;)\} \\
 & t = tp \cup (tq\,;tr) \;\&\; tp \cap (tq\,;tr) = \emptyset \;\& \\
 & \neg(tp \leftarrow (tq\,;tr)) \\
\Leftrightarrow & \{\text{definition of } (;)\} \\
 & t = tp \cup tq \cup tr \;\&\; tp \cap (tq \cup tr) = tq \cap tr = \emptyset \;\& \\
 & \neg(tp \leftarrow (tq \cup tr)) \;\&\; \neg(tq \leftarrow tr) \\
\Leftrightarrow & \{\text{set theory}\} \\
 & t = tp \cup tq \cup tr \;\&\; tp \cap tq = tp \cap tr = tq \cap tr = \emptyset \;\& \\
 & \neg(tp \leftarrow tq) \;\&\; \neg(tp \leftarrow tr) \;\&\; \neg(tq \leftarrow tr) \\
\Leftrightarrow & \{\text{definition of } (;)\} \\
 & t = (tp\,;tq) \cup tr \;\&\; (tp \cup tq) \cap tr = \emptyset \;\&\; \neg(tq \leftarrow tr) \;\&\; \neg(tp \leftarrow tr) \\
\Leftrightarrow & \{\text{definition of } (;)\} \\
 & t = (tp\,;tq)\,;tr
\end{array}$$

□

Theorem 2 (Associativity). $P\,;(Q\,;R) = (P\,;Q)\,;R$

Proof.

$$\begin{array}{ll}
 & (P\,;(Q\,;R))(t) \\
\Leftrightarrow & \{\text{definition of } (;)\} \\
 & \exists tp, tq, tr : P(tp) \;\&\; Q(tq) \;\&\; R(tr) \;\&\; t = tp\,;(tq\,;tr) \\
\Leftrightarrow & \{\text{Lemma 1}\} \\
 & \exists tp, tq, tr : P(tp) \;\&\; Q(tq) \;\&\; R(tr) \;\&\; t = (tp\,;tq)\,;tr \\
\Leftrightarrow & \{\text{definition of } (;)\} \\
 & ((P\,;Q)\,;R)(t)
\end{array}$$

□

Lemma 2. $t = tp * tq \;\Leftrightarrow\; t = tq * tp$

Proof.

$$\begin{array}{ll}
 & t = tp * tq \\
\Leftrightarrow & \{\text{definition of } (*)\} \\
 & t = tp \cup tq \;\&\; tp \cap tq = \emptyset \\
\Leftrightarrow & \{\text{set theory}\} \\
 & t = tq \cup tp \;\&\; tq \cap tp = \emptyset \\
\Leftrightarrow & \{\text{definition of } (*)\} \\
 & t = tq * tp
\end{array}$$

□

Theorem 3 (Commutativity). $P * Q = Q * P$

Proof.

$$\begin{array}{ll}
 & (P * Q)(t) \\
\Leftrightarrow & \{\text{definition of } (*)\} \\
 & \exists tp, tq : P(tp) \;\&\; Q(tq) \;\&\; t = tp * tq \\
\Leftrightarrow & \{\text{Lemma 2}\}
\end{array}$$

$\exists tp, tq : P(tp) \ \& \ Q(tq) \ \& \ t = tq * tp$
$\Leftrightarrow$ {definition of $(*)$}
$(Q * P)(t)$

□

Theorem 4 (Distributivity)**.**

1. $P\,;(Q \vee R) = (P\,;Q) \vee (P\,;R)$
2. $(Q \vee R)\,;P = (Q\,;P) \vee (R\,;P)$

Proof. We show the first part; the second is similar.

$(P\,;(Q \vee R))(t)$
$\Leftrightarrow$ {definition of $(;)$}
$\exists tp, t2 : P(tp) \ \& \ (Q \vee R)(t2) \ \& \ t = tp\,;t2$
$\Leftrightarrow$ {definition of $(\vee)$}
$\exists tp, t2 : P(tp) \ \& \ (Q(t2)$ or $R(t2)) \ \& \ t = tp\,;t2$
$\Leftrightarrow$ {propositional calculus}
$\exists tp, t2 : (P(tp) \ \& \ Q(t2) \ \& \ t = tp\,;t2)$ or
$(P(tp) \ \& \ R(t2) \ \& \ t = tp\,;t2)$
$\Leftrightarrow$ {definition of $(;)$}
$((P\,;Q) \vee (P\,;R))(t)$

□

Theorem 5 (Monotonicity)**.**

1. $P \models Q \ \Rightarrow \ (P * R) \models (Q * R)$
2. $P \models Q \ \Rightarrow \ (R * P) \models (R * Q)$

Proof. $(P * R)(t)$
$\Rightarrow$ {definition of $(*)$}
$\exists tp, tr : P(tp) \ \& \ R(tr) \ \& \ t = tp * tr$
$\Rightarrow$ {assumption}
$\exists tp, tr : Q(tp) \ \& \ R(tr) \ \& \ t = tp * tr$
$\Rightarrow$ {definition of $(*)$}
$(Q * R)(t)$

□

Theorem 6 (Unit, Zero)**.**

1. $(P\,;skip) = (skip\,;P) = P$
2. $P\,;false = false\,;P = false$

Proof. 1. $(P\,;skip)(t)$
$\Leftrightarrow$ { definition of $(;)$ and $skip$}
$\exists tp, tq : P(tp) \ \& \ skip(tq) \ \& \ t = tp\,;tq$
$\Leftrightarrow$ {$skip(tq)$ implies $tq = \emptyset$}
$P(t)$

2.
$$\begin{array}{ll} & P\,;\mathit{false} \\ = & \{\text{definition of } (;)\} \\ & \{t \mid \exists tp \in P : \exists tq \in \mathit{false} : t = tp\,;tq\} \\ = & \{\text{set theory, } \mathit{false} = \emptyset\} \\ & \emptyset \\ = & \{\text{definition of } \mathit{false}\} \\ & \mathit{false} \end{array}$$

□

Let $P^0 =_{\mathbf{def}} \mathit{skip}$ and $P^{n+1} =_{\mathbf{def}} P^n\,;P$, for all natural numbers n. Finite iteration of a program P, written P^∞, is then defined as follows:

Definition 7 (Iteration)**.** $P^\infty =_{\mathbf{def}} \bigcup_n P^n$.

By Kleene's fixpoint theorem, P^∞ is the least fixpoint of the function $\lambda X.(X;P)$. Trace sets are thus a model of Kleene algebra [7,5], where the additive and multiplicative operations are given respectively by disjunction and sequential composition (or, equally well, by disjunction and concurrent composition).

Lemma 3. *For all* $n \geq 0$*,* $P\,;P^n = P^{n+1}$

Proof. By induction on n. If n = 0:

$$\begin{array}{ll} & P\,;P^0 \\ = & \{P^0 = \mathit{skip}\} \\ & P\,;\mathit{skip} \\ = & \{\mathit{skip} \text{ is unit, Theorem 6}\} \\ & \mathit{skip}\,;P \\ = & \{\text{definition of } P^n\} \\ & P^1 \end{array}$$

Otherwise:

$$\begin{array}{ll} & P\,;P^{n+1} \\ = & \{\text{definition of } P^n\} \\ & P\,;(P^n\,;P) \\ = & \{\text{associativity of } (;)\} \\ & (P\,;P^n)\,;P \\ = & \{\text{induction on } n\} \\ & P^{n+1}\,;P \\ = & \{\text{definition of } P^n\} \\ & P^{n+2} \end{array}$$

□

Theorem 7 (Unfold)**.**

1. $(\mathit{skip} \vee (P^\infty\,;P) \models P^\infty$
2. $(\mathit{skip} \vee (P\,;P^\infty)) \models P^\infty$

Proof. 1. $\mathit{skip} \models P^0 \models P$, and

$$\begin{array}{ll} & (P^\infty \,; P)(t) \\ \Rightarrow & \{\text{definition of } (*) \text{ and } (;)\} \\ & \exists n, tp', tp : P^n(tp') \;\;\&\;\; P(tp) \;\;\&\;\; t = tp' \,; tp \\ \Rightarrow & \{\text{definition of } (*)\} \\ & \exists n : P^n(t) \\ \Rightarrow & \{\text{definition of } P^\infty\} \\ & P^\infty(t) \end{array}$$

2. From part 1) and Lemma 3.

□

Theorem 8 (Induction)**.**

1. $(Q \vee (P\,; R)) \models R) \;\Rightarrow (P^\infty \,; Q) \models R$
2. $((Q \vee (R\,; P)) \models R) \;\Rightarrow (Q\,; P^\infty) \models R$

Proof. We show by induction on n that, for all n, $(P^n \,; Q) \models R$. If $n = 0$ then:

$$\begin{array}{ll} & P^0 \,; Q \\ \models & \{P^0 = \mathit{skip}, \text{Theorem 6}\} \\ & Q \\ \models & \{\text{propositional logic}\} \\ & Q \vee (P\,; R) \\ \models & \{\text{assumption}\} \\ & R. \end{array}$$

Otherwise,

$$\begin{array}{ll} & P^{n+1} \,; Q \\ \models & \{\text{Lemma 3}\} \\ & (P\,; P^n)\,; Q \\ \models & \{\text{associativity of } (;)\} \\ & P\,;(P^n \,; Q) \\ \models & \{\text{induction on } n\} \\ & P\,; R \\ \models & \{\text{propositional logic}\} \\ & Q \vee (P\,; R) \\ \models & \{\text{assumption}\} \\ & R. \end{array}$$

□

The definitions of $P * Q$ and $P\,; Q$ are instances of general model theories of separation logic and bunched logic [13,3,2], and thus inherit all of the general properties implied by these theories. In the sequel we describe properties of the model that concern *interaction* between the connectives.

Lemma 4. $t = (tp * tq)\,;(tr * ts) \Rightarrow t = (tp\,; tr) *(tq\,; ts)$

Proof. $t = (tp * tq) ;(tr * ts)$

$\Rightarrow$ {definition of $(;)$}

$t = (tp * tq) \cup (tr * ts)$ & $(tp * tq) \cap (tr * ts) = \emptyset$ &
$\neg((tp * tq) \leftarrow (tr * ts))$

$\Rightarrow$ {definition of $(*)$}

$t = tp \cup tq \cup tr \cup ts$ & tp, tq, tr, ts are pairwise disjoint &
$\neg((tp \cup tq) \leftarrow (tr \cup ts))$

$\Rightarrow$ {set theory}

$t = tp \cup tq \cup tr \cup ts$ & tp, tq, tr, ts are pairwise disjoint &
$\neg(tp \leftarrow tr)$ & $\neg(tq \leftarrow ts)$

$\Rightarrow$ {definition of $(;)$}

$t = (tp ; tr) \cup (tq ; ts)$ & tp, tq, tr, ts are pairwise disjoint

$\Rightarrow$ {definition of $(*)$}

$t = (tp ; tr) *(tq ; ts)$

□

Theorem 9 (Exchange)**.** $(P * Q) ;(P' * Q') \models (P ; P') *(Q ; Q')$

Proof. $((P * Q) ;(P' * Q'))(t)$

$\Rightarrow$ {definition of $(*)$ and $(;)$}

$\exists tp, tq, tp', tq' : P(tp)$ & $Q(tq)$ & $P'(tp')$ & $Q'(tq')$ &
$t = (tp * tq) ;(tp' * tq')$

$\Rightarrow$ {Lemma 4}

$\exists tp, tq, tp', tq' : P(tp)$ & $Q(tq)$ & $P'(tp')$ & $Q'(tq')$ &
$t = (tp ; tp') *(tq ; tq')$

$\Rightarrow$ {definition of $(*)$ and $(;)$}

$((P ; P') *(Q ; Q'))(t)$

□

Intuitively, the exchange law holds because the antecedent restricts dependencies of P' and Q' on both P and Q, whereas the consequent only restricts dependencies of Q' on Q and of P' on P. The laws below are easily derived from the exchange law by substituting *skip* for some of the operands.

Corollary 1.

1. $P ;(Q * R) \models (P ; Q) * R$
2. $(P * Q) ; R \models P *(Q ; R)$

Proof. We show $P ;(Q * R) \models (P ; Q) * R$ here; the other proof is similar.

$P ;(Q * R)$

$\models$ {*skip* is a unit of $(*)$, by Theorem 6}

$(P * skip) ;(Q * R)$

$\models$ {Theorem 9, the Exchange law}

$(P ; Q) *(skip ; R)$

$\models$ {*skip* is a unit of (;), by Theorem 6}

$(P ; Q) * R.$

□

5. Hoare Logic

The familiar assertional triple $P\ \{\ Q\ \}\ R$ over predicates P,Q and R is defined as follows:

Definition 8. $P\ \{\ Q\ \}\ R \equiv_{\mathbf{def}} (P\,;Q) \models R.$

Note that our assertion language is the same as our programming language. Like programs, our assertions P and R describe the entire history of execution. Thus the triple defined above states that if P is a description of what has happened just before Q starts, then R describes what has happened when Q has finished. The more familiar kind of single-state assertions describe the history abstractly as the set of traces that end in a state satisfying the assertion. The usual axioms of assertional reasoning [4] are easily proved sound.

Theorem 10.

1. $P\ \{\ Q\ \}\ R\ \ \&\ \ P\ \{\ Q\ \}\ R' \ \Rightarrow\ P\ \{\ Q\ \}\ R \wedge R'$
2. $P\ \{\ Q\ \}\ R\ \ \&\ \ P'\ \{\ Q\ \}\ R \ \Rightarrow\ P \vee P'\ \{\ Q\ \}\ R$
3. $P\ \{\ Q\ \}\ S\ \ \&\ \ S\ \{\ Q'\ \}\ R \ \Rightarrow\ P\ \{\ Q;Q'\ \}\ R$
4. $P\ \{\ Q\ \}\ R\ \ \&\ \ P\ \{\ Q'\ \}\ R \ \Rightarrow\ P\ \{\ Q \vee Q'\ \}\ R$
5. $P\ \{\ Q\ \}\ R\ \ \&\ \ P'\ \{\ Q'\ \}\ R' \ \Rightarrow\ P * P'\ \{\ Q * Q'\ \}\ R * R'$
6. $P\ \{\ Q\ \}\ R \ \Rightarrow\ F * P\ \{\ Q\ \}\ F * R$

Proof. 1. We show $(P\,;Q) \models (R \wedge R')$:

$$\begin{array}{ll} & P\,;Q \\ \models & \{\text{assumption}\} \\ & R \text{ and } R' \\ \models & \{\text{definition of } (\wedge)\} \\ & R \wedge R' \end{array}$$

2. We show $((P \vee P')\,;Q) \models R$:

$$\begin{array}{ll} & (P \vee P')\,;Q \\ \models & \{\text{Theorem 4, distributivity}\} \\ & (P\,;Q) \vee (P'\,;Q) \\ \models & \{\text{assumption}\} \\ & R \vee R \\ \models & \{\text{propositional logic}\} \\ & R \end{array}$$

3. We show $(P\,;(Q\,;Q')) \models R$:

$$\begin{array}{ll} & P\,;(Q\,;Q') \\ \models & \{\text{definition of } (;)\} \\ & (P\,;Q)\,;Q' \\ \models & \{\text{assumption, monotonicity of } (;) \text{ by Theorem 5}\} \\ & S\,;Q' \\ \models & \{\text{assumption}\} \\ & S. \end{array}$$

4. We show $(P\,;(Q \vee Q')) \models R$:

$$\begin{array}{ll} & P\,;(Q \vee Q') \\ \models & \{\text{Theorem 4, distributivity}\} \\ & (P\,;Q) \vee (P\,;Q') \\ \models & \{\text{assumption}\} \\ & R \vee R \\ \models & \{\text{propositional logic}\} \\ & R. \end{array}$$

5.
$$\begin{array}{ll} & (P * P')\,;(Q * Q') \\ \models & \{\text{Theorem 9, the Exchange law}\} \\ & (P\,;Q) *(P'\,;Q') \\ \models & \{\text{assumption}\} \\ & R * R' \end{array}$$

6.
$$\begin{array}{ll} & P\,;Q \models R \\ \Rightarrow & \{\text{Theorem 5, monotonicity of } (*)\} \\ & F *(P\,;Q) \models F * R \\ \Rightarrow & \{\text{Corollary 1}\} \\ & (F * P)\,;Q \models F * R \end{array}$$

□

The last two laws are reminiscent of two of the basic axioms of separation logic [10]. But the standard separation logic rule is stronger and more useful. This is because the standard definition of separating conjunction applies to assertions about the current state. The state is an abstraction of the history of events which caused it. The validity of the abstraction depends on assumptions about the consistency of memory, which in this work we have not made. We leave for future work the investigation of the logical connections between various kinds of weak memory and the validity of the relevant program proof rules.

The weakest precondition of program Q and postcondition R is defined as follows:

Definition 9. $(Q\,-\!;R)(tr) \ \equiv_{\mathbf{def}}\ \forall tq : Q(tq)\ \&\ (tr\,;tq)$ *defined* $\Rightarrow R(tr\,;tq)$.

Informally, a trace satisfies $(Q\,-\!;R)$ if, whenever followed by a trace of Q, the combined trace satisfies R. The following theorem states in terms of Hoare triples that $(Q\,-\!;R)$ is a pre-condition of R under program Q, and that it is the weakest such condition.

Theorem 11 (Galois adjoint)**.** $P\ \{\ Q\ \}\ R\ \Leftrightarrow\ P \models (Q\,-\!;R)$

Proof. In one direction:

$$\begin{array}{ll} & P(tp) \text{ and } Q(tq) \text{ and } tp\,;tq \text{ defined} \\ \Rightarrow & \{\text{definition of } (;)\} \\ & (P\,;Q)(tp\,;tq) \\ \Rightarrow & \{\text{assumption}\} \\ & R(tp\,;tq). \end{array}$$

In the other direction:

$$
\begin{array}{ll}
 & (P\,;Q)(tr) \\
\Rightarrow & \{\text{definition of } (;)\} \\
 & \exists tp, tq : P(tp) \ \& \ Q(tq) \ \& \ tr = tp\,; tq \\
\Rightarrow & \{\text{assumption}\} \\
 & \exists tp, tq : R(tp\,; tq) \ \& \ tr = tp\,; tq \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & R(tr).
\end{array}
$$

□

The $(*)$ operator has a similarly defined adjoint called the magic wand, usually written $-\!\!*$.

6. The Rely/Guarantee Calculus

A predicate is called *acyclic* when all of its traces are acyclic. Some theorems in this section require this assumption to ensure linearizeability.

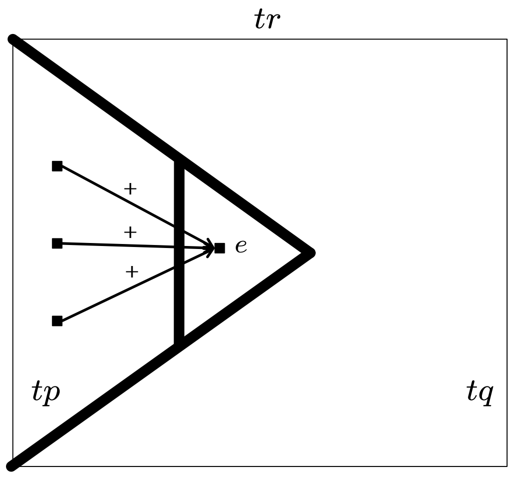

Figure 13. Linearity

Theorem 12 (Linearity). *Suppose $e \in tr$, and tr is acyclic. Then for some tp, tq, $tr = tp\,;\{e\}\,; tq$.*

Proof. Let $tp = \{d \in tr \mid d \xrightarrow{+} e\}$ and $tq = tr \setminus (tp \cup \{e\})$, as shown in Figure 13. It is not the case that $e \rightarrow tp$, because then $e \xrightarrow{+} e$, which violates acyclicity; so $tp\,;\{e\}$ is defined. It is not the case that $tq \rightarrow tp$ because if for some $d \in tq$ such that $d \rightarrow tp$, then also $d \xrightarrow{+} e$, which contradicts the definition of tq; so $tp\,; tq$ is defined. Finally, it is not the case that $tq \rightarrow e$, because $d \in tq$ such that $d \rightarrow e$ again contradicts the definition of tq. □

Predicate G is called an *invariant* when every event of a trace that satisfies G also satisfies G:

Definition 10. G *invariant* $\equiv_{\mathbf{def}} \forall tr : (G(tr) \;\Leftrightarrow\; \forall e \in tr : G(\{e\}))$.

Invariants are also satisfied by the empty trace, which informally means that an invariant can be satisfied by doing nothing.

Theorem 13. *For invariant G:*

1. $skip \models G$
2. $G(tp \cup tq)$ *iff* $G(tp)$ *and* $G(tq)$
3. $G\,;G = G * G = G$
4. $G * [\ell] = G\,;[\ell]\,;G$*, when* $G * [\ell]$ *is acyclic.*

Proof. 1. $\forall e \in \emptyset : G(\{e\})$ is vacuously true, and so $G(\emptyset)$ by definition of invariant.

2.
$$
\begin{array}{ll}
 & G(tp \cup tq) \\
\Leftrightarrow & \{\text{definition of invariant}\} \\
 & \forall e \in (tp \cup tq) : G(\{e\}) \\
\Leftrightarrow & \{\text{set theory}\} \\
 & \forall e \in tp : G(\{e\}) \text{ and } \forall e \in tq : G(\{e\}) \\
\Leftrightarrow & \{\text{definition of invariant}\} \\
 & G(tp) \text{ and } G(tq).
\end{array}
$$

3.
$$
\begin{array}{ll}
 & (G * G)(t) \\
\Leftrightarrow & \{\text{definition of } (*)\} \\
 & \exists tp, tq : t = tp * tq \;\&\; G(tp) \;\&\; G(tq) \\
\Leftrightarrow & \{G \text{ is invariant, part 2}\} \\
 & \exists tp, tq : t = tp * tq \;\&\; G(tp * tq) \\
\Leftrightarrow & \{\text{set theory}\} \\
 & G(t)
\end{array}
$$

4. In one direction:

$$
\begin{array}{ll}
 & (G * [\ell])(t) \\
\Rightarrow & \{\text{definition of } (*) \text{ and } [\ell]\} \\
 & \exists tr, e : G(tr) \;\&\; t = tr * \{e\} \\
\Rightarrow & \{\text{Theorem 12, linearity}\} \\
 & \exists tr1, tr2, e : G(tr1 * tr2) \;\&\; t = tr1\,;\{e\}\,;tr2 \\
\Rightarrow & \{\text{G is invariant, part 3}\} \\
 & \exists tr1, tr2, e : G(tr1) \;\&\; G(tr2) \;\&\; t = tr1\,;\{e\}\,;tr2 \\
\Rightarrow & \{\text{definition of } (;)\} \\
 & (G\,;[\ell]\,;G)(t)
\end{array}
$$

In the other direction:

$$
\begin{array}{ll}
 & G\,;[\ell]\,;G \\
\Rightarrow & \{(;) \subseteq (*) \text{ and commutativity of } (*)\} \\
 & G * G * [\ell] \\
\Rightarrow & \{\text{G is invariant, part 3}\} \\
 & G * [\ell]
\end{array}
$$

□

The strongest invariant implied by both G and G' is $(G \wedge G')$; the predicate $G \nabla G'$ defined below is the weakest invariant that implies both G and G':

Definition 11. $G \nabla G'(tr) \; \equiv_{\mathbf{def}} \; \forall e \in tr : G(\{e\}) \vee G'(\{e\})$.

It is easy to see that a predicate G is an invariant iff $G = G \nabla G$; other facts are collected below.

Theorem 14. *For invariant G:*

1. $Q \models G \;\; \& \;\; Q' \models G' \;\; \Rightarrow \;\; (Q * Q') \models (G \nabla G')$
2. $Q \models G \;\; \& \;\; Q' \models G' \;\; \Rightarrow \;\; (Q\,; Q') \models (G \nabla G')$

Proof. We show $(Q * Q') \models (G \nabla G')$; the other implication follows because $Q\,; Q' \models Q * Q'$. Let t be an arbitrary trace.

$$\begin{array}{ll} & (Q * Q')(t) \\ \Rightarrow & \{\text{definition of } (*)\} \\ & \exists tq, tq' : Q(tq) \;\; \& \;\; Q'(tq') \;\; \& \;\; t = tq * tq' \\ \Rightarrow & \{\text{assumption}\} \\ & \exists tq, tq' : G(tq) \;\; \& \;\; G'(tq') \;\; \& \;\; t = tq * tq' \\ \Rightarrow & \{\text{definition of } (\nabla) \text{ and } (*)\} \\ & (G \nabla G')(t). \end{array}$$

□

As in the Jones rely/guarantee calculus, invariants are used to describe ways in which one process is permitted to interfere with another. A rely condition is an invariant that describes the assumptions a process makes about interference from its environment during execution; similarly, a guarantee condition is an invariant that describes the guarantee that a process makes regarding its own interference with the environment. Satisfaction of the invariant condition G by process Q is simply expressed by the implication $Q \models G$.

The *Jones quintuple* $P \; R \; \{ \; Q \; \} \; G \; S$ is a partial correctness specification of program Q in the presence of interference from other threads. It allows the program Q to rely on the environment to satisfy the invariant R, and in turn guarantees the condition G. $(R * Q)$ also satisfies postcondition S on the assumption of precondition P:

Definition 12. $P \; R \; \{ \; Q \; \} \; G \; S \; \equiv_{\mathbf{def}} \; P \; \{ \; R * Q \; \} \; S \;\; \& \;\; Q \models G$.

The base rule of the Jones calculus reduces concurrent reasoning to sequential reasoning within a single thread.

Theorem 15. $P \; R \; \{ \; [\ell] \; \} \; G \; S \; \Leftrightarrow \; P \; \{ \; R\,; [\ell]\,; R \; \} \; S \;\; \& \;\; [\ell] \models G$*, when* $R * [\ell]$ *is acyclic.*

Proof.

$$\begin{array}{ll} & P \; R \; \{ \; [\ell] \; \} \; G \; S \\ \Leftrightarrow & \{\text{definition}\} \\ & P \; \{ \; R * [\ell] \; \} \; S \text{ and } [\ell] \models G \\ \Leftrightarrow & \{\text{Theorem 13, part 4}\} \\ & P \; \{ \; R * [\ell] * R \; \} \; S \text{ and } [\ell] \models G \end{array}$$

□

Theorem 16 (Concurrency)**.** $P \; R \; \{ \; Q \; \} \; G \; S \;\; \& \;\; P' \; R' \; \{ \; Q' \; \} \; G' \; S' \; \Rightarrow$ $(P \wedge P') \; (R \wedge R') \; \{ \; Q * Q' \; \} \; (G \nabla G') \; (S \wedge S')$*, when* $G' \models R$ *and* $G \models R'$.

Proof. $Q * Q' \models G\nabla G'$ follows from Theorem 14. Below we show $(P \wedge P')\ \{\ (R \wedge R') * (Q * Q')\ \}\ S$. By a similar argument it can be shown that $(P \wedge P')\ \{\ (R \wedge R') * (Q * Q')\ \}\ S$, from which the conclusion follows from Theorem 10.

$$\begin{array}{ll}
 & (P \wedge P')\,;((R \wedge R') * (Q * Q')) \\
\models & \{\ (;) \text{ and } (*) \text{ are monotonic}\ \} \\
 & P\,;(R * (Q * Q')) \\
\models & \{\ (*) \text{ is monotonic and } Q' \models G'\ \} \\
 & P\,;(R * (Q * G')) \\
\models & \{\ (*) \text{ is monotonic and } G' \models R\ \} \\
 & P\,;(R * (Q * R)) \\
\models & \{\ \text{associativity and commutativity of } (*)\ \} \\
 & P\,;((R * R) * Q) \\
\models & \{\ R \text{ is invariant, so } R * R = R \text{ by Theorem 13, part 3}\ \} \\
 & P\,;(R * Q) \\
\models & \{\ P\,;(R * Q) \models S \text{ by assumption}\ \} \\
 & S.
\end{array}$$

□

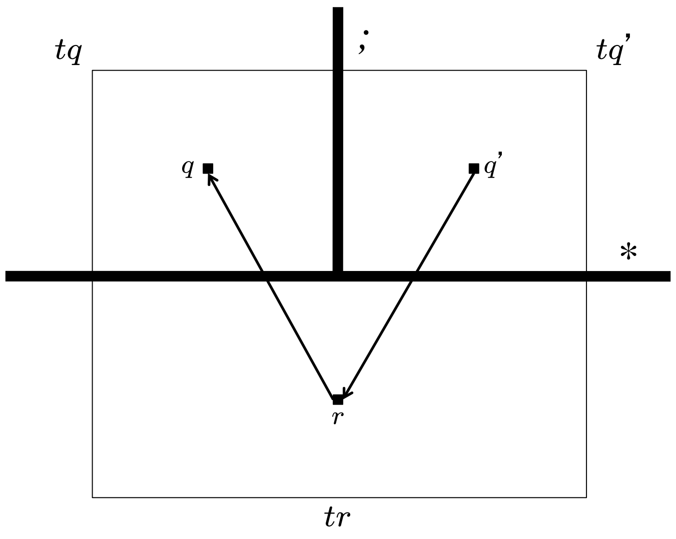

Figure 14. Weak sequentiality violates the Jones rule for sequential composition

The Jones rule for sequential composition is:

$$P\ R\ \{\ Q\ \}\ G\ S\ \ \&\ \ S\ R'\ \{\ Q'\ \}\ G'\ S'\ \Rightarrow\ P\ (R \wedge R')\ \{\ Q\,;Q'\ \}\ (G\nabla G')\ S'.$$

To prove soundness of this rule, we have to show $Q\,;Q' \models G\nabla G'$ and $P\ \{\ (R \wedge R') * (Q\,;Q')\ \}\ S$. The first assertion follows from Theorem 14, but the second assertion is not valid in this setting. Consider the following model, where $q \leftarrow r \leftarrow q'$:

$$\begin{array}{ll} P = \{\emptyset\} & \\ R = \{\{r\}, \emptyset\} & R' = \{\{r\}, \emptyset\} \\ Q = \{\{q\}\} & Q' = \{\{q'\}\} \\ S = \{\{r, q\}, \{q\}\} & S' = \{\{q, q'\}\} \end{array}$$

The trace $\{r, q, q'\}$ is shown in Figure 14. The reader can check that the antecedents hold in this model, and also that $\{r, q, q'\}$ satisfies $P\,;((R \wedge R') * (Q\,;Q'))$, but not S'. The countermodel resembles the paradoxical example in Figure 3 in that there is no direct dependency between q and q', but interference from the environment yields an indirect dependency from the second operand to the first.

The Jones rule is of course valid for normal strong sequential composition, since all the events of the second operand would be forced by control dependency to be executed after those of the first operand. The anomalous dependency is then ruled out by acyclicity. But this would sacrifice all opportunity for standard optimisations. All that is necessary is to ensure that the "critical" events in the trace of $P\,;Q$ are connected by a control dependency. In practice (e.g., in a parallel ALGOL language [1]), critical events are protected by an exclusion semaphore; and the definition of the acquisition and release of semaphores requires that they be linearly ordered by a control arrow.

To prove the weak sequential composition axiom in the Jones calculus, we formalize our assumptions as follows. We say an event $p \in tp$ is *critical with respect to tr* when, for some event $r \in tr$, either $p \leftrightarrow r$. In established parlance, traces with critical events events would be called critical regions. We say tp is *protected from tr* if every pair of events $p, p' \in tp$ that are critical with respect to tr are connected; i.e., $p \overset{+}{\leftrightarrow}_{tp} p'$. (Recall from Section 2 that we write $\overset{+}{\leftrightarrow}$ to mean $\overset{+}{\rightarrow} \cup \overset{+}{\leftarrow}$, and in particular *not* to indicate the transitive closure of $\leftrightarrow$.) Finally, P is protected from R when every $tp \in P$ is protected from every $tr \in R$, and $P * R$ is acyclic.

In Figure 3, the two events in the trace $tp\,;tq$ are both critical with respect to tr. To protect them, it is necessary to connect these critical events by a chain of dependencies. Then the possibility of a backward dependency is ruled out by acyclicity.

The sequential composition theorem below is weakened to require that the program $(Q\,;Q')$ be protected from the environment $(R \wedge R')$, which rules out the counterexample.

Lemma 5. *If $tq\,;tq'$ is protected from tr and $tr * (tq\,;tq')$ is defined, then there exists $tr1, tr2$ such that $tr = tr1\,;tr2$ and $(tr1 * tq)\,;(tr2 * tq')$ is defined.*

Proof. Let $tr1 = \{e \in tr \mid \exists q \in tq : e \overset{+}{\rightarrow} q\}$ and $tr2 = tr \setminus tr1$. First, $t = tr1\,;tr2$ because if there were some $e1 \in tr1$ and $e2 \in tr2$ such that $e2 \rightarrow e1$, then also $e2 \overset{+}{\rightarrow} q$ for some $q \in tq$, which implies $e2 \in tr1$.

To show that $(tr1 * tq)\,;(tr2 * tq')$ is defined, we also have to show 1) that $tr1 * tq$ and $tr2 * tq'$ are defined, 2) $tq\,;tr2$ is defined, and 3) $tr1\,;tq'$ is defined. By assumption, tr is disjoint from tq and tq', which implies that $tr1 * tq$ and $tr2 * tq'$ are defined. Next, $tq\,;tr2$ holds because if there were some $r \in tr2$ and $q \in tq$ such that $r \rightarrow q$, then also $r \overset{+}{\rightarrow} q$, which implies $r \in tr1$, a contradiction.

Finally, suppose $tr1\,;tq'$ does not hold. Then, for some $q' \in tq'$ and $r \in tr1$, $q' \rightarrow r$. By definition of $tr1$, $q' \rightarrow r \overset{+}{\rightarrow} q$, for some q in tq. Hence also $q \overset{+}{\leftrightarrow}_{(tq\,;tq')} q'$, because

$tq\,;tq'$ is protected from tr. But then either $tq \overset{+}{\leftarrow} tq'$, which violates definedness of $tq\,;tq'$, or $q \overset{+}{\rightarrow} q' \overset{+}{\rightarrow} q$, which violates the acyclicity. □

Lemma 6. *For invariants R, R', $(R \wedge R') * (Q\,;Q') \models (R * Q)\,;(R' * Q)$, when $Q\,;Q'$ is protected from $(R \wedge R')$.*

Proof.
$$
\begin{array}{ll}
 & ((R \wedge R') * (Q\,;Q'))(t) \\
\Rightarrow & \{\text{definition of } (;) \text{ and } (*) \text{ for trace sets}\} \\
 & \exists tr, tq, tq' : (R \wedge R')(tr) \ \&\ (Q)(tq) \ \&\ (Q')(tq') \ \& \\
 & t = tr * (tq\,;tq') \\
\Rightarrow & \{\text{Lemma 5}\} \\
 & \exists tr1, tr2, tq, tq' : (R \wedge R')(tr1\,;tr2) \ \&\ (Q)(tq) \ \&\ (Q')(tq') \ \& \\
 & t = (tr1 * tq)\,;(tr2 * tq') \\
\Rightarrow & \{R \text{ and } R' \text{ are invariant, Theorem 13, part 3}\} \\
 & \exists tr1, tr2, tq, tq' : (R)(tr1) \ \&\ (R')(tr2) \ \&\ (Q)(tq) \ \&\ (Q')(tq') \ \& \\
 & t = (tr1 * tq)\,;(tr2 * tq') \\
\Rightarrow & \{\text{definition of } (*) \text{ and } (;)\ \} \\
 & ((R * Q)\,;(R' * Q'))(t)
\end{array}
$$

□

Theorem 17. $P\ R\ \{\ Q\ \}\ G\ S\ \ \&\ \ S\ R'\ \{\ Q'\ \}\ G'\ S' \Rightarrow$
$P\ (R \wedge R')\ \{\ Q\,;Q'\ \}\ (G \nabla G')\ S'$, *when $(Q\,;Q')$ is protected from $(R \wedge R')$.*

Proof. To show the consequent, we have two obligations:

1. $(Q\,;Q') \models (G \nabla G')$ — By assumption, $Q \models G$ and $Q' \models G'$. The desired property follows from Theorem 14.
2. $P\ \{\ (R \wedge R')\,;(Q\,;Q')\ \}\ S'$ —

$$
\begin{array}{ll}
 & P\ \{\ R;Q\ \}\ S\ \ \&\ \ S\ \{\ R';Q'\ \}\ S' \\
\Rightarrow & \{\text{sequential composition of Hoare logic}\} \\
 & P\ \{\ (R * Q)\,;(R' * Q')\ \}\ S' \\
\Rightarrow & \{\text{Lemma 6}\} \\
 & P\ \{\ (R \wedge R') * (Q\,;Q')\ \}\ S.
\end{array}
$$

□

7. Graphical Models of Advanced Programming Primitives

In this section, graphical models are described for more advanced programming primitives. The same notational conventions are used as in Section 3.

First consider the CCS [8] program $b!x \mid b?y.P(y)$. There are two possibilities for its execution: either the output from the first process is directly communicated to the second process, as previously shown in Figure 9; or the input and output commands are interleaved, as in Figure 15. In spite of the fact that the channel name b is the same for both input and output, the channel may be multiplexed in CCS, so that both output and input communicate with their common environment, instead of with each other. This models the sequential programming phenomenon of interference.

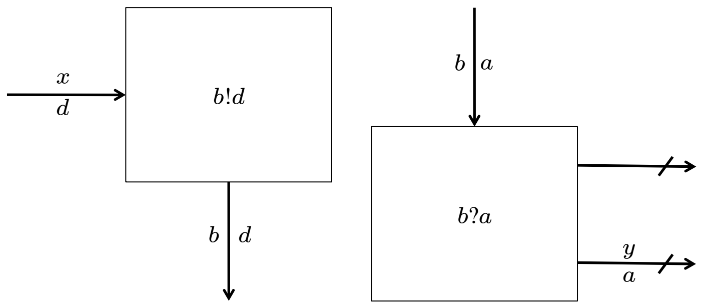

Figure 15. $b!x \mid (b?y.P(y))$: CCS communication

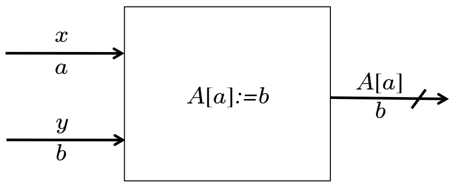

Figure 16. $A[x] := y$: Array assignment

Figure 16 shows a trace of an array assignment command, $A[x] := y$. The trace contains a single event indicating the assignment. There is an input arrow for the value a of variable x, and value b of variable y. There are also output arrows, all labeled with the resource $A[a]$ and its value after the assignment, b.

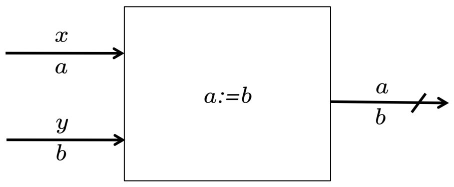

Figure 17. $[x] := y$: Indirect assignment

Figure 17 shows a trace of an indirect assignment command $[x] := y$, as found in an imperative programming languages with references or pointers. The trace is similar to Figure 16. There is an incoming arrow for the value (memory location) a of variable x, and the value b of variable y. Again, there are output arrows, all labeled with the memory location a and its value after the assignment, b. Here we have modeled each element of the array A as a separate resource. This is more realistic that the traditional treatment of arrays in models of Hoare logic, where an assignment to a subscripted variable is treated as assigning an array value to the complete array. An advantage of the new model is that it enables different parts of the same array to be owned and updated simultaneously by concurrent processes.

Figure 6 in Section 3 showed a direct output command, in which the name of channel is fixed in advance. Another level of indirection is needed, however, if channel names are dynamic, as in the π-calculus [9]. An indirect output command $y!x$, as in the π-calculus,

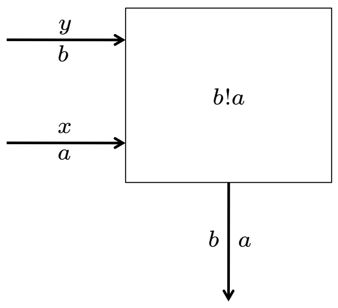

Figure 18. $y!x$: Indirect output

is shown in Figure 18. The trace is identical that of the direct output command, but for another input arrow that provides the output channel name b from variable y.

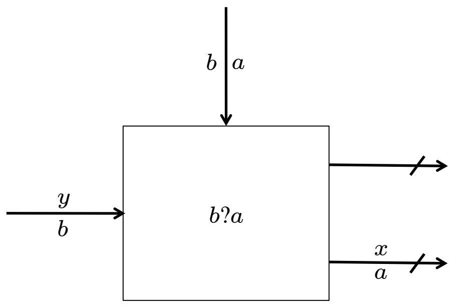

Figure 19. $y?x.P(x)$: Indirect input

Figure 19 shows an indirect blocking input command, $y?x.P(x)$, as found in the π-calculus. The trace is identical to the direct blocking input command, shown in Figure 7 in Section 3, but for another input arrow that provides the input channel name b from variable y.

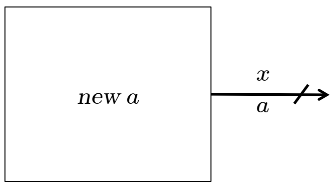

Figure 20. *new* $x.P(x)$: Allocation

The π-calculus can declare a new channel for use in the process P by the notation *new* $x.P(x)$. Figure 20 shows a trace of the command *new* $x.P(x)$ from the π-calculus. Allocation of the new channel is modelled as an event added to the events of P. The additional event has no incoming arrows. As in the case of the input command, its outgoing arrows are just the set of input arrows of P that are labelled by with resource x and value a, the name of the new channel. In contrast with the input command, there are no control arrows, so the exact timing of the new event is undetermined. The semantics of the *new* command is embodied in two constraints on the definition of a trace: 1) for each memory location a, at most one event event labeled *new* a must appear; and 2) a *new* a event must be connected to every event e labeled with a (i.e., $d \overset{*}{\rightarrow} e$, where $label(d) = \textit{new}\ a$).

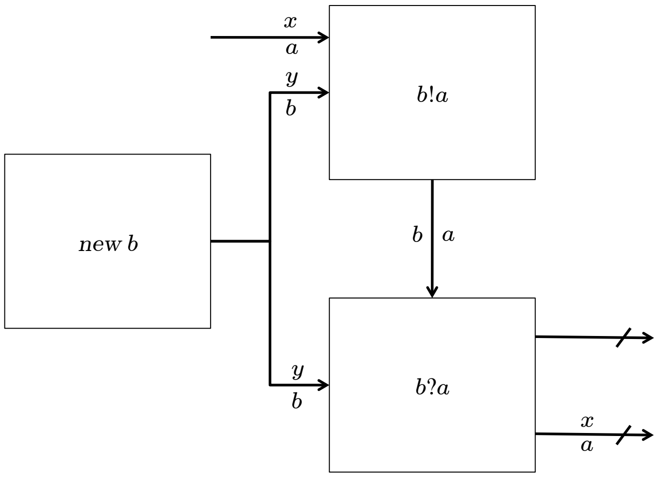

Figure 21. *new* $y.(y!x \mid y?x.P(x))$: π-calculus communication

Consider again the program $b!x \mid (b?y.P(y))$, which transfers the value stored in variable x across the shared channel b. In the π-calculus as in CCS, communication may occur either directly between the two processes, or may be interleaved with events from the environment. By using a newly allocated channel, interleaving can be avoided because no other processes can interfere. A trace of program *new* $y.(y!x \mid y?x.P(x))$ is shown in Figure 21. Note that the allocation event provides the channel name to both the channel input and output events, and the only input arrow is for the value of x to be communicated.

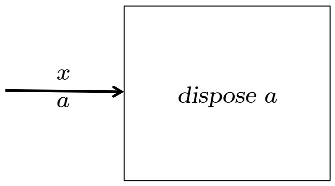

Figure 22. *dispose* x: Deallocation

The trace of the *new* command in Figure 20 could also be a trace of a memory allocation command in a C-like language. In Figure 22, we show a trace of a dispose command, which frees a previously allocation memory location. The trace contains a single event and no output arrows. There is a single input arrow that gives the memory location to be disposed. As with the *new* command, the semantics of the *dispose* command is embodied in two constraints on the definition of a trace: 1) for each memory location a, at most one event event labeled *dispose* a must appear; and 2) every event e labeled with a must be connected to a *dispose* a event (i.e., $e \overset{*}{\rightarrow} d$, where $label(d) = \textit{dispose}\ a$).

Acknowledgements

The authors thank the following colleagues for their comments: Josh Berdine, Ernie Cohen, Philippa Gardner, Bernhard Möller, Aleksander Nanevski, Viktor Vafeiadis, Glyn Winskel, Hongseok Yang and the members of IFIP WG 2.3.

References

[1] Stephen D. Brookes. The essence of parallel ALGOL. *Inf. Comput.*, 179(1):118–149, 2002.

[2] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 123–134. ACM, 2007.

[3] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[5] Peter Höfner and Georg Struth. Automated reasoning in Kleene algebra. In *CADE'07, volume 4603 of LNAI*, pages 279–294. Springer, 2007.

[6] Cliff B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.

[7] Dexter Kozen. On Kleene algebras and closed semirings. In *Proceedings, Math. Found. of Comput. Sci.*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer-Verlag, 1990.

[8] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[9] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[10] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[11] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[12] Vaughan R. Pratt. The pomset model of parallel processes: Unifying the temporal and the spatial. In Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 1984.

[13] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.

[14] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[15] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[16] Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.