PERMISSION-BASED SEPARATION LOGIC FOR MESSAGE-PASSING CONCURRENCY

ADRIAN FRANCALANZA, JULIAN RATHKE, AND VLADIMIRO SASSONE

ICT, University of Malta
e-mail address: adrian.francalanza@um.edu.mt

ECS, University of Southampton, UK
e-mail address: jr2@ecs.soton.ac.uk

ECS, University of Southampton, UK
e-mail address: vs@ecs.soton.ac.uk

Abstract. We develop local reasoning techniques for message passing concurrent programmes based on ideas from separation logics and resource usage analysis. We extend processes with permission-resources and define a reduction semantics for this extended language. This provides a foundation for interpreting separation formulas for message-passing concurrency. We also define a sound proof system permitting us to infer satisfaction compositionally using local, separation-based reasoning.

1. Introduction

Reasoning about concurrent programmes is widely acknowledged to be a difficult business due to the intricate interferences between threads scheduled non-deterministically and to the intrinsic difficulty of scaling reasoning techniques to account for these. The use of local reasoning techniques in the guise of separation logic [33, 28] represents a promising advance for this area. Here, the state of resources acted upon by threads are reasoned about independently, where possible. This approach has spawned numerous papers [7, 5, 37, 11, 15, 29, 10] targeting the shared-variable concurrency model.

An alternative, albeit slightly higher-level, model of concurrency is that of message-passing whereby the only shared resources allowed are the message-passing channels themselves. Access to these shared resources is controlled by the message-passing programming interface and so interfering behaviour is more explicit and therefore can be tracked more readily. This paradigm has been extensively studied using process calculi [21, 26, 27, 34] but has also been efficiently implemented and deployed in more programming oriented settings [16, 32, 3].

In this paper we develop a local reasoning proof system for message-passing concurrent programmes, based on ideas from both concurrent separation logics [28] and permission-based resource analyses [6, 5]. Our initial step towards the broader and ambitious goal of local reasoning

1998 ACM Subject Classification: MANDATORY list of acm classifications.
Key words and phrases: MANDATORY list of keywords.

Copyright © 2023, Creative Commons
for message-passing systems focusses on the study of confluent value-passing programmes, a class large enough to present a significant theoretical challenge while still being of considerable practical interest.

Our approach to using processes as a model for separation-based Hoare-style reasoning centers around the conceptual partitioning of message-passing programmes into ‘programme state’, i.e., the values emitted on asynchronous outputs, and ‘programme code’, i.e., the remaining parallel processes acting on this state. For instance, one way how to view the programme

\[
c_1!4 \parallel c_2!2 \parallel c_1?x.c_2?y. \text{if } x = y \text{then } (c_1!(x, y, x+y)||d!) \text{ else } (c_2!(x, y, x+y)||d!) \quad (1.1)
\]

would be to consider the asynchronous outputs

\[
c_1!4 \parallel c_2!2 \quad (1.2)
\]

as the ‘state’, holding values 4 and 2 at ‘addresses’ \(c_1\) and \(c_2\) and the process

\[
c_1?x.c_2?y. \text{if } x = y \text{then } (c_1!(x, x+x)||d!) \text{ else } (c_2!(x, y, x+y)||d!) \quad (1.3)
\]

as the ‘code’, or state transformer, consuming the values on channels \(c_1\) and \(c_2\) and producing a new state holding the previous values consumed from \(c_1\) and \(c_2\) together with their summation on either of the previously used channels \(c_1\) and \(c_2\), depending on whether these values were equal or not, and signals on channel \(d\). Using such an analogy, we can decompose our analysis and reason about sub-programmes independently. We can interpret assertions over processes such as

\[
c_1(4) \ast c_2(2) \quad (1.4)
\]

This assertion, a conjunction, describes a process reducing to a ‘soup’ of two asynchronous outputs on channels \(c_1\) and \(c_2\), holding values 4 and 2, respectively; the process in \(1.2\) would satisfy this assertion. This state-based process view also permits an intuitive formulation of Hoare-style sequents of the form

\[
\{c_1(4) \ast c_2(2)\} \quad \vdash \quad \{c_2(4, 2, 6) \ast d(\}) \quad (1.5)
\]

Such a sequent describes a process that, once composed with the state described by the precondition \(c_1(4) \ast c_2(2)\), reduces to some other stable state described by the postcondition \(c_2(4, 2, 6) \ast d(\)\), with values 4, 2, 6 on channel \(c_2\) and an empty tuple on channel \(d\) acting as a signal, indicating that the data on channel \(c_2\) can now be accessed; the process in \(1.3\) would satisfy this sequent. In compositional fashion, we can then determine that the entire programme of \(1.1\) reduces to a stable state satisfying \(c_2(4, 2, 6) \ast d(\)\) from separate analyses relating to the two sub-programmes.

This state-based logical view of processes lends itself well to the specification of deterministic computation whose operation can be decomposed into asynchronous parallel subcomponents. Application examples range from parallel processing of data, [23], to distributed agreement problems, [25]. State-based specifications would allow a more natural expression of the expected behaviour of these algorithms because they are agnostic wrt. the specific temporal order of the generation and consumption of this state. For instance, as opposed to temporal logics such as [35], the formula \(1.4\) does not specify whether the sub-state \(c_1(4)\) is to be produced before \(c_2(2)\) or vice-versa. Dually, sequents such as \(1.5\) do not necessarily specify if and how this data on channels is to be consumed. The temporal agnosticism in ‘spatial’ specifications is also more amenable to intuitive decompositions and composition of analysis; we can verify that a process \(P\) satisfies the formula \(1.4\) from sub-processes making up \(P\) that satisfy \(c_1(4)\) and \(c_2(2)\).

The state-based logical process view is also appealing because the specifications of the algorithms we are considering are also, in some sense, more data-centric rather than control-centric and focus more on the relationships between data at the beginning and the end of computation. One
can in fact view the sequent in (1.5) as a description on how the data on channels \( c_1 \) and \( c_2 \) in the precondition changes to the data on \( c_2 \) in the postcondition; the dependencies between such data will be made more explicit later on once we introduce value variables. Finally, data-centric applications such as in-place sorting also tend to reuse data-placeholders during computation, possibly at different types and formats e.g., the code in (1.3), in order to minimise resource usage. Correctness specifications such as the sequent in (1.5) handle this aspect rather naturally as opposed to traditional correctness analysis for message passing programs, such as type systems in [4, 36], which often limit channel usage to one form of data.

A central assumption underlying our process interpretations is the absence of programme interference and the deterministic reduction of processes. In a message-passing paradigm, programme interference is caused by races for values, through multiple outputs or inputs competing for shared channels. In cases such as (1.1) above, where channels are reused, rudimentary analysis based on the free names of processes e.g., [1] are too coarse for adequate race detection. Moreover, these type based safety analyses e.g., [4, 36] tend to avoid reasoning about data, approximating control over branching as a result.

To reason about such interferences in the presence of channel reuse, we define a resource-semantics for processes, based on linear input and output permissions. Every process is embellished with a set of permissions, \([P]_\rho\), denoting that process \( P \) ‘owns’ the permissions in set \( \rho \) (cf. ownership hypothesis [28]). The resource-semantics limits communication to the permissions owned by a process. Thus, for example, for the following reduction to occur

\[
\left\{ c_1 !4 \right\}_\rho \parallel \left\{ c_1 ?x.P \right\}_\mu \rightarrow \left\{ P\upharpoonright x\downarrow \right\}_\rho \downarrow \mu \quad \text{(1.6)}
\]

the output process, \( c_1 !4 \) (resp. the input process, \( c_1 ?x.P \)), must have the permission to output (resp. to input) on channel \( c_1 \) in its permission-set \( \rho \) (resp. \( \mu \)). Since permissions are not part of the original process semantics (they are only added in the resource-semantics to aid reasoning) the above enriched reduction also describes the implicit transfer of permissions \( \rho \) from the output process, \( c_1 !4 \), to the input process, \( c_1 ?x.P \), i.e., adding \( \rho \) to the already owned permissions \( \mu \), as a result of their synchronisation (cf. ownership transfer [28]).

\[
\begin{align*}
\{ c_1(4) \ast c_2(2) \} & \quad \text{c1!4, if } x = y \text{ then } c_1!(x, x+x) \parallel d! \quad \text{else } c_2!(x, y, x+y) \parallel d! \quad \{ c_2(4, 2, 6) \ast d(); \} \\
\{ c_1(1), c_2(2) \} & \quad \text{c14, c22, if } x = y \text{ then } c_1!(x, x+x) \parallel d! \quad \text{else } c_2!(x, y, x+y) \parallel d! \quad \{ c_2(4, 2, 6) \ast d(); \} \\
\{ c_1(1), c_2(2) \} & \quad \text{c14, c22, if } x = y \text{ then } c_1!(x, x+x) \parallel d! \quad \text{else } c_2!(x, y, x+y) \parallel d! \quad \{ c_2(4, 2, 6) \ast d(); \} \\
\end{align*}
\]

The earlier sequent (1.5) can now be stated in terms of the process of (1.3) confined by the permissions \( \downarrow c_1 \), \( \downarrow c_2 \) and \( \uparrow d \), as shown in (1.7). Note how channel reuse manifests itself through the fact that our permission-confined process in (1.7) does not own the output permissions \( \uparrow c_1 \) and \( \uparrow c_2 \), even though they are clearly used in this code. These however will be obtained from the precondition; from a permission perspective, the inputs on channels \( c_1 \) and \( c_2 \) act as guards, masking the use of the permissions \( \uparrow c_1 \) and \( \uparrow c_2 \).

Making ownership explicit also simplifies the detection of races in the model and provides an immediate notion of process separation in terms of owned permissions. For instance, in (1.6) we determine that there are no races across the two processes \( \left\{ c_1 !4 \right\}_\rho \) and \( \left\{ c_1 ?x.P \right\}_\mu \) without having to analyse the actual structure of the respective confined processes \( c_1 !4 \) and \( c_1 ?x.P \); instead we simply check that their permission sets are disjoint i.e., \( \rho \cap \mu = \emptyset \) (cf. separation property [28]). This assumed disjunction of permissions will also play a major role in the semantic definition of (1.7), as it allows us to give a separation interpretation to our sequents.
\begin{align*}
&\{c_1(4) \ast c_2(2)\} \quad \begin{array}{ll}
& \text{if } x = y \text{ then } c_1!(x, x+x) \parallel d!
& \text{else } c_2!(x, y, x+y) \parallel d!
\end{array} \quad \{c_2(4, 2, 6) \ast d(\})
\end{align*}

Another pleasing property of this embellishment is that, in the absence of races, this resource-semantics corresponds to the standard (permission-less) reduction semantics. Thus the permission semantics can be used as a narrative to support reasoning about confluent reductions of processes. This therefore means that we can abstract over the existence of such a narrative in our sequents and express (1.7) simply as the permission-less sequent in (1.8), thereby returning to our original aim and obtaining Hoare-triple specifications in terms of processes.

We define a sound proof system for the aforementioned logic and resource-confined processes with judgements of the form:

$$\Gamma; b \vdash \{\varphi\} S \{\psi\}$$

The environment, $\Gamma$, associates channels with ownership transfer invariants of permissions, and $S$ denotes a system of processes confined by permissions. These sequents depart slightly from previous work on concurrent separation logic [28], as value-domain assertions - assertions interpreted exclusively in terms of the domain of values communicated and thus independent of the process structure, $S$ - are extracted from the pre and post-conditions, $\varphi$ and $\psi$, and consolidated as a boolean expression, $b$. Correctness proofs in this proof system weave together two inter-dependent mechanisms. On the one hand, they verify, in sequential fashion, the satisfaction of the post-condition $\psi$ for system $S$, assuming the precondition $\varphi$; the soundness of this sequential analysis stems from the non-interference properties guaranteed by the resource semantics of $S$. On the other hand, sequents construct race-free systems $S$, using assumptions from the environment, $\Gamma$, and the pre-condition, $\varphi$.

We have already argued for the naturality of our specifications wrt. deterministic message-passing programmes and how our analysis can handle more refined branching control analysis, even when this is data dependent as in (1.1). Another, perhaps even more crucial advantage of our approach over existing analysis techniques for message-passing concurrency (e.g., [20, 2, 13]) is locality of reasoning. By concentrating on deterministic code, our reasoning need not take into account the different interleaving of concurrent code executing in context; this facilitates substantially proof compositionality and induces a lightweight sequential form of analysis. Explicit permission ownership simplifies interference delineation, even in the presence of channel reuse; such delineation is a major obstacle when defining manageable compositional proof rules (e.g., [13]).

The paper is structured as follows. We introduce our language in Section 2. In Section 3 we define a resource-semantics for permission-confined processes and state its key properties. We define our assertion logic and interpret it using a separation model over confined processes in Section 4. In Section 5 we present our proof system and declare its soundness whereas in Section 6 we apply this system to prove properties about message-passing programmes. Finally, in Section 7 we make concluding remarks regarding related and future work.

2. Language

Our language, an asynchronous value-passing CCS, is described in Figure 1 and consists of three syntactic categories. Values, $v, u \in \text{VALUES}$, are numerals denoting integers. Side-effect free expressions, $e$, denote integer operations that may contain variables $x, y \in \text{VARS}$. We assume an evaluation function from closed expressions to values, $e \downarrow v$. We also assume a denumerable set of
Values, Expressions, Boolean Expressions and Processes

\[ v, u ::= 0 \mid 1 \mid \ldots \]
\[ e ::= v \mid x \mid e + e \mid e - e \]
\[ b ::= e \leq e \mid \neg b \mid b \land b \]
\[ P, Q ::= c!\bar{v} \mid c?\bar{x}.P \mid \text{if } b \text{ then } P \text{ else } Q \mid K(\bar{d})[\bar{e}/\bar{d}] \mid \text{nil} \mid P \parallel Q \mid (\text{new } c)P \]

Structural Equivalence Rules

\[ \text{sCom} \quad P \parallel Q \equiv Q \parallel P \]
\[ \text{sAss} \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \]
\[ \text{sNew} \quad (\text{new } c)\text{nil} \equiv \text{nil} \]
\[ \text{sSwap} \quad (\text{new } c)(\text{new } d)P \equiv (\text{new } d)(\text{new } c)P \]
\[ \text{sNil} \quad P \parallel \text{nil} \equiv P \]
\[ \text{sExt} \quad P \parallel (\text{new } c)Q \equiv (\text{new } c)(P \parallel Q) \quad \text{if } c \notin \text{fn}(P) \]

Reduction Rules

\[ \text{rThen} \quad \frac{b \parallel tt}{\text{if } b \text{ then } P \text{ else } Q \rightarrow P} \quad \text{rElse} \quad \frac{b \parallel ff}{\text{if } b \text{ then } P \text{ else } Q \rightarrow Q} \]
\[ \text{rCom} \quad \frac{c!\bar{v} \parallel c?\bar{x}.P \rightarrow P[\bar{v}/\bar{x}]}{P \rightarrow P'} \quad \text{rPar} \quad \frac{P \parallel Q \rightarrow P'}{P \rightarrow Q} \]
\[ \text{rRes} \quad \frac{P \rightarrow P'}{(\text{new } c)P \rightarrow (\text{new } c)P'} \quad \text{rRPar} \quad \frac{P \parallel Q \parallel P' \rightarrow P'}{P \rightarrow Q} \]
\[ \text{rStr} \quad \frac{P \equiv P' \rightarrow Q' \equiv Q}{P \rightarrow Q} \]

Figure 1: Processes, Structural Equivalence and Reduction

channel names \(c, d \in \text{NAMES}\) and process names \(K \in \text{PNames}\) and denote lists of values, variables and channels as \(\bar{v}, \bar{x}\) and \(\bar{c}\) respectively.

2.1. Syntax. The main syntactic category is that of processes which can asynchronously send the evaluation of expressions on a channel\(^1\) \(c!\bar{c}\), receive values on a channel, \(c?\bar{x}.P\), and branch on the evaluation of boolean expressions, \(\text{if } b \text{ then } P \text{ else } Q\). Processes may assume a number of parameterised (possibly recursive) process definitions, \(K(\bar{d}) \equiv P\); these can be invoked by the call \(K(\bar{d})[\bar{e}/\bar{d}]\), instantiating the process variables \(\bar{x}\) with the evaluation of \(\bar{e}\) and renaming the names \(\bar{d}\) to \(\bar{e}\). Finally, processes may also be inactive, \(\text{nil}\), execute in parallel, \(P \parallel Q\), and can restrict the scope of channels to subsets of processes, \((\text{new } c)P\).

2.2. Reduction Semantics. The rules for the judgement \(P \rightarrow Q\) in Figure 1 describe the dynamics of closed processes i.e., processes whose message variables \(\bar{x}\) are all bound by input constructs \(c?\bar{x}\), and process names are all defined. Closed boolean expressions, i.e., boolean formulas without free variables, have a classical interpretation over the boolean domain \([tt, ff]\), characterised by the two judgements \(b \parallel tt\) and \(b \parallel ff\). Although this is entirely standard, we explicitly stated here in Definition\(^2\) due to its central role in subsequent development (cf. Section 5).

\(^1\)Our language does not allow channel names to be communicated, as in the piCalculus\(^2\)\(^\[27\]\(^\[34\]\).
Definition 2.1 (Boolean Condition Interpretation).

\[
e_1 \leq e_2 \begin{cases} 
  \text{tt} & \text{if } e_1 \downarrow v_1, e_2 \downarrow v_2 \text{ and } v_1 \leq v_2 \\
  \text{ff} & \text{if } e_1 \downarrow v_1, e_2 \downarrow v_2 \text{ and } v_2 < v_1 \\
  \neg \begin{cases} 
    \text{tt} & \text{if } b \downarrow \text{ff} \\
    \text{ff} & \text{if } b \downarrow \text{tt}
  \end{cases} & \text{if } b_1 \land b_2 \\
  \begin{cases} 
    \text{tt} & \text{if } b_1 \downarrow \text{tt} \text{ and } b_2 \downarrow \text{tt} \\
    \text{ff} & \text{if } (b_1 \downarrow \text{ff} \text{ and } b_2 \downarrow \text{tt}) \text{ or } (b_1 \downarrow \text{tt} \text{ and } b_2 \downarrow \text{ff}) \text{ or } (b_1 \downarrow \text{ff} \text{ and } b_2 \downarrow \text{ff})
  \end{cases}
\end{cases}
\]

A number of shorthand conventions are used. We write \(c\) for \(c!\vec{e}\) and \(c?.P\) for \(c?\vec{x}.P\) when \(|\vec{x}| = 0\) (resp. \(|\vec{x}| = 0\)). We elide arguments and renaming from process calls, resp. \(K[\vec{c}/\vec{d}]\) and \(K(\vec{c})\), whenever these are empty lists. We also write \(e_1 = e_2\) for \((e_1 \leq e_2) \land (e_2 \leq e_1)\), \(e_1 < e_2\) for \(\neg(e_2 \leq e_1)\), \(\text{true}\) for \(0 \leq 1\), \(\text{false}\) for \(1 \leq 0\), \(b_1 \lor b_2\) for \(\neg(\neg b_1 \land \neg b_2)\) and \(b_1 \Rightarrow b_2\) for \(\neg b_1 \lor b_2\). Finally, we use the shorthand \(\vec{e}[\vec{v}]/\vec{v}\) for the evaluation of lists of expressions \(e_1[\vec{v}_1], \ldots, e_n[\vec{v}_n]\) whenever \(\vec{e} = e_1[\vec{v}], \ldots, e_n[\vec{v}]\). Substitutions, \(\sigma \in \text{Sub}\), are total maps from variables to values, \(\text{Vars} \rightarrow \text{Values}\), and are used to define the semantics of rules \(\kappa\text{Com}\) and \(\kappa\text{PrC}\). They are finitely denoted as \(\vec{v}/\vec{z}\), meaning that every \(x_i \in \vec{x}\) is mapped to its respective \(v_i \in \vec{v}\), while abstracting over all the other variable mappings in the substitution. In the case of \(\kappa\text{PrC}\) only, we abuse this notation to express the \emph{renaming} of \(\vec{d}\) to \(\vec{c}\). In Section 5 we abuse again this notation to describe substitutions from variables to expressions, \(\vec{v}/\vec{z}\). Our semantics assumes the following property of expression evaluations, which will be useful later in Section 5.

Assumption 2.2. \(e_1[\vec{v}/\vec{z}] \downarrow v_1\) and \(\vec{e}[\vec{v}]/\vec{v}\) implies \(e_1[\vec{v}/\vec{z}] \downarrow v_1\).

As is standard in process calculi presentations [27] [34], the definition of the reduction semantics is kept compact through the rule \(\kappa\text{Str}\) and the use of process structural equivalence rules, \(P \equiv Q\), defined also in Figure 1. Later on, this structural equivalence will play a role in abstracting away from the precise structure of processes when describing the satisfaction of our logic (cf. Section 4).

2.3 Process Determinism. The reduction semantics of Figure 1 induces the following definitions relating to stability, evaluation and determinism, where \(\rightarrow^*\) denotes the reflexive transitive closure of \(\rightarrow\).

Definition 2.3 (Stability). \(P \rightarrowdef \exists Q. P \rightarrow Q\)

Definition 2.4 (Evaluation). \(P[Q \downarrowdef \exists Q'. P \rightarrow^* Q' \text{ and } Q' \rightarrowdef]\) and \(Q' \equiv Q\)

Our definition of process determinism, Definition 2.6, differs from that given in [26] in that it requires convergence, \(P \downarrow\), defined as the least set of processes satisfying the conditions given in Definition 2.5. We also define divergence, as the dual of convergence in standard fashion, in order to describe the existence of an infinite reduction path. Defining determinism in terms of convergence carried other advantages apart from the obvious relevance of termination in resource-aware settings of computation; it arguably allows for a more intuitive definition of determinism in terms of the comparison of the stable processes evaluated to (the second clause in Definition 2.6). Moreover, it fits well with our running theme of a state-based view of processes.

Definition 2.5 (Convergence and Divergence).

\(P \downarrowdef P \rightarrow \text{ or } (\forall Q. P \rightarrow Q \implies Q \downarrow)\)

\(P \uparrowdef P \not\downarrow\)
Definition 2.6 (Determinism). P is deterministic iff:
(1) $P \downarrow$
(2) $P \parallel Q_1$ and $P \parallel Q_2$ implies $Q_1 \equiv Q_2$

Concurrent code is notoriously hard to analyse. One major source of complication is the potential non-deterministic behaviour of this code, which impacts the ability to tractably define manageable compositional proof rules (e.g., \cite{13}). More precisely, generic non-deterministic code requires one to take into account the various interleaving of concurrent code executing in its context potentially affecting its execution.

Although message passing concurrency minimises this interference to well defined interfaces, problems persist due to races on shared channels. Channel reuse together with the lack of an explicit account of resource usage makes interference hard to delineate.

Example 2.7. The (composite) process $Prg$ takes two inputs $x_1$, $x_2$ on channels $c_1$, $c_2$ respectively. It discards $x_2$ and, if $x_1$ is less than 10, outputs the value $x_1$ itself together with its double on $c_1$ while using $c_4$ as a signal. Otherwise, it uses $c_4$ to output $x_1$ by itself.

\[
Prg \triangleq (\text{new } c_3) (\text{Fltr} \parallel \text{Dbl})
\]
\[
Dbl \triangleq c_2 \parallel x_2, c_3 \parallel x_4, c_1 \parallel (x_4 + x_4)
\]
\[
\text{Fltr} \triangleq c_1 \parallel x_1 \text{ if } x_1 \leq 9 \text{ then } c_3 \parallel x_1 \parallel (c_1 \parallel x_3, (c_1 \parallel x_1, x_3)) \text{ else } c_4 \parallel x_1
\]

Internally, $Prg$ is composed of two sub-processes, $\text{Fltr}$ and $\text{Dbl}$, sharing a scoped channel, $c_3$. Process $\text{Fltr}$ filters whether $x_1$ is less than 10 and forwards the value to process $\text{Dbl}$ on channel $c_3$, which, in turn, reuses channel $c_1$ to return the doubled value.

The process $Prg$ trivially converges as it is stable. When placed in the context of race-free outputs such as $c_1 \parallel v_1 \parallel c_2 \parallel v_2$, $Prg$ still converges and evaluates deterministically to

\[
Prg \parallel c_1 \parallel v_1 \parallel c_2 \parallel v_2 \downarrow c_4 \parallel c_1 \parallel (v_1, 2 \times v_1)
\]
when $v_1 \leq 9$ and;

\[
Prg \parallel c_1 \parallel v_1 \parallel c_2 \parallel v_2 \downarrow c_4 \parallel v_1 \parallel (\text{new } c_3) (c_3 \parallel x_4, c_1 \parallel (x_4 + x_4))
\]
when $v_1 > 9$

On the other hand, races on, for example, channel $c_1$ make $Prg$ behave non-deterministically. For instance, when placed in the context of two outputs on $c_1$, such as $c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel c_3$, we have:

\[
Prg \parallel c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel 1 \downarrow c_4 \parallel c_1 \parallel (1, 2) \parallel c_1 \parallel 13
\]
or;

\[
Prg \parallel c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel 1 \downarrow c_4 \parallel c_1 \parallel (3, 6) \parallel c_1 \parallel 1
\]

When placed in the context of two outputs on $c_1$ with values that are bigger or equal to 10, such as $c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel 10$, non-deterministic behaviour varies even more widely in structure. In fact we can have:

\[
Prg \parallel c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel 10 \downarrow c_4 \parallel c_1 \parallel (1, 2) \parallel c_1 \parallel 10
\]
or;

\[
Prg \parallel c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel 10 \downarrow c_4 \parallel 10 \parallel (\text{new } c_3) (c_3 \parallel x_4, c_1 \parallel (x_4 + x_4)) \parallel c_1 \parallel 1
\]

Dually, when $Prg$ is placed in the context of $c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel x.nil$, which introduces another input competing for the output on $c_1$, we have even more non-deterministic behaviour. We can have:

\[
Prg \parallel c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel x.nil \downarrow c_4 \parallel c_1 \parallel (1, 2) \parallel c_1 \parallel x.nil
\]
or;

\[
Prg \parallel c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel x.nil \downarrow (\text{new } c_3) (\text{Fltr} \parallel c_3 \parallel x_4, c_1 \parallel (x_4 + x_4))
\]
or even;

\[
Prg \parallel c_1 \parallel c_2 \parallel v_2 \parallel c_1 \parallel x.nil \downarrow c_1 \parallel x_3, (c_1 \parallel x_1, x_3) \parallel c_4
\]

In practice, a substantial body of concurrent code is expected to behave deterministically under some form of non-interference assumptions. One example is the in-place quicksort algorithm, which can be encoded in our language as shown in Example 2.8. In this example, determinism is even
harder to ascertain because, apart from channel reuse, the code is also recursively defined. This gives us scope for developing refined analysis techniques for deterministic code which lend themselves better to compositionality.

**Example 2.8 (In-Place Quicksort).** The process definition $Qck(i, j)$ defines a quicksort algorithm, sorting arrays of values *in-place* and signalling on channel $r$ once sorting completes. Arrays of integers $a = [v_1, \ldots, v_n]$ are represented as a set of messages $a_1!v_1 \parallel \ldots \parallel a_n!v_n$ on an indexed set of channels $a_1 \ldots a_n$. When arrays are of length 1, $Qck(i, i)$ signals immediately on channel $r$. Otherwise, it chooses the value at the lowest index, $a_1!v_1$, as the pivot, partitions the array, and then calls quicksort recursively on the two partitions, renaming the returning signal to a fresh channel name in each case. Once the two sub-sortings signal back, the process can signal back on $r$.

$$Qck(i, j) \triangleq \begin{cases} 
    \text{if } i = j \text{ then } r! \\
    \text{else (new } r_3) & \begin{cases} 
    \text{Prt}(i, j)[r/h] \\
    \| r_3?x.(\text{new } r_1, r_2) & \begin{cases} 
    Qck(i, x-1)[r/h] \\
    \| Qck(x+1, j)[r/h] \\
    \| r_1?.r_2?.r!
    \end{cases}
    \end{cases}
\end{cases}$$

At the heart of quicksort is $\text{Prt}(i, j)$, which partitions an array into two sub-arrays separated by a pivot cell, $a_p!v_p$, and signals completion by outputting the partition index as a value, $r!p$. After partitioning completes, the values in the first sub-array (i.e., indexes less than $p$) are less than $v_p$ and the values of the second sub-array (i.e., indexes greater than $p$) are bigger or equal to $v_p$. Partitioning calls the array traversal process $\text{Trv}(l, h, x, p, c)$, initialising the pivot value $x$ to $v_i$, the pivot index $p$ to $i$, the counter index $c$ to $i+1$ and low and high array boundaries $l$ to $i$ and $j$ respectively.

$$\text{Prt}(i, j) \triangleq a_i?x. \text{Trv}(i, j, x, i, i+1)$$

Traversal loops through the indexes $i + 1$ up to $h$, (6) then (1), comparing their values with the pivot value, (2). If the current value is greater or equal to $x$, in-place partitioning restores the cell and increments the counter, (3). Otherwise, it increments the pivot index to $p+1$, swaps the current value with the value at the new pivot index, and proceeds to the next index, (4). Since reads are destructive in value passing concurrency, swapping occurs only when the two indexes are distinct, (5). Once traversal exceeds the highest index of the array, (6), the pivot value at the lowest index $l$ is swapped with the value at the current pivot index $p$ and the pivot index is returned as the return value $r!p$, (7); again swapping is avoided if these two indexes are the same.

$$\text{Trv}(l, h, x, p, c) \triangleq \begin{cases} 
    \text{(6) if } c > h \text{ then (2) if } l = p \text{ then (3) else (1) } a_c!?y. & \begin{cases} 
    \text{else } a_p?!y. & \begin{cases} 
    \text{else (5) if } c = p + 1 \text{ then (4) else (4) } a_p+1?!z. & \begin{cases} 
    \text{else } a_p+1?!z. & \begin{cases} 
    \text{else } a_p+1?!z. & \begin{cases} 
    \text{else } a_p+1?!z.
    \end{cases}
    \end{cases}
    \end{cases}
    \end{cases}
    \end{cases}
\end{cases}$$

We note that the splitting of the array during recursive calls in $Qck(i, j)$ in Example 2.8 is *data dependent*, based on the pivot value returned after a call to $\text{Prt}(i, j)$. This fact complicates confluence analysis through static techniques such as type systems for resource usage (e.g., [4, 36]). To be able to deal with the refined analysis required for this example, we define a resource-semantics

---

2Since our language can branch on integer values, channel indexing can be encoded.
for our processes in Section 3, which does not approximate over data dependent branching. This extended semantics then serves as a model for a resource-aware separation logic for processes, given in Section 4. In Section 5 we then define a compositional proof system for verifying properties in this logic.

3. Resourcing for Processes

We define a reduction semantics for our programmes by confining their behaviour through linear permissions for channel input and output. This confined-process semantics helps us to reason about deterministic behaviour of processes and lays the foundation for the semantics of the logic to be presented in Section 4. In particular, it (1) gives us a basis for process separation, (2) assists race detection, and (3) acts as a narrative as to why a process is deterministic.

3.1. Systems. We start by defining permission sets. These are used as logical embellishments to readily track channel usage and detect race conditions through conflicting permission usage.

**Definition 3.1 (Permissions).** The set of permissions is \( \text{Perm} \) = \( \{\downarrow, \uparrow\} \times \text{Names} \), where \( \downarrow c \) (resp. \( \uparrow c \)) represents the permission to input (resp. output) on channel \( c \). A permission-set, ranged over by the variables \( \rho, \mu \), is a subset of permissions, \( \rho \subseteq \text{Perm} \).

Permissions are linear in the sense that there are at most one output permission and one input permission per channel. This is not to be confused with linear (resp. affine) assumptions[17] or types[24], which restrict channel usage to exactly (resp. at most) once. In our case, permissions are not consumed once used, but are instead transferred around and reused. Thus, instead of restricting the number of uses of a particular channel, they ensure that, at any stage during computation, there is at most one process that can output (resp. input) on a particular channel.

Figure 2 defines the syntax and semantics of systems of confined processes, \( S, T, R \in \text{Sys} \), whereby processes, \( P \), are confined by permission sets, \( \rho \), and denoted as \([P]_\rho \). Systems can also be composed in parallel and their channels can also be scoped.

Confinement allows us to define separation across systems, \( S \perp T \), on the basis of the (visible) permissions owned by a system, Definition 3.2. In what follows, we assume systems of confined processes to always be well-resourced, meaning that all confined parallel processes are separate, i.e., there is no overlap across owned permission sets, and that permissions are linear. System well-resourcing, denoted \( \vdash S \), is formalised in Definition 3.4. It can be easily checked statically by induction on the structure of systems.

**Definition 3.2 ((Visibly) Owned Permissions).**

\[
\text{prm}(S) \begin{cases} 
\rho & \text{if } S = [P]_\rho \\
\text{prm}(T) \cup \text{prm}(R) & \text{if } S = T \parallel R \\
\text{prm}(T) \setminus \{\downarrow c, \uparrow c\} & \text{if } S = (\text{new } c)T 
\end{cases}
\]

**Definition 3.3 (Separation).** \( S \perp T \) = \( \text{prm}(S) \cap \text{prm}(T) = \emptyset \)

**Definition 3.4 (Well-Resourced System).** A system \( S \) is well-resourced, denote as \( \vdash S \), if it is included in the least set defined by the following three rules.

\[
\begin{align*}
\text{wPre} & \quad \vdash [P]_\rho \\
\text{wPar} & \quad \vdash S \quad \vdash T \\
\text{wRes} & \quad \vdash S \quad \vdash (\text{new } c)S
\end{align*}
\]
Confined Processes (Systems)

\[ S, T, R ::= [P]_p \mid S \parallel T \mid (\text{new } c) S \]

Permission Violation Detection Rules

\[
\begin{align*}
\text{PermOut} & : [c\xi]_p \rightarrow_{\text{err}} [c\xi]_p \quad & \ell \not\in \rho
\end{align*}
\]

\[
\begin{align*}
\text{PermIn} & : [c?\chi]_p \rightarrow_{\text{err}} [c?\chi]_p \quad & \ell \not\in \rho
\end{align*}
\]

\[
\begin{align*}
\text{PermPar} & : S \rightarrow_{\text{err}} S \parallel S \rightarrow_{\text{err}} S \parallel T \rightarrow_{\text{err}} T \rightarrow_{\text{err}}
\end{align*}
\]

Structural Equivalence Rules

\[
\begin{align*}
\text{scCom} & : S || T \equiv T || S
\end{align*}
\]

\[
\begin{align*}
\text{scAss} & : (S || T || R) \equiv (S || T) || R
\end{align*}
\]

\[
\begin{align*}
\text{scNew} & : (\text{new } c)[\text{nil}]_b \equiv [\text{nil}]_b
\end{align*}
\]

\[
\begin{align*}
\text{scSwp} & : \text{(new } c)(\text{new } d)S \equiv (\text{new } d)(\text{new } c)S
\end{align*}
\]

\[
\begin{align*}
\text{scNil} & : S || [\text{nil}]_b \equiv S
\end{align*}
\]

\[
\begin{align*}
\text{scExt} & : S || (\text{new } c)T \equiv (\text{new } c)S || T \quad \text{if } c \not\in \text{fn}(S)
\end{align*}
\]

Reduction Rules

\[
\begin{align*}
\text{cTIn} & : \begin{cases} b||tt & [\text{if } b \text{ then } P \text{ else } Q]_p \rightarrow [P]_p \quad \text{cElS} & \begin{cases} b||ff & [\text{if } b \text{ then } P \text{ else } Q]_p \rightarrow [Q]_p
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{cCom} & : \begin{cases} c\xi \mu & [c\xi]_p \parallel [c?\chi]_\mu \rightarrow [P][\chi][\xi]_p \quad \text{cPrC} & [K(x) \equiv P]_p \rightarrow [c\xi]_p \parallel [c\xi]_p \rightarrow [P][\chi][\xi]_p
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{cRes} & : \begin{cases} (\text{new } a)S & (\text{new } a)S' \rightarrow S' \quad \text{cPar} & \begin{cases} S \parallel T & S' \rightarrow S' \parallel T \quad \text{cStr} & \begin{cases} S \equiv S' \quad S' \rightarrow T' \quad T' \equiv T
\end{cases}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{cSpl} & : \begin{cases} [P][Q]_{(\text{new } c)} = [P]_p \parallel [Q]_\mu \quad \text{cLcl} & \begin{cases} [\text{new } c]P \rightarrow (\text{new } c)[P][\text{nil}]_{(\text{new } c)} \parallel \text{new } c)\S
\end{cases}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{cTGH} & : \begin{cases} (c, c) \cap \rho \neq \emptyset & [\text{new } c][\text{new } c][P]_p \parallel S \rightarrow [P][\text{nil}]_{(\text{new } c)} || (\text{new } c)S
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{cDisc} & : \begin{cases} \rho \neq \emptyset & [\text{nil}]_p \rightarrow [\text{nil}]_{(\text{new } c)}
\end{cases}
\end{align*}
\]

Figure 2: A Permission-Confined CCS

Process confinement also facilitates the detection of races, which leads to non-deterministic behaviour in the process semantics of Section 2. The judgement \( S \rightarrow_{\text{err}} \), defined by the rules in Figure 2 describes the detection of permission violations. As we shall see later on, the absence of permission violations also implies the absence of channel communication races.

The reduction rules in Figure 2 enforce proper permission usage. Rule cCom imposes additional restrictions to cCom of Figure 2: the output process (resp. the input process) is required to own the permission to output, \( \uparrow c \) (resp. input, \( \downarrow c \)) on channel \( c \). Confined processes cannot arbitrarily create permissions but need to transfer them to other processes at specific interaction points (i.e., communication through cCom). The new rules cSpl and cLcl enforce this resourcing of permissions: cSpl requires that newly spawned processes partition the parent permissions amongst
3.2. Dynamic Properties of Systems. Reductions preserve locality. This means that the permissions owned by a process provide a footprint for its reductions and that any process it reduces to will be confined to these permissions. This property is key for compositional reasoning when ensuring that global properties, such as that of being well-resourced, are preserved. For instance, if the system $S \parallel T$ is well-resourced, then by Definition 3.4 it must be the case that the two sub systems are separate $i.e.$, $S \perp T$. If $S \rightarrow S'$, locality $i.e.$, $\mathsf{prm}(S') \subseteq \mathsf{prm}(S)$ immediately implies that $S' \perp T$ and therefore, that the global system $S' \parallel T$ is still well-resourced. Thus reduction also preserves well-resourcing.

A brief note on some conventions used. To improve readability we have attempted to minimise the use of universal and existential quantifiers in our statements. Thus, unless explicitly stated, free variables introduced to the left of an implication are to be understood as universally quantified, whereas free variables introduced to the right of an implications are understood as existentially quantified.

Lemma 3.5 (Locality). $S \rightarrow T$ implies $\mathsf{prm}(T) \subseteq \mathsf{prm}(S)$

Lemma 3.6 (Resourcing). $\vdash S \text{ and } S \rightarrow T$ implies $\vdash T$

(\textit{Proof for Lemma 3.6 & Lemma 3.5}) The proof is by rule induction on $S \rightarrow T$. The main cases are:

- **cCom:** $S = [c!\overrightarrow{e}l] \parallel [c?\overrightarrow{x}.P]_{\rho \cup \mu}$ and $T = [P][\overrightarrow{y}]_{\rho \cup \mu}$ where $\overrightarrow{e} = \overrightarrow{v}$. It is immediate that $\mathsf{prm}(S) = \mathsf{prm}(T)$. Moreover, $\vdash T$ by wPrc.

- **cPar:** We have $S = R_1 | R_2$ and $T = R_1' | R_2$ and $R_1 \rightarrow R_1'$. Moreover, $\vdash S$ implies $\mathsf{prm}(R_1) \cap \mathsf{prm}(R_2) = \emptyset$, $\vdash R_1$ and $\vdash R_2$. Also recall that $\mathsf{prm}(S) = \mathsf{prm}(R_1) \cup \mathsf{prm}(R_2)$ and that $\mathsf{prm}(T) = \mathsf{prm}(R_1') \cup \mathsf{prm}(R_2)$.

  By $\vdash R_1$, $R_1 \rightarrow R_1'$ and I.H. we obtain $\vdash R_1'$ and $\mathsf{prm}(R_1') \subseteq \mathsf{prm}(R_1)$. By, $\mathsf{prm}(R_1') \subseteq \mathsf{prm}(R_1)$ and $\mathsf{prm}(R_1) \cap \mathsf{prm}(R_2) = \emptyset$ we deduce $\mathsf{prm}(R_1') \cap \mathsf{prm}(R_2) = \emptyset$ and by $\vdash R_1'$ and $\vdash R_2$ we deduce $\vdash T$. Moreover, by $\mathsf{prm}(R_1') \subseteq \mathsf{prm}(R_1)$ we obtain $\mathsf{prm}(T) \subseteq \mathsf{prm}(S)$.

- **cSfl:** $S = [P][\overrightarrow{Q}]_{\rho \cup \mu}$ and $T = [P]_{\rho \cup \mu}$. $\rho \cup \mu$ implies $\mathsf{prm}(P)_{\rho} \cap \mathsf{prm}(Q)_{\mu} = \emptyset$ and since $\vdash [P]_{\rho}$ and $\vdash [Q]_{\mu}$ (by wPrc), we get $\vdash T$. Moreover $\mathsf{prm}(T) = \mathsf{prm}(S)$.

- **cDsc:** $S = [\text{nil}]_{l}$ and $T = [\text{nil}]_{l}$. Trivially, $\vdash T$ (by wPrc) and $\mathsf{prm}(T) = \emptyset \subseteq \mathsf{prm}(S)$. \qed
Another important property of our resource semantics is that reductions do not hide prior permission violations i.e., permission violations are preserved by reductions. This allows us to ignore intermediary steps during the evaluation of a confined process (cf. Definition 3.8) and simply inspect the resulting stable system to determine whether that evaluation resulted in any permission violations.

**Lemma 3.7 (Violation Preservation).** \( S \rightarrow^* T \) and \( S \rightarrow_{err} \) implies \( T \rightarrow_{err} \)

**Proof.** First we show \( S \rightarrow^* T \) and \( S \rightarrow_{err} \) implies \( T \rightarrow_{err} \) by rule induction on \( S \rightarrow^* T \). The main cases are:

- **cCom:** \( S = [c!\overrightarrow{c}]_\rho \parallel [\overrightarrow{c}?x.P]_\mu \) where \( \uparrow c \in \rho \) and \( \downarrow c \in \mu \). By case analysis, if \( S \rightarrow_{err} \) then either \([c!\overrightarrow{c}]_\rho \rightarrow_{err}\) because \( \uparrow c \notin \rho \) (by eOut) or \([\overrightarrow{c}?x.P]_\mu \) because \( \downarrow c \notin \mu \) (by eIn); both cases lead to a contradiction.

- **cPar:** \( S = R_1 \parallel R_2 \), \( T = R_1' \parallel R_2' \) and \( R_1 \rightarrow R_1' \) and \( R_2 \rightarrow R_2' \). By \( S = R_1 \parallel R_2 \), ePar, eStr and scCom we know \( S \rightarrow_{err} \) because either:
  - \( R_1 \rightarrow_{err} \) By \( R_1 \rightarrow R_1' \) and I.H. \( R_1' \rightarrow_{err} \) and by \( T = R_1' \parallel R_2' \) and ePar we get \( T \rightarrow_{err} \).
  - \( R_2 \rightarrow_{err} \) By \( T = R_1' \parallel R_2, ePar, eStr \) and scCom we obtain \( T \rightarrow_{err} \).

The second part of the proof is by induction on the number \( n \) of reductions used i.e., \( S \rightarrow^n T \). \( \Box \)

### 3.3. System Determinism.

The first two main results of our resource semantics establish that system evaluation is deterministic up-to the terminal permissions owned (cf. Theorem 3.11 and Theorem 3.12).

We first lay the ground for these results by giving the following definitions. Systems evaluation in Definition 3.8, \( S \downarrow T \), is limited to safe-stability, \( T \triangledown \), and excludes reductions to racy systems. The operation \(|-|\) denotes a permission-erasure function whereby \(|S|\) returns the process in \( S \) stripped of all its confining permissions; it allows us to express equivalence up-to owned permissions in Theorem 3.11 System Convergence. Definition 3.10 is the least set of systems that converge to a stable state (but not necessarily a safe one) and is used for Theorem 3.12.

**Definition 3.8 (Safety-Stability and Evaluation).**

\( S \triangledown \overset{def}{=} S \rightarrow \) and \( S \rightarrow_{err} \) implies \( S \downarrow T \overset{def}{=} \exists T'. S \rightarrow^* T' \) and \( T' \triangledown \) and \( T \equiv T' \)

**Definition 3.9 (Permission Confinement Erasure).**

\[ |S| \overset{def}{=} \begin{cases} P & \text{if } S = [P]_\rho \\ |T| \parallel |R| & \text{if } S = T \parallel R \\ (\text{new } c) |T| & \text{if } S = (\text{new } c) T \end{cases} \]

**Definition 3.10 (System Convergence).** \( S \downarrow T \overset{def}{=} S \rightarrow \) or (\( \forall T. S \rightarrow^* T \) implies \( T \downarrow \))

In conformance with Definition 2.6 by system determinism we understand that (1) no system can evaluate to two distinct safely-stable systems, up-to owned permissions i.e., Theorem 3.11 and that (2) no system can evaluate to a safely-stable system and, at the same time, diverge along a different execution path i.e., Theorem 3.12.

**Theorem 3.11 (Evaluation Determinism).** \( S \downarrow T_1 \) and \( S \downarrow T_2 \) implies \( |T_1| \equiv |T_2| \)

**Theorem 3.12 (System Evaluation implies System Convergence).** \( S \downarrow \) implies \( S \downarrow \)
These properties follow, at an intuitive level, from the partial-confluence property, as stated in Lemma 3.13.

**Lemma 3.13** (Partial Confluence). $S \rightarrow T_1$ and $S \rightarrow T_2$ implies either of the following:

1. $|T_1| \equiv |T_2|$ or;
2. $\exists T_3. T_1 \rightarrow T_3$ and $T_2 \rightarrow T_3$

However, the full technical details of the proofs for Theorem 3.11 and Theorem 3.12 are more delicate; on first reading, the reader may skip them and progress to Section 3.4. Before though, we highlight Proposition 3.14, which establishes sufficient and necessary conditions on the structure of safely-stable systems; these conditions will then act as a guiding principle when formulating our logic formulas. In essence, safely stable systems consist of mismatching asynchronous outputs and input-blocked processes composed in parallel, each owning the respective output and input permissions so as not to generate an error.

**Proposition 3.14** (Safe-Stability and System Structure).

\[
S \not\approx T \iff S \equiv (\text{new } \bar{d}) \left( \|_{i=0}^n [c_i] \bar{e}_i \|_{\rho_i} \|_{j=0}^m [c_j] \bar{x}_j, P \|_{\mu_j} \right)
\]

where

- $\{c_1, \ldots, c_n\} \cap \{c'_1, \ldots, c'_m\} = \emptyset$
- $\bigwedge_{i=0}^n c_i \in \rho_i$
- $\bigwedge_{j=0}^m c_j \in \mu_j$

and where $\|_{i=0}^n [c_i] \bar{e}_i \|_{\rho_i}$ and $\|_{j=0}^m [c_j] \bar{x}_j, P \|_{\mu_j}$ denote $\| \bar{nil}_0 \|_{\emptyset}$.

The proofs for Theorem 3.11 and Theorem 3.12 require us to work at a tighter relation than process structural equivalence for the intermediary steps of an evaluation, namely $\equiv$ defined in Definition 3.15, because process structural equivalence, $\equiv$, loses information wrt. the currently owned permissions of a system. The relation $\approx$ lies between system structural equivalence and the respective process structural equivalence after confinement erasure (cf. Proposition 3.16).

**Definition 3.15** (Equivalence up-to owned permissions). $S \equiv T$ is defined as the least relation satisfying the following rules:

<table>
<thead>
<tr>
<th>$[P]<em>{\rho} \equiv [P]</em>{\mu}$</th>
<th>$S_1 \equiv S_2 \quad T_1 \equiv T_2$</th>
<th>$S_1 \equiv S_2 \quad (\text{new } c)S_1 \equiv (\text{new } c)S_2$</th>
<th>$S_1 \equiv S_2 \equiv T_2 \equiv T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1</td>
<td></td>
<td>T_1 \equiv S_2</td>
<td></td>
</tr>
</tbody>
</table>

**Proposition 3.16.** $S \equiv T$ implies $S \equiv T$ implies $|S| \equiv |T|$

Note that $|S| \equiv |T|$ does not imply $S \equiv T$. For instance, $|[P] || \bar{Q} || \rho \equiv |[P] || \bar{Q} || \mu |$ but $|[P] || \bar{Q} || \rho \equiv \neq |P || \bar{Q} || \mu |$. $[P] || \bar{Q} || \rho \equiv \neq |P || \bar{Q} || \mu |$.

**Lemma 3.17** (Properties of $\equiv$ with respect to reductions).

1. $S \equiv T$ and $T \rightarrow T'$ implies $S \rightarrow S'$ and $S' \equiv T'$
2. $S \equiv T$ and $S/ \rightarrow$ implies $T/ \rightarrow$

**Proof.** See Appendix A.2
The system relation ≊ allows us to specify a tighter relationship which characterises more precisely Partial Confluence, i.e., Lemma 3.18. This is then used to prove Lemma 3.21 upon which Theorem 3.11 rests. We here relegate the proofs of Lemmas used by Lemma 3.21 to Appendix A.2. Note also that Lemma 3.13, stated earlier to give an intuition for how linear permissions ensure confluence, follows immediately from Lemma 3.18 and Proposition 3.16.

**Lemma 3.18 (Partial Confluence).** $S \rightarrow T_1$ and $S \rightarrow T_2$ implies either of the following:

1. $T_1 \equiv T_2$ or;
2. $\exists T_3. T_1 \rightarrow T_3$ and $T_2 \rightarrow T_3$

**Proof.** See Appendix A.2.

**Definition 3.19 (System Evaluation Predicates).** $S \downarrow \overset{\text{def}}{=} \exists T. S \downarrow T$

**Lemma 3.20 (Evaluation Preservation for ≊).**

$S \equiv T$ and $S \downarrow$ and $T \rightarrow T'$ implies $S \rightarrow S'$ where $S' \equiv T'$ and $S' \downarrow$

**Proof.** See Appendix A.2.

**Lemma 3.21 (Evaluation and ≊).**

$S \equiv T$ and $S \rightarrow^n S' \checkmark$ and $T \rightarrow^m T' \checkmark$ implies $S' \equiv T'$ and $n = m$

**Proof.** By (strong) induction on the number of reductions leading to a safely-stable system from any system $S \rightarrow^n S'$.

$n = 0$: By $S \not\rightarrow$ and Lemma 3.17(2) we know $T \not\rightarrow$ which implies $m = 0$ and $T' = T \equiv S$.

$n = k + 1$: We have

\[ \exists S'' \text{ such that } S \rightarrow S'' \rightarrow^k S' \quad (3.1) \]

Lemma 3.7 and $S' \checkmark$, $T' \checkmark$ implies

$S \not\rightarrow_{\text{err}}$ and $T \not\rightarrow_{\text{err}}$

and $S \rightarrow S''$ and Lemma 3.17(1) implies that $m > 0$ i.e.,

$\exists T''$ such that $T \rightarrow T'' \rightarrow^{m-1} T'$ \quad (3.2)

Moreover, $S \rightarrow^n S' \checkmark$ and $T \rightarrow^m T' \checkmark$ imply $S \parallel S'$, $T \parallel T'$ respectively, and by $S \equiv T$, $S \rightarrow S''$ and Lemma 3.20 we obtain

$\exists T_1, T_1', l$ such that $T \rightarrow T_1$

$T_1 \equiv S''$ \quad (3.3)

$T_1 \rightarrow^l T_1' \checkmark$ \quad (3.4)

By $S'' \rightarrow^k S'$ from (3.1), (3.4), (3.5) and I.H. we obtain

$S' \equiv T_1'$ and $l = k$ \quad (3.6)

i.e., $T_1 \rightarrow^k T_1' \checkmark$. Now by Lemma 3.18 (3.3) and $T \rightarrow T''$ from (3.2) we have two sub-cases:

$T_1 \equiv T''$: By (3.5) and (3.6) we know $T_1 \rightarrow^k T_1' \checkmark$ and by, $T'' \rightarrow^{m-1} T'$ from (3.2) I.H. we deduce

$T' \equiv T_1'$ and $(m - 1) = k$

and by transitivity and (3.6) we conclude $T' \equiv S'$ and $m = (k + 1) = n$ as required.
∃T₃, T₁ → T₃ and T’’ → T₃: We here have two further sub-cases:

∃T₃', h such that T₃ →ₕ T₃': This implies T₁ →ₕ₊₁ T₃' and by (3.1), (3.4) and I.H. we obtain

T₃' ≊ S' and (h + 1) = k

(3.7)

We also know that T’’ →ₕ₊₁ T₃' and by (3.7) we obtain T’’ →ₖ T₃' and, since T’’ ≊ T’’ (reflexivity of ≊), using (3.2) and I.H. we obtain

T₃' ≊ T' and (m - 1) = k

which, first implies m = (k + 1) = n and then, by (3.7), implies T' ≊ S' as required.

T₃ $\not\rightarrow$: By T₁ → T₃, T₁ ≊ T₁ (reflexivity of ≊), (3.5) and Lemma 3.20 we know

∃T₄, T₄', i such that T₁ → T₄

(3.8)

T₄ ≊ T₃

(3.9)

T₄ →ᵢ T₄'

(3.10)

Similarly, by T’’ → T₃, T’’ ≊ T’’', (3.2), T' and Lemma 3.20

∃T₅, T₅', j such that T’’ → T₅

(3.11)

T₅ ≊ T₃

(3.12)

T₅ →ᵢ T₅'

(3.13)

Now (3.8) and (3.10) imply T₁ →ᵢ₊₁ T₄' and by (3.6), T₁ ≊ T₁ and I.H. we obtain

T₄' ≊ T₁' ≊ S' and (i + 1) = k i.e., T₄ →ᵢ₋₁ T₄'

(3.14)

Moreover, (3.9),(3.12) and transitivity imply T₄ ≊ T₅, and by (3.14), (3.13) and I.H. we obtain

T₅' ≊ T₅' ≊ S' and j = (k - 1)

(3.15)

By (3.11) and (3.15) we obtain T’’ →ₖ T₅' and by T’’ ≊ T’’, (3.2) and I.H. we obtain

T' ≊ T₅' ≊ S' and (m - 1) = k

which also implies m = (k + 1) = n as required.

Theorem 3.11 (Evaluation Determinism). S $\parallel T₁$ and S $\parallel T₂$ implies |T₁| ≡ |T₂|

Proof. By reflexivity we know S ≊ S and by Lemma 3.21 we know T₁ ≊ T₂ which, by Proposition 3.16 implies |T₁| ≡ |T₂|.
Convergence for systems, Theorem 3.12, largely follows from Lemma 3.20 and Lemma 3.21. We prove Theorem 3.12 by generalising the hypothesis to systems related by $\equiv$ in Lemma 3.22, so as to make the induction go through.

**Lemma 3.22.** $S \downarrow$ and $S \equiv T$ implies $T \downarrow$.

**Proof.** By induction on $n$ where $S \rightarrow^n R/ \diamond$ for some witness safely-stable $R$ justifying $S \downarrow$.

$n = 0$: This means that $S \rightarrow$ and thus by Lemma 3.17(2) we have $T \rightarrow$ which implies $T \downarrow$.

$n = k + 1$: We have

$$S \rightarrow S' \rightarrow^k R/ \diamond \quad (3.16)$$

We have two sub-cases. If $T \rightarrow$ then this trivially implies convergence. Otherwise, if $T \rightarrow T'$, by Lemma 3.20 we obtain

$$S \rightarrow S''$ such that $S'' \equiv T'$ and $S'' \downarrow \quad (3.17)$$

$S'' \downarrow$ implies that for some $m$ and $R'$, $S \rightarrow^m R/ \diamond$, and since $S \equiv S$, by (3.16) and Lemma 3.21 this implies that $m = k + 1$ which means that $S'' \rightarrow^k R'$. Thus by $S'' \equiv T'$ from (3.17) and I.H. we obtain $T' \downarrow$ which implies $T \downarrow$. $\square$

**Theorem 3.12** (System Evaluation implies System Convergence). $S \downarrow$ implies $S \downarrow$

**Proof.** Immediate by Lemma 3.22 and $S \equiv S$. $\square$

3.4. **Process Determinism.** The second main batch of results relate system evaluations in our resource semantics with process determinism in the unconstrained semantics of Section 2 (cf. Corollary 3.25). In particular, Theorem 3.23 states that any well-resourced permission allocation $S$ that allows a process $|S|$ to evaluate down to a safely-stable system, $T$, implies that any evaluation for process $|S|$ - in the unconstrained semantics - corresponds, up to structural equivalence, to this system $T$ stripped off its constraining permissions i.e., $|T| \equiv Q$ whenever $|S| \downarrow Q$. On the other hand, Theorem 3.24 states that if $S$ evaluates successfully to a safely-stable process, then the corresponding process $|S|$ must be convergent. Together, these two theorems effectively state that finding a single allocation (narrative) $S$ of linear permissions for a process $|S|$ that allows it to evaluate to some $T$ suffices to show that $|S|$ is deterministic in the unconstrained semantics (Corollary 3.25).

**Theorem 3.23** (Process Evaluation Determinism). $S \downarrow T$, $|S| \downarrow Q_1$, $|S| \downarrow Q_2$ implies $Q_1 \equiv Q_2 \equiv |T|$

**Theorem 3.24** (Process Convergence). $S \downarrow$ implies $|S| \downarrow$

**Corollary 3.25.** $S \downarrow$ implies $|S|$ is deterministic.

**Proof.** By Definition 2.6 Theorem 3.23 and Theorem 3.24. $\square$
We next discuss in detail the proofs for Theorem 3.23 and Theorem 3.24; the reader may safely skip them on first reading and proceed to Section 3.5.

Theorem 3.23 follows directly from Lemma 3.31, which in turn relies heavily on Lemma 3.28. In essence, this lemma states that a system that evaluates to a safely stable system can match any sequence of reductions (in the unconstrained semantics) of the system stripped of its constraining permission. This lemma is based on Lemma 3.27, which proves the property for the case of a single unconstrained reduction, and also depends on the property of corrective reductions, Lemma 3.26. This lemma states that any system that can evaluate safely, \( S \downarrow \), is guaranteed to be able to “correct” wrong partitioning of permissions (cf. cStr in Figure 2) along a particular reduction path that results in systems that cannot evaluate safely. Stated otherwise, this means that there must exist a permission partition that leads to a full evaluation along that particular execution path.

**Lemma 3.26** (Corrective Reductions).

\[ S \downarrow \land S \longrightarrow^n T \quad \text{and} \quad T \not\downarrow \implies \exists R \text{ such that } S \longrightarrow^n R \text{ and } R \equiv T \text{ and } R \downarrow \]

*Proof.* Immediate from Lemma A.9 from Appendix A.2 and the fact that \( S \equiv S \).

**Lemma 3.27** (Reduction Correspondence).

\[ S \downarrow \land |S| \longrightarrow Q \implies \exists R \text{ such that } S \longrightarrow^+ R \text{ and } |R| \equiv Q \]

*Proof.* By rule induction on \( |S| \longrightarrow Q \); see Appendix A.2.

**Lemma 3.28** (Multi-step Reduction Correspondence).

\[ |S| \longrightarrow^n Q \land S \downarrow \implies S \longrightarrow^{n+m} R \text{ such that } R \downarrow \land |R| \equiv Q. \]

*Proof.* Proof by induction on the number of reduction steps that lead to a stable process \( |S| \longrightarrow^n Q \):

- \( n = 0 \): Immediate since \( Q = |S| \) and \( S \longrightarrow^0 S \) where \( S \downarrow \).
- \( n = k + 1 \): This means that \( \exists \ P \text{ such that } |S| \longrightarrow P \longrightarrow^k Q \). By \( S \downarrow \land \text{Lemma 3.27} \) we know:
  \[ \exists T \text{ such that } S \longrightarrow^l T, \ l > 0 \text{ and } |T| \equiv P \] \hspace{1cm} (3.18)
  Thus by \( P \longrightarrow^k Q \), \( |T| \equiv P \) from (3.18) and \( \text{rStr} \) we have
  \[ |T| \longrightarrow^k Q \] \hspace{1cm} (3.19)

  At this point we have two cases:

  **\( T \downarrow \):** By I.H. implies we deduce that \( T \longrightarrow^{k+m} R \) such that \( R \downarrow \) and \( |R| \equiv Q \), and by \( S \longrightarrow^l T \) from (3.18) we obtain
  \[ S \longrightarrow^{k+m+l} R \text{ such that } R \downarrow \text{ and } |R| \equiv Q. \]

  **\( T \not\downarrow \):** By \( S \longrightarrow^l T \) from (3.18) and \( \text{Lemma 3.26} \) we know
  \[ \exists T' \text{ such that } S \longrightarrow^l T' \text{ and } T' \equiv T \text{ and } T' \downarrow \] \hspace{1cm} (3.20)
  Now, by Proposition 3.16 \( T' \equiv T \) and implies \( |T'| \equiv |T| \). Thus by (3.19) and \( \text{rStr} \) we deduce \( |T'| \longrightarrow^k Q \). Thus by \( T' \downarrow \land \text{I.H.} \) we obtain \( T' \longrightarrow^{k+m} R \) such that \( R \downarrow \) and \( |R| \equiv Q \), and by \( S \longrightarrow^l T' \) from (3.20) we obtain \( S \longrightarrow^{k+m+l} R \) such that \( R \downarrow \) and \( |R| \equiv Q \). \( \square \)

**Lemma 3.31** uses also Lemma 3.30 which maps stable processes to safely stable systems.

**Lemma 3.29** (Correspondence).

\[ S \longrightarrow T \implies |S| \longrightarrow |T| \text{ or } |S| \equiv |T| \]

*Proof.* The proof is by rule induction on \( S \longrightarrow T \) and relegate this to Appendix A.2.
Lemma 3.30 (Correspondence and Termination). $|S| \rightarrow$ and $S \parallel T$ implies $|T| \equiv |S|$

Proof. By induction on $n$ where $S \rightarrow^n T$. The inductive case uses the contrapositive of Lemma 3.29. See Appendix A.2.

Lemma 3.31 (Evaluation Determinism). $|S|\parallel Q$ and $S \parallel T$ implies $Q \equiv |T|$.

Proof. $|S|\parallel Q$ implies that

$$|S| \rightarrow^n Q \rightarrow$$ for some $n$ (3.21)

By $|S| \rightarrow^n Q$, $S \parallel T$ and Lemma 3.28 we know that $S \rightarrow^{n+m} R$ such that $R\parallel$ and $|R| \equiv Q$. Since $Q \rightarrow$, (3.21), then by Corollary A.2 we obtain $|R| \rightarrow$ and thus, by $R\parallel$ and Lemma 3.30 we know that

$$R \parallel T' \text{ for some } T' \text{ where } |T'| \equiv |R|$$ (3.22)

By $S \rightarrow^{n+m} R$ and $R\parallel T'$ of (3.22) we deduce that $S \parallel T'$ and by $S \parallel T$ and Theorem 3.11 from Section 3.3 we know $|T| \equiv |T'|$. Thus by transitivity we obtain $|T| \equiv |T'| \equiv |R| \equiv Q$ as required.

Theorem 3.23 (Process Evaluation Determinism). $S \parallel T$, $|S|\parallel Q_1$, $|S|\parallel Q_2$ implies $Q_1 \equiv Q_2 \equiv |T|

Proof. By Lemma 3.31 we know $Q_1 \equiv |T|$ and $Q_2 \equiv |T|$ and the required result follows by transitivity of $\equiv$.

The theorem relating system evaluation and process convergence uses the following corollary, obtained from Proposition 3.14 of Section 3.3.

Corollary 3.32. $S \rightarrow_{err}$ and $S \rightarrow$ implies $|S| \rightarrow$


Theorem 3.24 (Process Convergence). $S \parallel$ implies $|S| \downarrow$

Proof. By contradiction. Assume that $|S| \uparrow$. Since, by $S \parallel$ and Theorem 3.12 any reduction sequence starting from $S$ is finite, by $|S| \uparrow$ there must exist a long enough reduction sequence

$$|S| \rightarrow^n Q \rightarrow \ldots$$

where, by Lemma 3.28, $S \parallel T$ and $|T| \equiv Q$. Now since $T \downarrow$, then by Corollary 3.32 we must have $Q \rightarrow$ which contradicts our assumption. Thus $|S| \downarrow$.

3.5. Confined Semantics Application. The following examples expound on the use of linear permission allocations for reasoning about deterministic code.

Example 3.33. $Prg \parallel c_1!v_1 \parallel c_2!v_2$ can be shown to be deterministic by finding a permission assignment for every process below that permits a safe evaluation.

$$[Prg]_{\rho_1} \parallel [c_1!2]_{\rho_2} \parallel [c_2!5]_{\rho_3} \downarrow [c_1!(2,4)]_{\mu_1} \parallel [c_4!]_{\mu_2}$$

Two possible assignments for $\rho_1$, $\rho_2$ and $\rho_3$ that permit the above evaluation are:

$$\rho_1 = \{\downarrow c_1, \downarrow c_2, \uparrow c_4\}, \quad \rho_2 = \{\uparrow c_1\}, \quad \rho_3 = \{\uparrow c_2\} \quad \text{or;} \quad (3.23)$$

$$\rho_1 = \{\downarrow c_1, \downarrow c_2\}, \quad \rho_2 = \{\uparrow c_1, \uparrow c_4\}, \quad \rho_3 = \{\uparrow c_2\} \quad (3.24)$$

Stated otherwise, we have at least two possible linear-permission based narratives explaining why $Prg \parallel c_1!v_1 \parallel c_2!v_2$ is deterministic. For both assignments $\uparrow c_1 \in \mu_1$ and $\uparrow c_4 \in \mu_2$ must hold for the resulting safely-stable system $[c_1!(2,4)]_{\mu_1} \parallel [c_4!]_{\mu_2}$, but the remaining permissions $\downarrow c_1$, $\downarrow c_2$ and
For the most part, we have abstracted away from structural manipulation of terms, with the exception that the permission associated with it.

Using the permission assignment in \((3.23)\) we can have the reduction sequence below. Reduction \((3.25)\) can be derived using the rules cLct, cStr and cPar from (cf. Figure 2) whereas reduction \((3.26)\) is derived using cStr, cPar and cRes; other reductions can be derived in similar fashion. For the most part, we have abstracted away from structural manipulation of terms, with the exception of reduction \((3.25)\) which employs cTat and cStr to discard the redundant scoped channel name \(c_3\) and the permissions associated with it.

\[
\begin{align*}
\text{Prg} & \triangleq (\text{new } c_3)\left(\text{Fltr}\|\text{Dbf}\right) \\
\text{Fltr} & \triangleq c_1!x_1, \text{if } x_1 \leq 9 \text{ then } c_3!x_3, (c_1!(x_1, x_3) \parallel c_4!) \text{ else } c_4!x_1
\end{align*}
\]

\((3.25)\)
From a compositional perspective, permission-sets also delineate the footprint of every process and, indirectly, the requirement for well-resourcing of Definition 3.4 defines an interface for detecting race conditions. Consider for example the system:

\[
\left[ P_{rg} \right]_{[c_1, c_2, \top c_4]}
\]

In order for this system to be safe, it needs the permission \( \downarrow c_1 \) (otherwise it would yield a permission violation through rule \( \text{eln} \)). Recall the context \( c_1 !1 \parallel c_2 !v_2 \parallel c_1 ?x \cdot \text{nil} \) from Example 2.7 which had introduced a race condition on inputs on channel \( c_1 \). In order for this system not to violate permissions itself, it must own a permission set \( \mu \), i.e.,

\[
\left[ c_1 !1 \parallel c_2 !v_2 \parallel c_1 ?x \cdot \text{nil} \right]_{\mu}
\]

where \( \downarrow c_1 \in \mu \) as well. However, the separation condition for well-resourcing prohibits us from composing these two systems together because their respective permissions are not disjoint i.e., \( \{ \downarrow c_1, \downarrow c_2, \top c_4 \} \not\subseteq \mu \).

**Example 3.34.** If, in the array \( a_1 !v_1 \parallel \ldots \parallel a_n !v_n \) to be sorted, we assign the permission set \( \mu_i = \{ \uparrow a_i \} \) to every element \( a_i !v_i \) and assign the permission set \( \rho = \{ \downarrow a_1, \ldots, \downarrow a_n, \top r \} \) to \( Qck(1, n) \) then it turns out that we can show that

\[
\left[ Qck(1, n) \right]_{\rho} \parallel \left[ a_1 !v_1 \right]_{\mu_1} \parallel \ldots \parallel \left[ a_n !v_n \right]_{\mu_n} \parallel \left[ r ! \right]_{\rho}
\]

for some safely stable system \( T \) where

\[
T = [a_1 !u_1]_{\mu_1} \parallel \ldots \parallel [a_n !u_n]_{\mu_n} \parallel [r !]_{\rho}
\]

Note how, as in Example 3.33, \( \rho \) in \( [Qck(1, n)]_{\rho} \) defines an interface that parallel processes to be composed with it to respect, in order for it to evaluate deterministically.

### 3.6. Discussion

Process spawning, \( cSpl \), is intentionally non-deterministic: apart from alleviating permission annotation\(^3\), its non-deterministic nature is in line with the unspecified way that permissions can be allocated in a confined system. Correspondingly, through Theorem 3.11 and Corollary 3.25, we have seen how there may be more than one way how to validly distribute permissions across processes so as to prove determinacy.

Since we eventually plan to use confined processes as part of the model for our logic (cf. Section 4), we here opt for the most flexible solution i.e., non-deterministic splits for parallel composition, which permits more narratives explaining process determinism while still restricting the permission allocations that can be used. This setup gives better separation of concerns between confined process reduction and the model used for our logic. In particular, this model incorporates environments describing permission-transfer invariants, apart from confined processes. These environments are however orthogonal to the properties of confined processes derived in this section. In fact, their purpose is that of allowing for better compositional analysis when determining assertion satisfactions, as we shall see in Section 4 and Section 5.

\(^3\) The current formulation leads to a more lightweight form of annotation for confined processes. The other alternative would have been to extend the definition of parallel composition at the process level and have systems of the form \( [P \parallel \mu_1, \mu_2]Q \), whereby \( \mu_1 \) and \( \mu_2 \) specify deterministically how \( \rho \) is to be apportioned amongst \( P \) and \( Q \).
4. Logic

We define a separation-based logic that enables us to reason about programmes that deterministically evaluate to stable systems satisfying assertions describing their state. Our logic concentrates more on describing data held at asynchronous outputs in stable systems, and abstracts away from issues dealing with control for deterministic evaluation. For this reason, the logic semantics is not defined directly on bare processes. Instead, the confined processes of Section 3 together with the definitions for safe-stability and evaluations, Definition 3.8, provide the basis for a model to our separation logic whereby the permissions owned constitute our units of separation (cf. Definition 3.3). Together with the associated proof system of Section 5, this amounts to our proposal for a logical framework for reasoning over non-interfering concurrent programmes.

4.1. Permission Environments. In our logic, channels have a dual role. Apart from acting as a mechanism for communicating data, they also act as delimiters of mutual-exclusion groups of resources, modeling condition-critical regions. Each input process \( c?\vec{x}.P \) abides to use certain permissions in \( P \) only after it synchronises on channel \( c \) whereas each output-process \( c!\vec{e} \) obliges to own the permissions guarded by \( c \); these guarded permissions are transferred dynamically upon communication on \( c \) using rule \( cCom \) of Figure 2 and enable us to reason about channel reuse in deterministic systems.

The invariants relating to permission mutual-exclusion are characterised as permission environments, \( \Gamma \in \text{Chans} \rightarrow \mathcal{P}(\text{Perm}) \), partial maps associating channels \( c \) to permission-sets \( \rho \). They require abiding processes to own all the permissions in \( \rho \) when outputting on \( c \) and, dually, allow processes to assume the acquisition of all permissions in \( \rho \) when inputting on \( c \). The constraints in Definition 4.1 ensure that (1) permission transfer always includes the permission \( \uparrow c \) to output over the communicating channel, but never the capability \( \downarrow c \) to input over it, as this must already belong to the receiving process; (2) environments are suitably closed.

**Definition 4.1 (Permission Environment).** \( \Gamma \) is a finite map from names to permission sets such that:

1. For all \( c \in \text{dom}(\Gamma) \), \( \downarrow c \notin \Gamma(c) \) and \( \uparrow c \in \Gamma(c) \),
2. \( \rho \in \text{cod}(\Gamma) \) implies \( \text{nm}(\rho) \subseteq \text{dom}(\Gamma) \),

where \( \text{nm}(\rho) \overset{\text{def}}{=} \{ c | \downarrow c \in \rho \text{ or } \uparrow c \in \rho \} \).

4.2. Logical Formulas. Our logic formulas, ranged over by the meta-variables \( \varphi, \psi \), characterise a ‘spatial’ notion of state for deterministic processes in terms of the data held on asynchronous channels at stable processes. In order to simplify our conceptual process interpretations, we limit ourselves to describing only the states of stable processes, abstracting away from the intermediary reductions that lead to stability. For this we require asynchronous output data assertions, \( c(\vec{e}) \), the ‘separated conjunction’, \( \varphi \ast \psi \), and its unit, \( \text{emp} \); formulas constructed using just these constructs are denoted by the metavariable \( \chi \) and are called state formulas. Guided by Proposition 3.14, stability requires our formulas to describe (input) blocked processes, \( \text{blk}(c) \). Finally, we also describe unrestricted terminating process by \( \text{any} \) whenever we want to abstract away completely from the structure of a terminating process.

**Definition 4.2 (Formulas).**

\[
\begin{align*}
\chi, \eta \in \text{SFrM} & \colon \coloneqq \text{emp} \mid c(\vec{e}) \mid \chi \ast \chi \\
\varphi, \psi \in \text{Frm} & \colon \coloneqq \text{emp} \mid \text{any} \mid c(\vec{e}) \mid \text{blk}(c) \mid \varphi \ast \psi
\end{align*}
\]
Our formulas are interpreted over permission environments and well-formed systems, i.e., \( \Gamma, S \models \phi \). They are defined in Figure 3, inductively on the structure of closed formulas i.e., formulas with no free variables in the expressions \( \vec{e} \) of \( \langle \vec{e} \rangle \). Our definition of formula satisfaction relies heavily on the evaluation judgement, \( S \downarrow T \), which is only defined for closed systems (Definition 3.8); recall that system evaluation existentialises over a reduction path leading to a stable system.

The satisfaction relation in Figure 3 describes the state of a system once it stabilises. The main assertion satisfaction is that for data assertions, \( \langle \vec{e} \rangle \), as it relates the data held on asynchronous outputs of a stable system with the data stated in the assertion. To do this, the definition relies on the assumption that \( S \) is closed to establish the equality between the two expressions \( \vec{e} \) and \( \vec{e}' \). Moreover, it uses the environment, \( \Gamma \), to ensure that the (stable) asynchronous output owns the permissions imposed by the permission guarding invariants. Its use has already been discussed in Section 4.1 and will be elaborated further when we consider compositional analysis of satisfaction in Section 5.

Data assertions are typically composed together using the separating conjunction assertion, \( \phi_1 \ast \phi_2 \), and the empty assertion, \( \text{emp} \). For the satisfaction for \( \text{emp} \), the system \( \langle \text{nil} \rangle \) is chosen to be the identity interpretation for our model wrt. separation, thereby making the interpretation for just these constructs a commutative monoid (cf. Lemma 4.9).

The satisfaction definition of the separating conjunction, \( \phi_1 \ast \phi_2 \), is however more complicated than one would have expected, as it needs to handle conjunctions with \( \text{blk}(c) \) and \( \text{any} \) formulas as well; the interpretation for the latter two formulas is rather straightforward. Thus, apart from relying on the system well-resourcing assumption to guarantee that the partitioned sub-systems are separate, \( S_1 \perp S_2 \) (cf. Definition 3.3), satisfaction for the separating conjunction also enforces that a system is stable before it is split, i.e., \( S \downarrow S_1 \parallel S_2 \). This condition rules out systems whose subcomponents satisfy the sub-formulas of a conjunction \( \phi_1 \ast \phi_2 \), but then violate stability once composed together; we return to this later in Example 4.4. The fact that separating conjunction ranges over input-blocked processes also requires a satisfaction definition that ignores scoping of channel names across separation i.e., \( S \downarrow (\text{new} d) (S_1 \parallel S_2) \); these scoped names \( d \) refer to channels used in the continuations of blocked processes, as explained later in Example 4.3, and cannot be abstracted away using structural equivalence rules such as \( \text{scExt} \) and \( \text{scNew} \) from Figure 2.

Figure 3: Formula Satisfaction
Example 4.3 (Satisfiability). Recall the process definitions

\[ \text{Pr} \triangleq (\text{new } c) (\text{Fltr} \parallel \text{Dbl}) \]
\[ \text{Dbl} \triangleq c_2 ? x_2 . c_3 ? x_4 . c_1 !(x_4 + x_4) \]
\[ \text{Fltr} \triangleq c_1 ? x_1 . \text{if } x_1 \leq 9 \text{ then } c_3 ! x_1 \parallel c_1 ? x_3 . (c_1 !(x_1, x_3) \parallel c_4 ! x_1) \text{ else } c_4 ! x_1 \]

from Example 2.7. Assuming the environment
\[ \Gamma = c_1 : \{\uparrow c_1\}, \ c_2 : \{\uparrow c_2\}, \ c_4 : \{\uparrow c_4, \downarrow c_1\} \]

we have the following satisfactions:

\[ \Gamma, [\text{Pr}][c_1, c_2, c_4] \parallel [c_1 !2][c_2, c_4] \parallel [c_2 !5][c_2] \models c_1 (2, 4) * c_4 \] \hspace{1cm} (4.1)
\[ \Gamma, [\text{Pr}][c_1 !2 \parallel c_2 !5][c_1, c_2, c_4, c_1, c_1, c_2] \models c_1 (2, 4) * c_4 \] \hspace{1cm} (4.2)
\[ \Gamma, [c_1 !(5 - 3, 3 + 1)][c_1] \parallel [c_4 !][c_4] \models c_1 (2, 4) * c_4 \] \hspace{1cm} (4.3)

whereby, according to the definition in Figure 3, satisfaction is only concerned with the existence of a reduction path to a stable system, where the outputs corresponding to data assertions are required to own the permissions expected by permission environment \(\Gamma\); the reduction path (4.1) and (4.2) has already been discussed in Example 3.33. Satisfaction for (4.3) is more straightforward to determine as the system is stable. On the other hand, for \(\Gamma\) defined above, the following do not satisfy their respective assertions:

\[ \Gamma, [\text{Pr}][c_1, c_2, c_4] \parallel [c_1 !2][c_2, c_4] \parallel [c_2 !5][c_2] \not\models c_1 (2, 4) * c_4 \] \hspace{1cm} (4.4)
\[ \Gamma, [\text{Pr}][c_1 !2 \parallel c_2 !5][c_1, c_2, c_4, c_1, c_1, c_2] \not\models c_1 (2, 4) * c_4 \] \hspace{1cm} (4.5)
\[ \Gamma, [c_1 !(5 - 3, 3 + 1)][c_1] \parallel [c_4 !][c_4] \not\models c_1 (2, 4) * c_4 \] \hspace{1cm} (4.6)
\[ \Gamma, [c_1 !(2, 3)][c_1] \parallel [c_4 !][c_4] \not\models c_1 (2, 4) * c_4 \] \hspace{1cm} (4.7)

The first two systems fail to satisfy the assertion because they cannot evaluate to safely-stable systems due to lack of permission. In particular, in (4.4) process \(c_1 !2\) does not own permission \(\uparrow c_1\) required for communication (cf. cCom in Figure 2) whereas in (4.5) \(c_1 !2\) is missing permission \(\downarrow c_1\). The third system, (4.6), fails to satisfy the assertion although it is already a safely-stable system, as it violates the permission obligations for outputs imposed by \(\Gamma\) i.e., output \(c_4 !\) does not own permission \(\downarrow c_1\). Finally, the fourth system (4.7) fails to satisfy the assertion due to a mismatch between the data expected by the assertions and the data communicated by the outputs. We also have the following satisfactions involving the other assertion forms of the logic:

\[ (\Gamma, c_3 : \{\uparrow c_3\}), [\text{Fltr} \parallel \text{Dbl}][c_1, c_2, c_4, c_3] \parallel [c_1 !10][c_1] \parallel [c_2 !5][c_2] \models c_4 (10) * \text{blk}(c_3) \] \hspace{1cm} (4.8)
\[ \Gamma, [\text{Pr}][c_1, c_2, c_4] \parallel [c_1 !10][c_1] \parallel [c_2 !5][c_2] \models c_4 (10) * \text{any} \] \hspace{1cm} (4.9)
\[ \Gamma, [\text{Pr}][c_1, c_2, c_4] \parallel [c_1 !10][c_1] \parallel [c_2 !5][c_2] \models \text{any} \] \hspace{1cm} (4.10)
\[ \Gamma, (\text{new } c_3)(c_1 . c_3 !)[c_1] \parallel [c_2 ? c_3, \text{nil}][c_2] \models \text{blk}(c_1) * \text{blk}(c_2) \] \hspace{1cm} (4.11)

Satisfaction (4.8) requires us to extend \(\Gamma\) to account for the permission invariants of channel \(c_3\), which is not scoped. We also need the input permission \(\downarrow c_3\) as dictated by the satisfaction of the sub-assertion \(\text{blk}(c_3)\) in Figure 3. In the subsequent satisfaction, (4.9), \text{any} is used to describe the input-blocked process on a scoped channel \(c_3\) that is scoped in \(\text{Pr}\) (recall that \(\text{Pr} \triangleq (\text{new } c_3)(\text{Fltr} \parallel \text{Dbl})\)). Note also how, in (4.11), since \(c_3 \notin \text{dom}(\Gamma)\) (cf. satisfaction for \(q_1 \ast q_2\) in Figure 3), theScoped of the formula i.e., \(\text{blk}(c_1)\) and \(\text{blk}(c_2)\).
The satisfaction of the conjunct formula $\phi$ satisfaction. Hence any such satisfying system would violate the evaluation condition imposed on unstable junctions. Briefly in the Introduction, in our eventual framework of Section 5 systems will have the dual role of acting both as state as well as state-transformers. We therefore opted for the simpler interpretation that is conceptually easier to work with and chose a satisfaction interpretation that can be easily reasoned about in terms of the eventual stable systems reached.

**Example 4.4 (Unsatisfiability).** Formulas such as the ones below are unsatisfiable under the interpretation given in Figure 3.

$c(5) \triangleright c(6)$

$c(1) \triangleright blk(c)$

In the first case, i.e., $c(5) \triangleright c(6)$, sub-systems respectively satisfying $c(5)$ and $c(6)$ can never be merged into a well-resourced system as they must conflict on the permission $\uparrow c$ irrespective of the narrative chosen, due to the environment constraints set out in Definition 3.1. This is desirable because any system satisfying the first formula will create a race condition for any inputs on the channel $c$.

In later case, i.e., $c(1) \triangleright blk(c)$, sub-systems satisfying the sub-formulas of the separating conjunction become unstable once they are composed in parallel violating their respective sub-formula satisfaction. Hence any such satisfying system would violate the evaluation condition imposed on the satisfaction of the conjunct formula $\varphi_1 \triangleright \varphi_2$ in Figure 3. In fact, any sub-system $S_1$ satisfying $c(1)$ must evaluate to a stable system of the form $[c!e]_p$ where $e \parallel 1$. Similarly any sub-system $S_2$ satisfying $blk(c)$ must evaluate to a stable system that is structurally equivalent to $(\text{new } \vec{d})[c?x.P]_u$ (where $c \notin \vec{d}$). This means that, by the semantics of Section 3.1, $[c!e]_p$, $(\text{new } \vec{d})[c?x.P]_u$ is not stable, even if it is well-resourced (i.e., $\rho \cap \mu = \emptyset$). Our satisfaction definition $\varphi_1 \triangleright \varphi_2$ rules out this possibility by first requiring the composite system evaluates to a stable system before splitting. There are two reasons for this stricter interpretation. First, once the reduction happens leading to an evaluation to some other stable state $S_3$

$[c!e]_p, (\text{new } \vec{d})[c?x.P]_u \rightarrow (\text{new } \vec{d})[P^{1/\vec{x}}]_{\mu \cap p} \parallel S_3$

it may be the case that $S_3$ does not satisfy $c(1) \triangleright blk(c)$ anymore. Second, and perhaps more important, the above reduction can potentially trigger permission-violating or non-terminating behaviour in $(\text{new } \vec{d})[P^{1/\vec{x}}]_{\mu \cap p}$. For instance, process $P$ may be of the form $d!1 || d!2 || d?x.c!(x+y)$ i.e., it has two competing outputs on channel $d$. This implies that, whereas $(\text{new } \vec{d})[c?x.P]_u$ is safely-stable, its continuation is permission-violating, irrespective of the permissions held at that point, because it can hold at most one permission to output on channel $d$.

Since structural equivalence is central to Definition 3.8 ($\parallel$ in Definition 3.8 incorporates it), satisfaction abstracts over structurally equivalent systems, which allows us to work up-to structural equivalence when reasoning about systems. Moreover, we can also reason about formula satisfaction from existing system-formula satisfaction and systems that reduce (converge) to them in zero or more steps.

**Proposition 4.5 (Satisfaction and Evaluation).** $\Gamma, S \models \varphi$ implies $\exists T. S \parallel T$ and $\Gamma, T \models \varphi$

**Proposition 4.6 (Structural Eq. and Satisfaction).** $\Gamma, S \models \varphi$ and $S \equiv T$ implies $\Gamma, T \models \varphi$

**Proposition 4.7 (Satisfaction and Convergence).** $\Gamma, S \models \varphi$ and $T \rightarrow^* S$ implies $\Gamma, T \models \varphi$
We overload $\models$ to denote semantic implication amongst formulas in standard fashion. We then are able to prove certain properties about our logic, stated in Lemma 4.9.

**Definition 4.8 (Semantic Implication).** $\phi \models \psi \overset{\text{def}}{=} \Gamma, S \models \phi \Rightarrow \Gamma, S \models \psi$

**Lemma 4.9 (Formula equivalence).** The following bidirectional implications hold:

1. $\text{emp} * \phi \models \phi$
2. $\phi_1 * (\phi_2 * \phi_3) \models (\phi_1 * \phi_2) * \phi_3$
3. $\phi * \psi \models \phi * \psi$

### 4.3. Composing satisfactions.

Recall, from Example 4.4, that the satisfaction of the sub-assertions $\phi_1$ and $\phi_2$ does not necessarily imply the satisfaction of the composite assertion, $\phi_1 * \phi_2$. Nevertheless it is possible to determine when it is safe to infer this by analysing the structure of the sub-formulas. This analysis is formalised as the formula separation judgement, denoted as $\phi \perp \psi$ and defined in Definition 4.10. This judgement relies on the functions $\text{edg}(\cdot)$ and $\text{trg}(\cdot)$ to conservatively approximate matching outputs and inputs across sub-systems satisfying the formulas $\phi_1$, $\phi_2$ and, by prohibiting such matching channel operations, it ensures that no new reductions are introduced when sub-systems are composed in parallel. As a result, sub-systems that satisfy sub-formulas in a separating conjunction formulas must still satisfy the conjunction formula once composed, as stated in Lemma 4.11. This formula separation judgement is used later on by the proof system in Section 5 to circumvent the construction of problematic formulas such as those discussed in Example 4.4.

**Definition 4.10 (Formula Edges, Triggers and Separation).**

$$
\text{edg}(\phi) \overset{\text{def}}{=} \begin{cases} 
\emptyset & \text{if } \phi = \text{emp} \text{ or } \phi = \text{blk}(c) \\
\{c\} & \text{if } \phi = c\langle \vec{e} \rangle \\
\text{edg}(\phi_1) \cup \text{edg}(\phi_2) & \text{if } \phi = \phi_1 * \phi_2 \\
\text{undefined} & \text{otherwise}
\end{cases}
$$

$$
\text{trg}(\phi) \overset{\text{def}}{=} \begin{cases} 
\emptyset & \text{if } \phi = \text{emp} \text{ or } \phi = c\langle \vec{e} \rangle \\
\{c\} & \text{if } \phi = \text{blk}(c) \\
\text{trg}(\phi_1) \cup \text{trg}(\phi_2) & \text{if } \phi = \phi_1 * \phi_2 \\
\text{undefined} & \text{otherwise}
\end{cases}
$$

$$
\phi \perp \psi \overset{\text{def}}{=} \text{edg}(\phi) \cap \text{trg}(\psi) = \emptyset \land \text{edg}(\psi) \cap \text{trg}(\phi) = \emptyset
$$

**Lemma 4.11 (Merging Assertions).**

$\Gamma, S \models \phi$ and $\Gamma, T \models \psi$ and $S \perp T$ and $\phi \perp \psi$ implies $\Gamma, S \parallel T \models \phi * \psi$

**Proof.** See Appendix A.3.

Note that, for a number of conjunctions, the sub-formulas are trivially separate making formula separation checks superfluous. For instance, $\text{emp}$ is separate from any formula; also state formulas $\chi_1 * \chi_2$ are trivially separate, $\chi_1 \perp \chi_2$ as stated in Proposition 4.12.

**Proposition 4.12.** For any environment, $\Gamma$, state formulas, $\chi, \eta$ and formula $\psi$ we have:

1. $\chi \perp \eta$
2. $\eta \perp \text{emp}$

**Proof.** Immediate from 4.10
5. Proof System

We complete our framework by developing a compositional proof-system for the logic of §4 interpreted according to the satisfaction of Figure 3. Our sequents, inspired by Hoare triples, have the format

\[ \Gamma; b \vdash \{\varphi\} S \{\psi\}, \]

where \( S \) is a well-resourced system, \( \varphi \) and \( \psi \) are respectively the pre-condition and post-condition, \( \Gamma \) is a permission environment, and \( b \) is a boolean expression defined in Figure 1, now serving as a boolean formula over our value domain. The system, formulas and boolean condition in a sequent are potentially open i.e., that may have free variables. Thus, the meaning of our sequents quantifies over all substitutions, \( \sigma \in \text{Sub} \) that make the boolean condition evaluate to true, and also over all systems \( T \in \text{Sys} \) which are separate from \( S \) and which satisfy the precondition in the following way.

**Definition 5.1 (Sequent satisfaction).**

\[ \Gamma, b \vdash \{\varphi\} S \{\psi\} \quad \text{def} = \quad \forall \sigma, T. t. b \sigma \Downarrow \text{tt}, \quad \Gamma, T \sigma \vdash \varphi \sigma, \quad T \sigma \perp S \sigma \quad \text{implies} \quad \Gamma, (T || S) \sigma \vdash \psi \sigma \]

As in [19], our sequents tease apart auxiliary reasoning about our value domain, since determining the truth (or otherwise) of these boolean formulas is process-independent. Such disentangling also allows us to make refined claims about derivations in our system. For instance, if we limit value expressions to Presburger arithmetic, we know that our boolean formula derivations exists and are decidable [30].

We note that our sequents deal with total-correctness. Formula satisfaction, defined in Figure 3, centers around system evaluation, \( S \Downarrow T \), which existentially quantifies over one sequence of system reductions. The strength of what may, at first, seem a rather weak behaviour assertion comes from the determinism properties afforded by our model of confined processes. In fact, Theorem 3.11 (Evaluation Determinism) allows us to extend such behaviour assertions to universal system behaviour, up-to redundant permissions. What we are ultimately interested in however is universal processes behaviour. This can then be retrieved in immediate fashion through Definition 5.6 (Process Satisfaction), defined later in Section 5.3, Theorem 3.24 (Process Convergence), and ultimately, Theorem 3.23 (Process Evaluation Determinism).

The proof system, defined by the rules in Figure 4, assumes the derivation judgement \( b_1 \vdash b_2 \) between two (possibly open) boolean formulas, with the expected property that

\[ \forall \sigma : \text{Sub}. b_1 \Downarrow b_2 \quad \text{and} \quad b_1 \sigma \Downarrow \text{tt} \quad \text{implies} \quad b_2 \sigma \Downarrow \text{tt} \]

Most of the logical rules are rather intuitive and their ‘naturality’ is, in part, due to the strong substratum provided by process confinement, in terms of absence of races. We have four logical axioms where \( \text{lNl} \), \( \text{lBlk} \) and \( \text{lOut} \) deal with stable systems. More precisely, \( \text{lNl} \) acts as a wire between the precondition and the postcondition, \( \text{lBlk} \) trivialises proofs with an unsatisfiable boolean condition, \( \text{lOut} \) generates input-blocked process assertions, and \( \text{lOut} \) generates data assertions.

The rule \( \text{lIn} \) is central to the proof system as it is the only rule that consumes part of the precondition. Together with \( \text{lOut} \) and \( \text{lPar} \) they capture process communication in our proof system. In particular, they observe the permission mutual-exclusion invariants dictated by the environment, whereby the side-condition in \( \text{lOut} \), i.e., \( \Gamma(c) \subseteq \rho \), forces outputs to own the permissions guarded by the mutual exclusion through the side-condition \( \Gamma(c) \in \rho \), whereas the premise in \( \text{lIn} \) permit inputs to assume ownership of these guarded permissions after communication, through the masking of these permissions in the conclusion, i.e., \( \rho \setminus \Gamma(c) \). The permission checking side-conditions in
the axioms LOut and LBLK ensure that stable systems are safe; similarly, the permission checking side-condition in LIN ensures that evaluations are also safe - recall that any permission violation is propagated down to the eventual stable system by Lemma [3,7].

The system parallel composition rule (LPar) is central to our proof system. It is the only rule that allows us to introduce a cut-middle formula in the hypotheses, q3. The asymmetry in the hypotheses of this rule guarantees the existence of a reduction sequence across two independently verified sub-systems since the unidirectional cut disallows mutual dependencies across the premise sequents; this prevents deadlocks and ensures total correctness. LPar also carries two side-conditions, ψ1 ⊥ ψ2 and q2 ⊥ q3, denoting formula separation, defined in Definition [4,10].

The proof system also has a rule for process parallel composition, (LSpl), which forces a partitioning of permission-resources, analogously to cSpl from Figure 2, similarly, the process scoping
rule \((\llcl)\) follows rule \(cLcl\) from Figure 2. The system scoping rule \((\lres)\) restricts the permission-guarding invariants relating to the scoped channels and filters assertions blocked by the scoping using the function \(\psi \setminus \vec{c}\), as defined in Definition 5.2; in particular this function over-approximates to any any message state assertions and input-blocked assertions affected by the name scoping of the restriction. \(\lres\) also uses an environment restriction operation \(\Gamma \setminus c\) defined in Definition 5.3.

**Definition 5.2** (Formula Restriction).

\[
\psi \setminus \vec{c} \triangleq \begin{cases} 
  d(\vec{c}) & \text{if } \psi = d(\vec{c}) \text{ and } d \notin \vec{c} \\
  \text{blk}(d) & \text{if } \psi = \text{blk}(d) \text{ and } d \notin \vec{c} \\
  \text{emp} & \text{if } \psi = \text{emp} \\
  (\psi_1 \setminus \vec{c}) \ast (\psi_2 \setminus \vec{c}) & \text{if } \psi = \psi_1 \ast \psi_2 \\
  \text{any} & \text{otherwise}
\end{cases}
\]

**Definition 5.3** (Environment Restriction).

\[
\Gamma \setminus c \triangleq \begin{cases} 
  \emptyset & \text{if } \Gamma = \emptyset \\
  \Gamma' \setminus c & \text{if } \Gamma = \Gamma', c : \rho \\
  (\Gamma' \setminus c), d : (\rho \setminus \downarrow c, \uparrow c) & \text{if } \Gamma = \Gamma', d : \rho \text{ and } c \neq d
\end{cases}
\]

**Proposition 5.4.** If \(\Gamma\) is a permission environment then \(\Gamma \setminus c\) is as well.

**Proof.** It is immediate to check that Definition 4.1 is still observed by \(\Gamma \setminus c\), in particular that it is suitably closed (Definition 4.1.2).

The remaining logical rules are fairly straightforward. In the conditional proof rule \(\llif\), the hypotheses on each branch are augmented with the corresponding assertion, as usual in Hoare logics; this mechanism works in pairs with the structural rule \(\llls\) which trivialises the proof obligations on unreachable branches. \(\lldef\) completes the treatment of the logical rules in the obvious way. Note that rules \(\llin\) and \(\lldef\) abuse the substitution notation, extending it from values to (possibly open) expressions.

The proof system also has a number of structural rules. The rule \((\llinst)\) permits instantiations of generic sequents whereas \((\llsub)\) permits substitutions of expressions to variables that can be inferred to be equivalent from the sequent boolean expression. The rule \(\llren\) renames channel names in sequents; the rule side-condition guarantees that the name \(d\) is fresh which make renaming injective. Finally, \((\llimp)\) endows proofs with a basic understanding of structural equivalence, \(\equiv\), and of logical implication, \(\vdash\).

**5.1. Derived Rules.** Although \(\llpar\) is used extensively when proving properties of parallel communicating processes, it turns out that we often do not require its full power which makes it somewhat cumbersome to use. We therefore derive lightweight versions of \(\llpar\), enabling parallel code to be either logically sequenced thereby focussing on cutting intermediary formulas (\(\llcut\)), or else considered totally separate, where composite pre-conditions are assumed to produce composite post-conditions (\(\llsep\)). These derived rules require fewer side-conditions relating to formula separation. For instance, \(\llcut\) disposes of the side-conditions entirely, and \(\llsep\) limits them to one check.

\[
\begin{align*}
\llcut & : \Gamma, b \vdash \{\psi_1\} S \{\psi\} & \llsep & : \Gamma, b \vdash \{\psi_1\} S \{\psi_1\} \\
\Gamma, b \vdash \{\psi_1\} S \parallel T \{\psi_2\} & \quad \Gamma, b \vdash \{\psi_2\} T \{\psi_2\} & \quad \psi_1 \equiv \psi_2 \\
\Gamma, b \vdash \{\psi_1 \ast \psi_2\} S \parallel T \{\psi_1 \ast \psi_2\} & \quad \Gamma, b \vdash \{\psi_1 \ast \psi_2\} T \{\psi_1 \ast \psi_2\}
\end{align*}
\]
For state formula pre and postconditions, an even simpler version lSep is obtained by Corollary 4.12 i.e., lSepSt, which requires no side-conditions at all.

\[ \frac{\Gamma; b \vdash \{ \eta_1 \} S \{ \chi_1 \} \quad \Gamma; b \vdash \{ \eta_2 \} T \{ \chi_2 \}}{\Gamma; b \vdash \{ \eta_1 \ast \eta_2 \} S \parallel T \{ \chi_1 \ast \chi_2 \}} \]

The derivations of these lightweight parallel rules are straightforward and use formula semantic implications from Lemma 4.9 together with properties for formula separation from Proposition 4.12; See Appendix A.4.

The output axiom rule lOut appears frequently in most derivations using our proof system. We find it convenient to formulate another derived rule that facilitates comparisons between the expression outputted by the process and that specified by the state formula, even when these expressions do not syntactically match.

\[ \frac{b \models e_1 = e_2 \quad \Gamma(c) \subseteq \rho}{\Gamma; b \vdash \{ \text{emp} \} [c!e_1]_{\rho} [c(e_2)]} \]

Dually, the rule lIn is used frequently to dispose of cut-formulas. However the direct use of this rule can become unwieldy due to necessary system structural manipulations required to get the system in form required by the rule. A more convenient version can be derived that abstracts away from structural equivalence manipulations.

\[ \frac{T \equiv [c ? P]_{[c(\rho)]} \parallel S \quad \bot c \in \rho \quad \Gamma; b \vdash \{ \psi \} [P[b]_{[c]}]_{\rho} \parallel S \{ \psi \}}{\Gamma; b \vdash \{ \psi \ast c(e_2) \} T \{ \psi \}} \]

The proofs for these derived rules are straightforward and relegated to Appendix A.4.

Derived rules similar to lIn can be obtained for lDef, lFin, lSep and lLcl using an analogous derivation. In Section 6 we shall often abuse this fact and use the derived rule named as the respective proof rule while at the same abstracting away from structural manipulations.

5.2. **Frame Rule.** The frame rule embodies local reasoning in separation-based logics [33]. For satisfiable post-conditions, a variant of the frame rule can be derived in our proof system.

\[ \frac{\Gamma; b \vdash \{ \psi_1 \} S \{ \psi_2 \} \quad \psi_2 \perp \psi}{\Gamma; b \vdash \{ \psi_1 \ast \psi \} S \{ \psi_2 \ast \psi \}} \]

Moreover, for the special case when the pre and post conditions are state formulas, the frame rule eliminates the need for the side condition as stated below.

\[ \frac{\Gamma; b \vdash \{ \chi_1 \} S \{ \chi_2 \}}{\Gamma; b \vdash \{ \chi_1 \ast \eta \} S \{ \chi_2 \ast \eta \}} \]

We here show the derivation for the more general version of frame rule, i.e., lFrm, using the proof rules (lNil), (lPar) and (lImpl) and the structural rule \( S \parallel [\text{nil}]_0 \equiv S \).

\[ \frac{\Gamma; b \vdash \{ \psi \} \parallel S \{ \psi \}}{\Gamma; b \vdash \{ \psi \ast \psi \} S \{ \psi \}} \]

\[ \frac{\Gamma; b \vdash \{ \psi \} \parallel S \{ \psi \}}{\Gamma; b \vdash \{ \psi \ast \psi \} S \{ \psi \}} \]

\[ \frac{\Gamma; b \vdash \{ \psi \} \parallel S \{ \psi \}}{\Gamma; b \vdash \{ \psi \ast \psi \} S \{ \psi \}} \]

Our proof-system is sound with respect to Definition 5.1

**Theorem 5.5 (Soundness).** \( \Gamma; b \vdash \{ \psi \} S \{ \psi \} \) implies \( \Gamma, b \vdash \{ \psi \} S \{ \psi \} \).
**Proof.** By rule induction on \(\Gamma; b \vdash \{q\} S \{\psi\}\). We here show the main rules:

**Out:** For arbitrary \(\sigma\), \(T\) we have:

\[
\begin{align*}
(b \sigma) \parallel \text{tt} & \quad (5.1) \\
\Gamma, T \sigma & \models \text{emp} \sigma \quad (5.2) \\
T \sigma & \perp [c!\vec{e}]_\rho \sigma \quad (5.3)
\end{align*}
\]

and the side-condition

\[
\Gamma(c) \subseteq \rho \quad (5.4)
\]

By Figure 3 and (5.2) we know

\[
T \sigma \parallel [\text{nil}]_\emptyset \quad (5.5)
\]

By (5.3) we know that \(T \sigma \parallel [c!\vec{e}]_\rho \sigma\) is well-resourced. Moreover, by (5.5) and cPar and scNt of Figure 2 we deduce

\[
T \sigma \parallel [c!\vec{e}]_\rho \sigma \rightarrow^* [\text{nil}]_\emptyset \parallel [c!\vec{e}]_\rho \sigma \equiv [c!\vec{e}]_\rho \sigma \quad (5.6)
\]

Clearly, \([c!\vec{e}]_\rho \sigma \not\rightarrow^*\). Moreover by the conditions imposed on environment mappings in Definition 4.1 we know \(\uparrow c \in \Gamma(c)\) and thus by (5.4) we deduce that \(\uparrow c \in \rho\) and hence that \([c!\vec{e}]_\rho \sigma \not\rightarrow\text{cat}\). As a result, from (5.6) we obtain \(T \sigma \parallel [c!\vec{e}]_\rho \sigma \perp [c!\vec{e}]_\rho \sigma\) and for some \(\vec{v}\) where \(\bar{e}\sigma \parallel \vec{v}\) and by (5.4) and Figure 3 we obtain \(\Gamma, (T \parallel [c!\vec{e}]_\rho \sigma) \models (c\langle\vec{e}\rangle \sigma)\).

**In:** For arbitrary \(\sigma\), \(T\) we have:

\[
\begin{align*}
(b \sigma) \parallel \text{tt} & \quad (5.7) \\
\Gamma, T \sigma & \models (q \ast c(\vec{e})) \sigma \quad (5.8) \\
T \sigma & \perp ([c?\vec{x}.P]_{\rho(\Gamma(c))} \parallel S) \sigma \quad (5.9)
\end{align*}
\]

and the side-condition

\[
\downarrow c \in \rho \quad (5.10)
\]

By (5.8) and Figure 3 we know

\[
T \parallel (\text{new } \vec{d})(T_1 \parallel T_2) \quad (5.11)
\]

where \(\vec{d} \notin \text{dom}(\Gamma)\)

\[
\Gamma, T_1 \models q \sigma \quad (5.12)
\]

and \(\Gamma, T_2 \models c\langle\vec{e}\rangle \sigma\)

By \(\Gamma, T_2 \models c\langle\vec{e}\rangle \sigma\) and Figure 3 we know

\[
T_2 \parallel [c!\vec{e}]_\mu \text{ where } \bar{e}\sigma \parallel \vec{v}, \vec{e}_\mu \parallel \vec{v} \text{ and } \Gamma(c) \subseteq \mu \quad (5.15)
\]

By (5.9) we know \(T \sigma \perp ([c?\vec{x}.P]_{\rho(\Gamma(c))} \parallel S) \sigma\) is well-resourced and by (5.12) and \(\Gamma(c) \subseteq \mu\) of (5.15) we know that \(c \notin \vec{d}\) and that \(\vec{d} \notin \text{nm}(\mu)\). Thus by (5.11), (5.15) and cPar, cCom, (5.12) and scExt we obtain

\[
\begin{align*}
T \sigma \parallel ([c?\vec{x}.P]_{\rho(\Gamma(c))} \parallel S) \sigma \rightarrow^* & \equiv (\text{new } \vec{d})(T_1) \parallel [c!\vec{e}]_\mu \parallel ([c?\vec{x}.P]_{\rho(\Gamma(c))} \parallel S) \sigma \quad (5.16) \\
(\text{new } \vec{d})(T_1) \parallel [c!\vec{e}]_\mu \parallel ([c?\vec{x}.P]_{\rho(\Gamma(c))} \parallel S) \sigma \rightarrow & (\text{new } \vec{d})T_1 \parallel ([c?\vec{x}.P]_{\rho(\Gamma(c))} \parallel S) \sigma \quad (5.17)
\end{align*}
\]
By (5.16) and Lemma 3.6 we know that \((\text{new } \vec{d})(T_1) \parallel (\lceil c?\vec{x}.P \rceil_{\rho,\Gamma(c)} \parallel S)\sigma\) is well-resourced, and by \(\Gamma(c) \subseteq \mu\) of (5.15) we deduce that

\[(\text{new } \vec{d})T_1 \perp (\lceil \psi \rceil_{\rho,\Gamma}) S)\sigma \quad (5.18)\]

By (5.13), (5.12) and Lemma A.17 we obtain

\(\Gamma, (\text{new } \vec{d})T_1 \vdash \psi\sigma\)

and thus by (5.7), (5.18), the premise \(\Gamma; b \vdash \psi\lceil P\{\vec{e}/\vec{x}\}\rceil_{\rho} \parallel S\) (\(\psi\)) and I.H. we obtain

\(\Gamma, (\text{new } \vec{d})T_1 \parallel (\lceil P\{\vec{e}/\vec{x}\}\rceil_{\rho} \parallel S)\sigma \quad (5.19)\)

By (5.13), (5.12) and Lemma A.17 we obtain

\(\Gamma, (\text{new } \vec{d})T_1 \perp (\lceil \psi \rceil_{\rho,\Gamma}) S)\sigma \quad (5.20)\)

and side-conditions

\(\psi_2 \perp \psi_3 \quad (5.23)\)

\(\psi_1 \perp \psi_2 \quad (5.24)\)

By (5.21) we know

\(R\sigma \parallel (\text{new } \vec{d})(R_1 \parallel R_2) \quad (5.25)\)

where \(\vec{d} \notin \text{dom}(\Gamma)\)

\(\Gamma, R_1 \vdash \psi_1\sigma \quad (5.27)\)

\(\Gamma, R_2 \vdash \psi_2\sigma \quad (5.28)\)

By (5.25), (5.22) and Lemma 3.6 we know

\(R_1 \perp R_2 \quad (5.29)\)

and \(R_1 \perp S\sigma \parallel T\sigma \quad (5.30)\)

\(\text{and } R_2 \perp S\sigma \parallel T\sigma \quad (5.31)\)

By (5.20), (5.27), \(R_1 \perp S\sigma\) from (5.30) and I.H. we have \(\Gamma, R_1 \parallel S\sigma \vdash (\psi_1 \ast \psi_3)\sigma\) and from the satisfaction definition of Figure 3, we obtain

\(R_1 \parallel S\sigma \parallel (\text{new } \vec{d})(S_1 \parallel S_2) \quad (5.32)\)

where \(\vec{d} \notin \text{dom}(\Gamma)\)

\(\Gamma, S_1 \vdash \psi_1\sigma \quad (5.34)\)

\(\text{and } \Gamma, S_2 \vdash \psi_3\sigma \quad (5.35)\)

\(\text{LPAR: For arbitrary } \sigma, R \text{ we have:}\)

\(b\sigma \parallel tt \quad (5.20)\)

\(\Gamma, R\sigma \vdash (\psi_1 \ast \psi_2)\sigma \quad (5.21)\)

\(R\sigma \perp S\sigma \parallel T\sigma \quad (5.22)\)
By (5.29) and (5.31) we know \( R_1 \parallel S \sigma \perp R_2 \). Thus, by (5.32) and Lemma 3.5 we derive \( S_1 \perp R_2 \), and by (5.28), (5.35), the rule side-condition (5.23) and Lemma 4.11 we obtain
\[
\Gamma, R_2 \parallel S_2 \parallel T \sigma = (\phi_2 \ast \phi_3) \sigma
\] (5.36)

Using (5.29) and (5.32) we can also derive \( R_2 \parallel S_2 \perp T \sigma \) and by (5.20), (5.36) and I.H. we derive
\[
\Gamma, R_2 \parallel S_2 \parallel T \sigma = \psi_2 \sigma
\] (5.37)

By (5.29) and (5.32) we also derive \( S_1 \perp (R_2 \parallel S_2 \parallel T) \) and by the rule side-condition (5.24) and Lemma 4.11 we obtain
\[
\Gamma, S_1 \parallel R_2 \parallel S_2 \parallel T \sigma = (\psi_1 \ast \psi_2) \sigma
\] (5.38)

Thus by (5.26), (5.33) and Lemma A.17 we deduce
\[
\Gamma, (\text{new } \vec{c}, \vec{d}) (S_1 \parallel R_2 \parallel S_2 \parallel T) \sigma = (\psi_1 \ast \psi_2) \sigma
\] (5.39)

5.3. Process Sequent Satisfaction. We conclude this section with Definition 5.6, which extends sequent satisfaction to processes by assuming the existence of a permission environment and the respective permission-set, required by the satisfaction definition of Figure 3. This allows for the possibility of having multiple narratives explaining determinism, and is in line with the “ownership is in the eye of the asserter” principle [28].

**Definition 5.6 (Process Sequent Satisfaction).**

\[
b \models \{ \psi \} P \{ \psi \} \equiv \text{exists } \Gamma, \rho \text{ such that } \Gamma, b \models \{ \psi \} [P]_\rho \{ \psi \}
\]

**Example 5.7.** According to 5.6, we can now state that \( P_{rg} \), from Example 2.7 satisfies the property
\[
x \leq 9 \models \{ c_1(x) \ast c_2(y) \} P_{rg} \{ c_1(x, 2x) \ast c_4() \},
\] (5.39)

while abstracting over the narrative as to why \( P_{rg} \) is deterministic. It can be read as saying that, given two values \( x \) and \( y \) on channels \( c_1 \) and \( c_2 \) respectively, \( P_{rg} \) returns the value of \( x \) together with its double on \( c_1 \) and a signal on \( c_4 \), provided that the value of \( x \) is less than 10. Mirroring the previous discussion in Example 2.7, \( P_{rg} \) also satisfies the property
\[
x > 9 \models \{ c_1(x) \ast c_2(y) \} P_{rg} \{ c_4(x) \ast \text{any} \},
\] (5.40)

where \( \text{any} \) abstracts over the blocked code \( (\text{new } c_3) (c_3 ? x_4 \cdot c_1 ! (x_4 + x_4)), \) as described earlier in Example 4.3.

We are also in a position to specify the correctness of our quicksort algorithm through some macro definitions for compactness.
Another possible narrative is the permission-set yielding the system specification together with the permission-transfer invariants. Moreover, the values returned are in Example 5.7 by first augmenting the satisfaction specification with an appropriate narrative for the permissions that they own.

Example 5.8 (Specifying Correctness for Parallel Quicksort). The expected behaviour of Qck(i, j) from Example 5.8 can be expressed through the sequent satisfaction

$$\left( \text{ord}(\vec{y}_i^j) \land \vec{x}_i^j = \vec{y}_i^j \right) \Rightarrow \{ A_i^j(\vec{x}_i^j) \} \text{ Qck}(i, j) \{ A_i^j(\vec{y}_i^j) \} \ast r(i) \tag{5.41} $$

using the following macro definitions, whereby $\vec{x}_i^j$ denotes lists of variables $x_i \ldots x_j$ when $i \leq j$ and the empty-list $\epsilon$ otherwise

$$A_i^j(\vec{x}_i^j) \overset{\text{def}}{=} \begin{cases} \text{emp} & \text{if } i > j \\ a_i(x_i) \ast A_i^{j+1}(\vec{x}_{i+1}^j) & \text{if } i \leq j \end{cases}$$

$$\text{ord}(\vec{x}_i^j) \overset{\text{def}}{=} \begin{cases} \text{true} & \text{if } i = j \\ x_i \leq x_{i+1} \land \text{ord}(\vec{x}_{i+1}^j) & \text{if } i < j \end{cases}$$

$$\vec{x}_i^j = \vec{y}_i^j \overset{\text{def}}{=} \begin{cases} \text{true} & \text{if } i = j \\ \bigvee_{i \leq k \leq j} (\vec{y}_i^j = \vec{y}_i^{k-1} \cdot x_i \vec{y}_i^{j-1}) \land (\vec{x}_{i+1}^j = \vec{y}_i^{k-1} \cdot x_{i+1} \vec{y}_i^{j-1}) & \text{if } i < j \end{cases}$$

The specification of (5.41) above states that when Qck(i, j) is composed with an array of arbitrary values on channels $a_1 \ldots a_j$, denoted by the assertion macro $A_i^j(\vec{x}_i^j)$, it returns another array of values on the same channel list, $A_i^j(\vec{y}_i^j)$, together with a signal on channel $r$ denoting completion. Moreover, the values returned are

1. ordered, expressed as the predicate $\text{ord}(\vec{y}_i^j)$
2. equal, up to reordering, to the original values, expressed as the predicate $\vec{x}_i^j = \vec{y}_i^j$.

6. Application

We conclude by revisiting the properties stated in Section 5.3 and show how our proof-system can be used to prove properties about them. In Example 6.1, we see how proofs about concurrent code are performed by running through only one possible reduction trace, even when other interleavings are possible. The main appeal of these proofs is however their amenability to compositionality as shown in Example 6.2. In this example proof, the behaviour of sub-programmes is verified in terms of their pre and post conditions only, without any concern towards external interference from other concurrent code. Independently verified sub-programmes are then merged together using lPar (and its variants lCut, lSep and lSepSt), as long as the sub-programmes are separate wrt. the permissions that they own.

Example 6.1 (Proving Satisfiability). We prove the specifications (5.39) and (5.40) stated earlier in Example 5.7 by first augmenting the satisfaction specification with an appropriate narrative for determinism as stated in Definition 5.6. One possible narrative is the permission-set $\{ \downarrow c_1, \downarrow c_2, \uparrow c_4 \}$ together with the permission-transfer invariants

$$\Gamma = c_1: \{ \uparrow c_1 \}, c_2: \{ \uparrow c_2 \}, c_4: \{ \uparrow c_4, \downarrow c_1 \}$$

yielding the system specification

$$\Gamma, x \leq 9 \models \{ c_1(\langle x \rangle) \ast c_2(\langle y \rangle) \} \left[ \text{Pr}_2 \right]_{\downarrow c_1, \downarrow c_2, \uparrow c_4} \{ c_1(\langle x, 2x \rangle) \ast c_4() \} \tag{6.1}$$

Another possible narrative is the permission-set $\{ \downarrow c_1, \downarrow c_2 \}$ and the environment

$$\Gamma' = c_1: \{ \uparrow c_1, \uparrow c_4 \}, c_2: \{ \uparrow c_2 \}, c_4: \{ \uparrow c_4, \downarrow c_1 \}$$
yielding a different intensional specification explaining the process determinism below:

\[ \Gamma', \ x \leq 9 \models \{c_1(x) \ast c_2(y)\} \ [\text{Prf}]_{\{c_1, \ x\leq 2, \ t_c\}} \ \{c_1(x, 2x) \ast c_4()\} \]

We here focus on the specification with the first narrative, (6.1), which by Theorem 5.5 follows from the proof of the sequent

\[ \Gamma; \ x \leq 9 \vdash \{c_1(x) \ast c_2(y)\} \ [\text{Prf}]_{\{c_1, \ x\leq 2, \ t_c\}} \ \{c_1(x, 2x) \ast c_4()\} \]

Since \( \text{Prf} \triangleq (\text{new } c_3) (\text{Fltr} \parallel \text{Db}l) \), we prove (6.2) by applying the proof rules LDef followed by lLcl and lRes, which leaves us with the following sequent to prove

\[ \Gamma''; \ x \leq 9 \vdash \{c_1(x) \ast c_2(y)\} \ [\text{Fltr} \parallel \text{Db}l]_{\{c_1, \ x\leq 2, \ t_c, \ t_{c3}, \ t_t\}} \ \{c_1(x, 2x) \ast c_4()\} \]

where \( \Gamma'' \) is the extended environment \( \Gamma'' = \Gamma, t_c : [t_{c3}, \ u] \). Note that, through lRes, in (6.3) we have also increased the permissions owned by the system with \( t_{c3} \) and \( u^c \), the permissions relevant to the scope of \( c_3 \), opened by lRes. Moreover for lRes, the post-condition is unaffected in this case, i.e., according to Definition 5.2, \( (c_1(x, 2x) \ast c_4()) \setminus c_3 = c_1(x, 2x) \ast c_4() \). After applying the logical rule Lspl, followed by two applications of LDef for Fltr and DbI we are left with

\[ \Gamma''; \ x \leq 9 \vdash \{c_1(x) \ast c_2(y)\} \ \begin{cases} 
\text{if } x_1 \leq 9 \text{ then } \text{c1}! x_1 \ | \ \text{c1}! t_{x_3}, \ (\text{c1}! (x_1, x_3)) || c_{4}() \\
\text{else } \text{c2}! x_1 \ | \ \ \text{[c2 xd2 c3 xd4 c4](x_4 + x_4)]}_{\{c_1, t_{c3}, t_{c4}\}} \end{cases} \ \{c_1(x, 2x) \ast c_4()\} \]

We proceed by applying LIn twice for \( c_1 \) and \( c_2 \) (in any order) and then by applying LFl, which gives us one unreachable branch since \( x \leq 9 \land \neg(x \leq 9) \Rightarrow \text{false} \); this can be discharged by LImp and the axiom LFls. The reachable premise can be proved as follows; we elide the environment and boolean condition from the sequents below as they remain unchanged throughout:

\[ \frac{\Gamma''(c_1) \subseteq \{t_{c1}\} \ \text{emp}}{\{t_{c1}, t_{\text{c}2}\} \ \{c_1(x, 2x)\} \ \text{LOUT}} \]

\[ \frac{\Gamma''(c_2) \subseteq \{t_{c1}, t_{c4}\} \ \text{emp}}{\{c_1! (x, x), c_1!(x, x), c_{4}()\} \ \text{LOUT}} \]

\[ \frac{\text{emp}}{\{t_{c1}, t_{c4}\} \ \{c_1! (x, x), x_2, c_{4}()\} \ \text{LOUTD}} \]

\[ \frac{x + x = 2x \ \text{emp}}{\{c_1!(x + x), c_{4}()\} \ \text{LOUT}} \]

\[ \frac{\text{emp}}{\{t_{c1}, t_{c3}\} \ \{c_1!(x), c_{4}()\} \ \text{LOUT}} \]

\[ \frac{\text{emp}}{\{c_3(x), c_{4}()\} \ \text{LIn}} \]

\[ \frac{\text{emp}}{\{c_3(x), c_{4}()\} \ \text{LOut}} \]

Similarly, the proof for the second specification (5.40) in Example 5.7 can also be proved by the sequent:

\[ \Gamma; \ x > 9 \vdash \{c_1(x) \ast c_2(y)\} \ [\text{Prf}]_{\{c_1, c_2, t_{c}\}} \ \{c_4(x) \ast \text{any}\}, \quad (6.4) \]
The proof is similar to that of (6.2), where we first apply `lDef`, `lLcl`, and `lRes`, which leaves us with the following sequent:

\[ \Gamma'' : x > 9 \vdash \{ c_1(x) \land c_2(y) \} \quad \{ Fltr \mid Dbl \}_{|t_{c1}, t_{c2}, t_{c3}, t_{c4}|} \{ c_4(x) \land b_{lk}(c_3) \} \] (6.5)

where, this time, we have the premise postcondition obtained as \( c_4(x) \land b_{lk}(c_3) \) \& \( c_3 = c_4(x) \land \text{any} \) according to Definition 5.2. Again, similar to the proof for (6.2), we apply `lSep` to (6.5) followed by two applications of `lDef` for `Fltr` and `Dbl`. Then we apply `lIn` twice for \( c_1 \) and \( c_2 \) to consume the state formula in the precondition, and then by applying `lL`. This time, the rule for conditional gives us a different unreachable branch since \( x > 9 \land x \leq 9 \Rightarrow \text{false} \). The reachable premise can be proved as follows:

\[
\begin{array}{c}
\Gamma''(c_4) \subseteq \{ c_1, c_1, c_1, c_1 \} \\
\text{\textbf{emp}} \quad \{ c_2 \! \! x \} |_{t_{c1}, t_{c2}, t_{c3}, t_{c4}} \quad \{ c_4(x) \} \quad \text{\textbf{LOut}} \\
\quad \downarrow t \in \{ t_{c2}, t_{c3}, t_{c2} \} \\
\quad \text{\textbf{emp}} \quad \{ c_2 \! \! x \} |_{t_{c1}, t_{c2}, t_{c3}, t_{c4}} \quad \{ b_{lk}(c_3) \} \quad \text{\textbf{LBk}} \\
\quad \{ c_2 \! \! x \} |_{t_{c1}, t_{c2}, t_{c3}, t_{c4}} \quad \{ b_{lk}(c_3) \} \quad \text{\textbf{LSep}} \\
\end{array}
\]

\[ \frac{\text{Example 6.2 (Proving Correctness for Parallel Quicksort). To prove the correctness property (5.41) for Qck(i, j), as stated in Example 5.8 we choose a narrative where the environment is}}}{\text{\textbf{emp}}} \quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\quad \{ r(i, j) \} \quad \text{\textbf{LSub}} \\
\end{array}
\]

\[ \frac{\text{\textbf{emp}}} \quad \{ r(i, j) \} \quad \text{\textbf{LDef}} \\
\frac{\text{\textbf{emp}}} \quad \{ r(i, j) \} \quad \text{\textbf{LDef}} \\
\frac{\text{\textbf{emp}}} \quad \{ r(i, j) \} \quad \text{\textbf{LDef}} \\
\frac{\text{\textbf{emp}}} \quad \{ r(i, j) \} \quad \text{\textbf{LDef}} \\
\frac{\text{\textbf{emp}}} \quad \{ r(i, j) \} \quad \text{\textbf{LDef}} \\
\]
The inductive case, \( n+1 = j-i \), i.e., adding \( i < j \) to the sequent boolean expression, assumes that the property holds for all \( m \leq n \), i.e., all \( m < j-i \) (the inductive hypothesis), and follows from proving the following two sequents

\[
\Gamma_1; b \vdash \left\{ A_i^j(\langle i \rangle^j) \right\} \quad [\text{Pr}t(i, j)]_{\rho(r_1, r_j)} \quad \left\{ A_i^p(\langle i \rangle^p) \ast \rho_p(y_p) \ast A_{p+1}^j(\langle i \rangle^j) * r_3(p) \right\}
\]

\[
\Gamma_1; b \vdash \left\{ A_i^p(\langle i \rangle^p) \ast \rho_p(y_p) \ast A_{p+1}^j(\langle i \rangle^j) * r_3(p) \right\}
\]

where \( \Gamma_1 \) extends \( \Gamma \) with the mapping for \( r_3 \) i.e., \( \Gamma_1 = \Gamma, r_3: \rho(r_3, i, j) \) and \( b \) is a stronger boolean condition defined as:

\[
b = \text{ord}(\langle y \rangle^i) \wedge \bar{y}_i^p \leq \bar{y}_i^p \wedge (\lambda k \geq k \ast \bar{y}_p^p \leq k) \wedge (\lambda k \geq k \ast y_p \leq k)
\]

It requires intermediary lists of values \( \bar{y}_p^p \) and \( \bar{y}_p^p \), returned by partitioning \( \text{Pr}t(i, j) \), to be reorderings of the final values \( \bar{y}_p^p \) and \( \bar{y}_p^p \), where the values in \( \bar{y}_p^p \) are less than the pivot, \( \bar{y}_p^p \), and also that the values \( \bar{y}_p^p \) are greater than or equal to the pivot, \( \bar{y}_p^p \).

The proof for sequent (6.6) is derived from (6.7) and (6.8) by applying the derived rule \text{LCut} which logically sequentialises the two systems; then we apply \text{LInst} to substitute \( \bar{y}_p^p \) for \( \bar{y}_p^p \) in \( b \) (notice that the substitution leaves the pre/post-conditions and the system unchanged as \( \bar{y}_p^p \) are not free in them), then \text{LImpl} to recover the boolean condition \( \text{ord}(\langle y \rangle^i) \wedge \bar{y}_i^p \wedge \bar{y}_i^p ) \), then \text{LRes} to recover \( \Gamma \) from \( \Gamma_1 \), and finally \text{LCl} and \text{LDep} to recover \( [\text{Qck}(i, j)]_{\rho(r_1, r_j)} \).

The proof of sequent (6.8) follows from the following three sequents (6.9), (6.10) and (6.11) below, where \text{LRes} is used to extend \( \Gamma_1 \) as

\[
\Gamma_2 = \Gamma_1, r_1: \rho(r_1, i, p - 1), r_2: \rho(r_2, p + 1, j)
\]

to account for the mappings associated with the channels \( r_1 \) and \( r_2 \). Notice how this rule allows us to choose the permission association relating to \( r_1 \) and \( r_2 \) dynamically, depending on the index \( p \) returned by the partitioning phase of sequent (6.7). Such data dependencies normally complicates similar dependency analyses based on type systems such as \text{[45] [4].}

\[
\Gamma_2; b \vdash \left\{ A_i^p(\langle i \rangle^p) \ast r_1 \rangle \wedge \bar{y}_i^p \wedge \bar{y}_i^p \right\}
\]

\[
\Gamma_2; b \vdash \left\{ A_i^p(\langle i \rangle^p) \ast \rho_p(y_p) \ast A_{p+1}^j(\langle i \rangle^j) * r_2 \right\}
\]

\[
\Gamma_2; b \vdash \left\{ A_i^p(\langle i \rangle^p) \ast r_1 \rangle * r_2 \right\}
\]

Sequents (6.9) and (6.10) follow from the inductive hypotheses. Sequent (6.11) can be easily derived using \text{L FrmSt}, which eliminates \( A_i^p(\langle i \rangle^p) \) from the pre and post-conditions, and then applying \text{LIn} twice for \( r_1 \) and \( r_2 \) respectively, followed by applying \text{LOut} once for \( r_2 \); the two inputs on \( r_1 \) and \( r_2 \) would hand over the permissions \( \downarrow a_1, \ldots, \downarrow a_{p-1} \) and \( \downarrow a_{p+1}, \ldots, \downarrow a_j \) respectively; these are necessary for the output on \( r \) to be derived.

We recover the proof of sequent (6.8) as follows. Sequents (6.9) and (6.10) can be composed together as separate parallel code using \text{LSeP}, and then extended to include \( a_p(y_p) \) in the pre and post-conditions using \text{L FrmSt}. This allows us to logically sequence these two systems before the
system \([r_1?, r_2?, r_1!|1_{a_{1\rho}}, r_1!|1_{a_{2\tau}}, r_1!|1_{a_\tau}]\) of sequent \((6.11)\), thereby cutting the pre-condition of this sequent, using \(\text{lCUT}\). Then we scope the two channels \(r_1\) and \(r_2\) using a combination of \(\text{lRes}\), \(\text{lLcl}\) and \(\text{lSep}\) (which leaves the pre and post conditions intact since they do not contain any mention of the channels \(r_1\) and \(r_2\)), and finally precede this whole system by an input on \(r_3\) using \(\text{lIn}\), which adds \(r_3\langle p \rangle\) to the pre-condition.

This leaves us with only sequent \((6.7)\) to prove to complete the main proof. This sequent proof follows immediately from a proof for the following sequent

\[
\Gamma_1; b \vdash \left\{ A_{r_1}^j(\langle \vec{c}^j_1 \rangle - 1) \right\} \left[ \text{Trv}(i, j, x, i, i + 1) \right]_{\rho(r_3, i, j) \cup \{\tau\}} \left\{ \begin{array}{l} A_{p+1}^j(\langle \vec{d}^j_{p+1} \rangle - 1) \ast a_p(\chi_{r_3}) \\ * A_{p+1}^j(\langle \vec{z}^j_{p+1} \rangle - 1) \ast r_3(\langle p \rangle) \end{array} \right\} \tag{6.12}
\]

through one application of \(\text{lIn}\), which reinstates \(a_i(\langle x_i \rangle)\) in the pre-condition, then an application of \(\text{lDef}\) to recover \(\text{Prf}(i, j)\)\(\rho(r_3, i, j)\).

We prove \((6.12)\) by proving the more general sequent

\[
\Gamma_1; b' \vdash \left\{ A_{r_1}^j(\langle \vec{c}^j_1 \rangle - 1) \right\} \left[ \text{Trv}(i, j, x, i, c) \right]_{\rho(r_3, i, j) \cup \{\tau\}} \left\{ \begin{array}{l} A_{p+1}^j(\langle \vec{d}^j_{p+1} \rangle - 1) \ast a_p(\chi_{r_3}) \\ * A_{p+1}^j(\langle \vec{z}^j_{p+1} \rangle - 1) \ast r_3(\langle p \rangle) \end{array} \right\} \tag{6.13}
\]

where \(b' = b \wedge (i \leq q < c \leq j + 1) \wedge (\vec{c}^j_{i+1} = \vec{c}^j_{i+q+1}) \wedge (\bigwedge_{k=i+q+1}^{c-1} w_k < x_j) \wedge (\bigwedge_{k=i+q+1}^{c-1} x_i \leq w_k)\).

Sequent \((6.13)\) allows us to stratify every iteration of the traversal, thereby proving the sequent by induction on \(n = (j + 1) - c\). At each iteration, \(c\), with pivot index \(q\) and pivot value \(x_j\), \((6.13)\) expects a pre-condition split into 3 parts: \(A_{i+1}^q(\vec{c}^q_{i+1} \langle \vec{w}^q_{i+1} \rangle)\) holds processed values that are less than the pivot \(x_j\), \(i\), \(A_{i+1}^{i-q} \langle \vec{c}^i_{i+q} \rangle - 1\) holds processed values that are greater than or equal to the pivot \(x_j\), \(i\), and \(A_{i+1}^j(\vec{c}^j_{i+1})\) is the part of the array that still needs to be traversed. Note also that the values preceding the current counter, \(\vec{c}^i_{i+q}\), must be equal, up to reordering, of the values already processed \(\vec{c}^i_{i+1}\), \(i\).

The base case, i.e., when \(c = j + 1\) (and thus \(A_{i+1}^j(\vec{c}^j_{i+1}) = A_{i+1}^j(\vec{c}^j_{i+1}) = \text{emp}\)), establishes the post-condition in \((6.13)\) whereas the inductive case works up towards the base case, whereby the value comparison at every iteration adds to the ordering information expressed by \(b'\). Both proof cases use a mixture of rules \(\text{lIn}, \text{lOut}, \text{lF}\) and, \(\text{lSepSt}\) and \(\text{lCut}\) in a manner similar to that discussed already above; the details are left for the interested reader.

To obtain \((6.12)\) from \((6.13)\), we take \(q\) and \(c\) to be \(i\) and \(i+1\) respectively. This case makes the array assertions \(A_{i+1}^q(\vec{c}^q_{i+1})\) and \(A_{i+1}^{i-q} \langle \vec{c}^i_{i+q} \rangle\) in the pre-condition of \((6.13)\) empty, i.e., \(A_{i+1}^q(\vec{c}^q_{i+1}) = A_{i+1}^{i-q} \langle \vec{c}^i_{i+q} \rangle\) = \(\text{emp}\), which by Lemma \((4.9)\) and \(\text{lmp}\), leaves us with \(A_{i+1}^j(\vec{c}^j_{i+1})\) i.e., the pre-condition of \((6.12)\). Moreover, for this case the boolean expression \(b'\) is of the form \(b \wedge (i \leq i < i + 1 \leq j + 1)\) which is implied by \(b\), i.e., \(b \models b'\). This means that we can recover \(b\) for our sequent simply by applying \(\text{lmp}\) as well.

7. Conclusion

We have developed a logic for deterministic processes, interpreted over systems whose behaviour is confined by sets of linear permissions. We also developed a sound proof system through which we can determine, in compositional fashion, the satisfaction of formulas in this logic.
applied this logic and proof system to specify and prove the correctness of an in-place parallel quicksort.

7.1. Related Work. Modal logics have traditionally been used in process calculi for the specification of behavioural properties. Proof systems for these logics have been developed in a variety of settings (e.g., [20, 21, 12, 13, 4]) and some of these have focused on compositional reasoning as a means of dealing with the scalability problem (e.g., [2, 13, 4]). However, there has been little focus on locality of reasoning in these efforts. Approaching compositionality without necessarily modelling locality does seem to have been at the expense of general, but long-winded proof rules for parallel composition (e.g., [13]). In addition, termination is often not a major focus in these logics; in fact, the bisimulation proof technique, often associated with these logics, is insensitive to divergence. Termination is central to the logical characterisations that we give in this work.

Despite the apparent resemblance, spatial logics for process calculi such as [8, 9] differ from our interpretation of the separating conjunction: we separate on permissions, logical embellishments on processes, whereas their logical separation is more intensional and operates on the structure of processes, describing parallel composition directly. Moreover, their aims appear to differ from ours since they model mobility and channel privacy; we focus on data, non-interference and locality, and deal with implicit transfer of permissions.

Following [28], the use of separation logic to support local reasoning for concurrent programmes has been studied intensively over the past few years for the shared-variable model of concurrency. The initial main idea of ownership transfer of resources between threads impacting upon local reasoning already appears in [28]. This was then extended to co-exist with Rely/Guarantee reasoning [37, 15] and recently refined through fractional permissions as Deny/Guarantee reasoning [14]. The latter is interesting to us as a means of widening our class of programmes under analysis. For instance, [18] uses this approach for dealing with dynamically allocated resource locks.

Separation Logic has been applied to process calculi on at least two occasions. In [22], they give a separation semantics for a variant of the piCalculus, based on traces. Their work differs from ours in a number of respects in that they only deal with explicit ownership transfer of resources and are not concerned with developing a proof system. In [31], they also use a process calculus as a model for a separation logic. They are quite general wrt. the form of resources and how these are transferred across processes and, as a result, our model of confined processes seems related to theirs. However, aspects such as the use of SCCS on their part, where processes evolve in synchrony, and the focus on value passing and stability on ours, lead to a substantially different satisfaction relation of the logics. The aim of their work is also different from ours; they establish a correspondence between strong bisimulation and logic satisfaction whereas we focus on developing a compositional proof system. Separation logic has also been applied to an imperative concurrent language with message passing in [38] where the main focus is the implementability of message-passing communication as a copy-less communication over a shared memory model. Although their technical development is considerably different from ours, this work can be seen as complementary to ours if implementation aspects of our language are considered.

7.2. Future Work. There is much further work to be done in the area of local reasoning for message-passing concurrency.

With respect to the work presented here, there are a number of design decisions that are worth exploring. For instance, at the level of the proof system, a partial correctness interpretation of our sequents (as opposed to total correctness) would probably allow us to design a version of the parallel proof rule, \( \text{P}\text{A}\text{R} \), that is more symmetric. Another avenue worth exploring is that of relaxing
the interpretation of our logical assertions so as to not limit them to safely-stabilising systems. This would simplify the verification of certain formulas, such as any, and would also allow us to have models where formulas such as $c(v) * \text{blk}(c)$ are satisfiable. At the same time, this satisfaction weakening would also entail that our existing assertion interpretation changes to one where systems satisfy a formula at some point during their evaluation but may then fail to satisfy it as computation progresses. Although it is not yet clear whether this is a desirable property to have from the point of view of the application of the logic, it has appealing benefits in terms of the assertion satisfaction definition, as it streamlines the satisfaction of core formulas like the separating conjunction with existing interpretations. Moreover, we also conjecture that this altered interpretation would eliminate the need for the side conditions present in the existing parallel rule, $l\text{Par}$. At a more general level, we also seek to widen the class of programmes we can treat by introducing non-confluent behaviour in a controlled way. We intend to extend our setting to allow for more interesting forms of data to be communicated, including say channel names. We also need to develop algorithms for inferring the permission-set maps, develop tools to support the proof-system reasoning. Finally, and perhaps most importantly, we need to expand our suite of case studies and consider larger example proofs.

**Acknowledgment**

The authors wish to acknowledge numerous referees for their incisive comments on a preliminary version of this paper.

**References**

APPENDIX A. PROOFS

A.1. Processes.

Lemma A.1 (Structural Equivalence and Reductions).

\[ P \equiv Q \text{ and } P \rightarrow P' \implies \exists Q'. Q \rightarrow Q' \text{ and } P' \equiv Q' \]

Proof. By rule induction on \( P \equiv Q \).

Corollary A.2 (Structural Equivalence and Reductions).

\[ P \equiv Q \text{ and } P \rightarrow \implies Q \rightarrow \]

A.2. Confined Processes.

Lemma A.3. \( S \equiv T \) implies \( |S| \equiv |T| \)

Proof. By rule induction on \( S \equiv T \).

Corollary A.4. \( |S| \rightarrow \implies S \rightarrow \text{ or } (\exists T. S \rightarrow T \text{ and } |S| \equiv |T|) \)

Lemma A.17 (Properties of \( \equiv \) with respect to reductions).

1. \( S \equiv T \text{ and } T \rightarrow T' \text{ and } S \rightarrow_{err} \implies \exists S'. S \rightarrow S' \text{ and } S' \equiv T' \)
2. \( S \equiv T \text{ and } S \rightarrow \implies T \rightarrow \)

Proof. The first clause is proved by case analysis of \( T \rightarrow T' \) using Lemma A.5 to infer the structure of \( T \), then use the definition \( S \equiv T \) to determine the structure of \( S \). The second clause is proved by assuming that \( \exists T' \text{ such that } T \rightarrow T' \) and then use the first clause to show that this leads to a contradiction.

Lemma A.5 (Reduction and System Structure). \( S \rightarrow T \) implies

1. \( S \equiv (new \tilde{c}) \left( \left\{ c!e_1 \parallel \left\{ c?x. P \right\}_p \parallel R \right\} \right). T \equiv (new \tilde{c}) \left( \left\{ P[\tilde{v}/\tilde{x}] \right\}_p \parallel R \right). \tilde{c} \in \rho, \downarrow \tilde{c} \in \rho, \tilde{v} \downarrow \tilde{v} \text{ or;}
2. \( S \equiv (new \tilde{c}) \left( \left\{ if \, b \text{ then } P \text{ else } Q \right\}_p \parallel R \right). T \equiv (new \tilde{c}) \left( \left\{ P \right\}_p \parallel R \right) \text{ or } T \equiv (new \tilde{c}) \left( \left\{ Q \right\}_p \parallel R \right) \text{ or;}
3. \( S \equiv (new \tilde{c}) \left( \left\{ K(\tilde{c})[d_1/d_2] \right\}_p \parallel R \right). T \equiv (new \tilde{c}) \left( \left\{ P[\tilde{v}/\tilde{x}] \parallel d_1/d_2 \right\}_p \parallel R \right). \tilde{c} \downarrow \tilde{v} \text{ or;}
4. \( S \equiv (new \tilde{c}) \left( \left\{ P \parallel Q \right\}_p \parallel R \right). T \equiv (new \tilde{c}) \left( \left\{ P \parallel Q \right\}_p \parallel R \right) \text{ or;}
5. \( S \equiv (new \tilde{c}) \left( \left\{ (new \tilde{c}) P \right\}_p \parallel R \right). T \equiv (new \tilde{c}) \left( \left\{ (new \tilde{c}) P \right\}_p \parallel R \right) \text{ or;}
6. \( S \equiv (new \tilde{c}) \left( \left\{ nil \parallel R \right\} \right). S \equiv (new \tilde{c}) \left( \left\{ nil \parallel R \right\} \right). \rho \neq \emptyset \)

Proof. By rule induction on \( S \rightarrow T \).
Proposition 3.14 (Safe-Stability and System Structure).
\[ S \not\rightarrow \quad \text{iff} \quad S \equiv (\text{new } \vec{d}) \left( \|_i \varnothing \mid c_j \mid c'_j, \|_{\mu_j} \left[ c'_j ? x' P \right]_{\mu_j} \right) \]

where
\[ \left\{ c_1, \ldots, c_n \right\} \cap \left\{ c'_1, \ldots, c'_m \right\} = \emptyset. \]
\[ \land_{i=0}^n x c_i \in \rho_i \quad \land \quad \land_{j=0}^m x c_j \in \mu_j \]

and where \( \|_i \varnothing \mid c_j \mid \) and \( \|_{\mu_j} \left[ c'_j ? x' P \right] \) denote \( n \|_i \varnothing \).

Proof. Immediate by case analysis of Lemma [A.5] and then the conditions for \( S \rightarrow \text{err} \) from Figure 2.

Lemma 3.18 (Partial Confluence). \( S \rightarrow T_1 \) and \( S \rightarrow T_2 \) implies either of the following:
\[ (1) \quad T_1 \equiv T_2 \quad \text{or}; \]
\[ (2) \quad \exists T_3, T_1 \rightarrow T_3 \quad \text{and} \quad T_2 \rightarrow T_3 \]

Proof. By case analysis of the possible forms of \( S \) using Lemma [A.5] then restricting the possibilities using properties of well-formed systems. We here overview the two main cases.

- For \( S \rightarrow T_1 \) we have \( S \equiv (\text{new } \vec{d}) \left( \|_i c_j \mid c'_j, \|_{\mu_j} \left[ c'_j ? x' P \right]_{\mu_j} \right) \), \( T_1 \equiv (\text{new } \vec{d}) \left( \|_i c_j \mid c'_j, \|_{\mu_j} \right) \). If \( c_1 \in \rho_1 \) then \( c_1 \in \mu_1 \). Also for \( S \rightarrow T_2 \) we have \( S \equiv (\text{new } \vec{d}) \left( \|_i c_j \mid c'_j, \|_{\mu_j} \right) \), \( T_1 \equiv (\text{new } \vec{d}) \left( \|_i c_j \mid c'_j, \|_{\mu_j} \right) \).

- For \( S \rightarrow T_1 \) we have \( S \equiv (\text{new } \vec{d}) \left( \|_i P \mid c_j \mid c'_j, \|_{\mu_j} \right) \), \( T_1 \equiv (\text{new } \vec{d}) \left( \|_i P \mid c_j \mid c'_j, \|_{\mu_j} \right) \). If \( c_1 \in \rho_1 \), then \( c_1 \in \mu_1 \). Also for \( S \rightarrow T_2 \) we have \( S \equiv (\text{new } \vec{d}) \left( \|_i P \mid c_j \mid c'_j, \|_{\mu_j} \right) \), \( T_1 \equiv (\text{new } \vec{d}) \left( \|_i P \mid c_j \mid c'_j, \|_{\mu_j} \right) \).

The following technical Lemmas deal with the restricted non-determinism of confined processes and how it can be characterised using the relation \( \equiv \). In particular, Lemma [A.8] is useful because it allows us to correct reductions that lead to systems that do not evaluate by instead reducing to systems that are related to them by \( \equiv \), which in turn means, by Proposition 3.16 that they contain the same process structure.

Lemma A.6. \( S \parallel S \) and \( S \rightarrow T \) and \( T \parallel S \) implies \( \exists P, Q, R. \quad S \equiv (\text{new } \vec{d}) \left( \|_i P \mid c_j \|_{\mu_j} \right) \) and \( T \equiv (\text{new } \vec{d}) \left( \|_i Q \mid c_j \|_{\mu_j} \right) \)

Proof. By induction on the number of reductions in \( S \parallel S \) leading to a safely-stable system \( i.e., S \rightarrow S' \not\rightarrow \) for some \( S' \).

\( n = 1 \): By Lemma 3.18 and \( S' \not\rightarrow \) \( (i.e., S \not\rightarrow) \) it must be the case that \( T \equiv S' \). By Lemma 3.17.2 this also implies \( T \rightarrow \) and since \( T \parallel S \) it must be the case that \( T \rightarrow \text{err} \). Now by case analysis of Lemma A.5, the only system structure that allows this is when \( S \equiv (\text{new } \vec{d}) \left( \|_i P \mid c_j \|_{\mu_j} \right) \) and \( T \equiv (\text{new } \vec{d}) \left( \|_i Q \mid c_j \|_{\mu_j} \right) \).
\[ n = k + 1: \] We have \( S \parallel S \rightarrow S'' \rightarrow^k S' \). By Lemma \[3.18\] we have two sub-cases. The first case subsumes the second in some cases, so we here consider the mutually exclusive variants:

- If \( T \rightarrow T' \) and \( \rightarrow S'' \rightarrow^k S' \), this implies \( T \rightarrow T' \), and \( S'' \rightarrow^k S' \). By Lemma \[3.18\] we know \( T \rightarrow T' \), and \( S'' \rightarrow^k S' \). Thus by that fact that \( S \equiv (\text{new} \, \vec{c} \, [P \parallel Q_{[p]} \parallel S']) \), and \( T \equiv (\text{new} \, \vec{c} \, [P' \parallel Q_{[p]} \parallel S'']) \), we have \( T \rightarrow T' \) and \( S'' \rightarrow^k S' \). Clearly, since \( T \rightarrow T' \), and \( S'' \rightarrow^k S' \). Also the fact that there is no common system \( T \) and \( S'' \) can reduce to means that the reductions from \( S \) where not from separate redexes. By case analysis of Lemma \[A.5\] the only possible option for having non-deterministic reductions from the same redex is the case where \( \equiv (\text{new} \, \vec{c} \, [P \parallel Q_{[p]} \parallel S']) \), and \( T \equiv (\text{new} \, \vec{c} \, [P' \parallel Q_{[p]} \parallel S'']) \).

**Lemma A.7.** \( S \parallel S \equiv (\text{new} \, \vec{c} \, [P \parallel Q_{[p]} \parallel R]) \) implies \( \exists \mu_1, \mu_2 \text{ such that } \mu_1 \parallel \mu_2 = \rho \) and

\( (\text{new} \, \vec{c} \, [P \parallel Q_{[p]} \parallel R]) \).

**Proof.** By induction on the number of reductions in \( S \parallel S \) leading to a safely-stable system i.e., \( S \rightarrow^n S' \). For some \( S'' \).

- If \( n = 1 \): By Lemma \[3.18\] we know \( S \equiv (\text{new} \, \vec{c} \, [P \parallel Q_{[p]} \parallel R]) \) can reduce to \( (\text{new} \, \vec{c} \, [P' \parallel Q_{[p]} \parallel R]) \). By Lemma \[3.18\] and \( S'' \rightarrow^k S' \), we must have \( S'' \equiv (\text{new} \, \vec{c} \, [P' \parallel Q_{[p]} \parallel R]) \). Thus by Lemma \[3.18\] we have \( \exists \mu_1, \mu_2 \) such that \( \mu_1 \parallel \mu_2 = \rho \).

- If \( n = k + 1 \): We have \( S \rightarrow^k S'' \rightarrow^k S' \). By Lemma \[A.5\] we have two sub-cases:

  1. \( T \rightarrow T' \) and \( \rightarrow S'' \rightarrow^k S' \): By Lemma \[A.5\] we have \( \exists \mu_1, \mu_2 \) such that \( \mu_1 \parallel \mu_2 = \rho \).
  2. \( T \rightarrow T' \) and \( \rightarrow S'' \rightarrow^k S' \): By Lemma \[A.5\] we have \( \exists \mu_1, \mu_2 \) such that \( \mu_1 \parallel \mu_2 = \rho \).

**Lemma A.8 (Corrective Reductions).** \( S \parallel S \rightarrow T \) and \( T \parallel S \) implies \( \exists R \text{ such that } S \rightarrow R \) and \( R \equiv T \) and \( R \parallel S \).

**Proof.** By Lemma \[A.4\] we know \( S \equiv (\text{new} \, \vec{c} \, [P \parallel Q_{[p]} \parallel S']) \), and \( T \equiv (\text{new} \, \vec{c} \, [P' \parallel Q_{[p]} \parallel S']) \). By Lemma \[A.7\] we know \( \exists \mu_1, \mu_2 \) such that \( \mu_1 \parallel \mu_2 = \rho \). Since \( T \equiv (\text{new} \, \vec{c} \, [P' \parallel Q_{[p]} \parallel S']) \), this implies that we can correct the permission split and be able to reduce to a safely-stable state.

In order to apply corrective actions to multiple reduction steps, we need to extend Lemma \[A.8\] to systems that are related by \( \equiv \), due to reductions of type (1) of Lemma \[3.18\]. The next Lemmas deal with this.

**Lemma A.9 (Evaluation Preservation for \( \equiv \)).** \( S \equiv T \) and \( S \parallel S \rightarrow T' \) implies \( \exists S' \) such that \( S \rightarrow S' \) where \( S' \equiv T' \) and \( S' \parallel S \).

**Proof.** By Lemma \[3.7\] we have \( S \rightarrow \rightarrow \) and by Lemma \[3.17\] we know \( \exists S_1 \) such that \( S \rightarrow S_1 \) and \( S_1 \rightarrow^* T' \). At this point we have two sub-cases: if \( S \parallel S_1 \), then the result follows immediately. Otherwise, if \( S \parallel S_1 \), then Lemma \[A.8\] states that \( \exists S_2 \) such that \( S \rightarrow S_2 \) and \( S_2 \equiv S_1 \). By transitivity we have \( S_2 \equiv S_1 \rightarrow^* T' \).
Lemma A.9 (Evaluation Preservation for $\equiv$). \( S \equiv T \) and \( S \parallel T \) implies \\
\( \exists S' \) such that \( S \longrightarrow^* S' \) \( \equiv \) \( T' \longrightarrow^* T' \).

Proof. By induction on \( n \), the number of reductions in \( T \longrightarrow^* T' \).

\( n = 0 \): Immediate.

\( n = k + 1 \): We have \( T \longrightarrow T'' \longrightarrow^* T' \). From \( T \longrightarrow T'' \) and Lemma 3.20 we obtain \( \exists S' \) such that \( S \longrightarrow S' \) \( \equiv \) \( T' \longrightarrow S'' \) \( \equiv \) \( T'' \longrightarrow S'' \). By I.H. we know \( S' \longrightarrow^* S'' \) for some \( S'' \) such that \( S'' \equiv T'' \) and \( S'' \parallel \). 

Lemma A.10. \( |S| \equiv Q \) and \( S \parallel \) implies \( \exists T \) such that \( S \longrightarrow^* T \) and \( T \parallel \) and \( |T| = Q \).

Proof. By rule induction on \(|S| \equiv Q\).

Lemma A.11. \( |S| \equiv Q \) implies \( \exists T. S \longrightarrow^* T \) or \( S \equiv T \) \( S \) where \(|T| = Q \).

Proof. By rule induction on \(|S| \equiv Q\) and then a tedious consideration of all the possible permutations of \( S \) that may lead to \(|S|\).

sAss:: If \(|S| = |P_1||P_2|P_3\) then \( Q = |P_1||P_2|P_3\) and \( S \) can be either of the following:

\[ S = |P_1||P_2||P_3| : \text{ By 2 applications of cStr then an application of cStr using scAss we obtain } S \longrightarrow^* ([P_1]|P_2||P_3|) \text{ where } \rho_1 \lor \rho_2 \lor \rho_3 = 0 \text{ and } |([P_1]|P_2||P_3|) = Q \]

\[ S = [P_1]|P_2||[P_3]| : \text{ By one application of cStr} \text{ and one application of cStr using scAss we obtain } S \longrightarrow^* ([P_1]|P_2||[P_3]) \text{ where } \rho_1 \lor \rho_2 \lor \rho_3 = 0 \text{ and } |([P_1]|P_2||[P_3]) = Q \]

The symmetric case where \(|S| = |P_1||P_2|P_3\) and \( Q = |P_1||P_2|P_3\) is similar.

sCom:: Similar to sAss case.

sNu:: If \(|S| = |P||\nu|\) and \( Q = P \) then we have two cases:

\[ S = |P||\nu| : \text{ By cStr, cStr and scNu we obtain } S \longrightarrow^* |P| \text{ and } |P| = P \]

\[ S = |P||\nu| : \text{ By cStr, cStr and scNu we obtain } S \longrightarrow^* |P| \text{ and } |P| = P \]

If \(|S| = |P||\nu|\) then by scNu we have \( S \equiv S \parallel \) and \(|S| \equiv |\nu| = Q \).

sNew:: The most difficult case is when \( S = |(new c)|\nu|\) and \( Q = \nu|\). By cLcl, cLcl, cStr with scNew we obtain \( S \longrightarrow^* |\nu| \) and \(|\nu| = Q \). The other cases are similar.

sExpr:: When \(|S| = |P||new c|Q\) we have three cases: \( S = |P||new c|Q|\), \( S = |P|\), \( |new c|Q|\), and \( S = |P| |new c|Q|\) and the proof follows using the rules cStr, cLcl, cStr and scExt.

The symmetric case when \(|S| = |new c|P|Q\) is similar.

Lemma 3.27 (Reduction Correspondence).

\( S \parallel \) and \( S' \longrightarrow Q \) implies \( \exists R \) \( S \longrightarrow^* R \) and \(|R| \equiv Q \)

Proof. By rule induction on \(|S| \longrightarrow Q\). We here consider the main cases:

rCom: We have \(|S| = e[\nu]\langle c ? x . P \rangle P \) and \( Q = P[\nu][\gamma] \) where \( e[\nu]\rangle \). We have two sub-cases for \( S \):

\[ S = [e\langle c ? x . P \rangle P] : \text{ By } \gamma \], \( \exists \mu_1, \mu_2 \) such that \( \mu_1 \lor \mu_2 \lor \mu_3 = 0 \) and \( S \longrightarrow [e\langle c ? x . P \rangle P] \longrightarrow [P[\nu][\gamma]] \text{ and } |[P[\nu][\gamma]] = Q \]

rPar: We have \(|S| = \gamma \) and \( Q = P[\nu][\gamma] \) because \( P_1 \longrightarrow P_1' \). We have two sub-cases for \( S \):

\[ S = [P_1][P_2] : \text{ By } \exists \mu_1, \mu_2 \) such that \( \mu_1 \lor \mu_2 \lor \mu_3 = 0 \) and \( [P_1][P_2] \longrightarrow [P_1][P_2] \text{ and } |[P_1][P_2] = Q \]

\[ S = S_1 \parallel S_2 \) \( S_1 \) and \( S_2 \) \( S_2 \) are similar
Lemma A.12. When \( \phi \rightarrow \) and \( \Gamma, S \models \phi \)
- \( S \equiv \exists c \left[ \overrightarrow{ \rho } \phi \right] \) implies \( \uparrow c \in \text{edg}(\phi) \) or \( \text{edg}(\phi) \) is undefined;
- \( S \equiv (\text{new} \overrightarrow{d}) \left[ \exists X \phi \right] \) implies \( \uparrow c \in \text{trg}(\phi) \) or \( \text{trg}(\phi) \) is undefined.

Proof. By induction on the structure of \( \phi \). □

Lemma A.13. \( \Gamma, S \models \psi \), \( S \rightarrow \) and \( \Gamma, T \models \psi \), \( T \rightarrow \) implies \( \Gamma, S \parallel T \rightarrow \)

Proof. Since \( S \rightarrow \) and \( T \rightarrow \), by Lemma A.5, we know that \( S \parallel T \rightarrow R \) for some \( R \) can only happen if:
- \( S \equiv (\text{new} \overrightarrow{d}) \left[ \exists X \phi \right] \) \( \parallel S' \) where \( c \not\in \overrightarrow{d} \) and \( \downarrow c \in \mu \)
- \( T \equiv \left[ c! \overrightarrow{\rho} \right] \parallel T' \) where \( \uparrow c \in \rho \)

or vice-versa. We here focus on the case where (A.1) and (A.2) have to hold; the dual case is identical. By \( \psi \vdash \psi \) we know that \( \text{trg}(\psi) \), \( \text{edg}(\psi) \), \( \text{trg}(\psi) \) and \( \text{edg}(\psi) \) must all be defined. Thus by (A.1), \( \Gamma, S \models \psi \) and Lemma A.12 we must have \( \uparrow c \in \text{trg}(\psi) \). Similarly by (A.2), \( \Gamma, T \models \psi \) and Lemma A.12 we must have \( \uparrow c \in \text{edg}(\psi) \). This however would contradict \( \psi \vdash \psi \) which requires that \( \text{trg}(\psi) \cap \text{edg}(\psi) = \emptyset \). Thus \( S \parallel T \rightarrow \). □

Lemma 4.11 (Merging Assertions). \( \Gamma, S \parallel \psi \) and \( \Gamma, T \parallel \psi \) and \( \psi \vdash \psi \) implies \( \Gamma, S \parallel T \vdash \psi \)

Proof. \( S \parallel T \vdash \psi \) implies \( S \parallel T \) is well-resourced. From \( \Gamma, S \parallel \psi \), \( \Gamma, T \parallel \psi \) and Proposition 4.5, we know that \( S \parallel S' \) and \( T \parallel T' \) where \( \Gamma, S' \models \psi \) and \( \Gamma, T' \models \psi \). Lemma A.13, we know also that \( S \parallel T \parallel S' \parallel T' \) and the result follows by satisfaction on Figure 3. □

A.4. The Proof System. Proofs for the derived rules from Section 5.1

The proof for LCut:

\( \text{LIMP} \quad \Gamma; b \vdash \{ q_1 \} S \parallel \{ \psi \} \) \quad \text{LIMP} \quad \Gamma; b \vdash \{ \psi \} T \parallel \{ q_1 \} \)

\( \text{LIMP} \quad \Gamma; b \vdash \{ q_1 \} S \parallel \{ \psi \} \) \quad \text{LIMP} \quad \Gamma; b \vdash \{ \psi \} T \parallel \{ q_1 \} \)

The proof for LStr:

\( \text{LIMP} \quad \Gamma; b \vdash \{ q_1 \} S \parallel \{ \psi_1 \} \) \quad \text{LIMP} \quad \Gamma; b \vdash \{ \psi_2 \} T \parallel \{ q_2 \} \)

\( \text{LIMP} \quad \Gamma; b \vdash \{ q_1 \} S \parallel \{ \psi_1 \} \) \quad \text{LIMP} \quad \Gamma; b \vdash \{ \psi_2 \} T \parallel \{ q_2 \} \)

\( \text{LIMP} \quad \Gamma; b \vdash \{ q_1 \} S \parallel \{ \psi_1 \} \) \quad \text{LIMP} \quad \Gamma; b \vdash \{ \psi_2 \} T \parallel \{ q_2 \} \)
The proof for $\text{LOutD}$:
\[
\begin{align*}
\frac{\Gamma(c) \subseteq \rho 
\quad \exists \bar{x}, \bar{e}_1 = e_1^1 \land e_2^1 = e_2^2 \quad \bar{x} \notin \text{fn}(b) \cup \text{fn}(e_2^2, e_1^1) 
\quad \exists \bar{x}, \bar{e}_1 = e_1^1 \land e_2^1 = e_2^2 
\quad \bar{x} \notin \text{fn}(b) \cup \text{fn}(e_2^1, e_1^1)}{\Gamma; b \vdash \exists \bar{x}, \bar{e}_1 = e_1^1 \land e_2^1 = e_2^2}
\end{align*}
\]
\[
\frac{\Gamma \vdash [\text{emp}] e_1^1 \Gamma \vdash [\text{emp}] e_2^2}{\Gamma; b \vdash [\text{emp}] e_1^1 \Gamma \vdash [\text{emp}] e_2^2}
\]

The proof for $\text{LIsD}$:
\[
\begin{align*}
\frac{\Gamma; b; \vdash \exists x \in \rho \quad \Gamma; b; \vdash \exists x \in \rho}{\Gamma; b; \vdash \exists x \in \rho}
\end{align*}
\]
\[
\frac{\Gamma, T; \vdash \exists x \in \rho}{\Gamma, T; \vdash \exists x \in \rho}
\]

Lemma A.14. Assume that $\llbracket \rho \rrbracket_{\Gamma}$ is a substitution that non-deterministically substitutes either $e$ or $v$ for $x$. Then we have
\[
S \llbracket x \rrbracket \rightarrow T \llbracket x \rrbracket
\]
either implies $S \llbracket \rho \rrbracket \rightarrow R$ where $R = T \llbracket \rho \rrbracket$
for some non-deterministic substitution $T \llbracket \rho \rrbracket$

Proof. By rule induction on $S \llbracket x \rrbracket \rightarrow T \llbracket x \rrbracket$

Lemma A.15. $\Gamma, T \llbracket \rho \rrbracket \models \exists x \in \rho$ implies $\models \exists x \in \rho$

Proof. By induction on the structure of $\exists x \in \rho$

Lemma A.16. $\Gamma, T \llbracket \rho \rrbracket \models \exists x \in \rho$, $\models \exists x \in \rho$

Proof. Follows from Lemma A.14 and Lemma A.15

Lemma A.17. $\models \exists x \in \rho$ and $d \notin \text{dom}(\Gamma)$ implies $\Gamma, (\text{new } d) \models \exists x \in \rho$

Proof. By induction on the structure of $\exists x \in \rho$. For instance:
\[
c(\vec{c})\quad \text{We know } S \llbracket c(\vec{c}) \rrbracket \models \exists x \in \rho
\]
and $\Gamma(c) \subseteq \rho$. By $c\text{Res}$ and then by $c\text{Tom}$ and $d \notin \text{nm}(c(\vec{c})) \cup \text{nm}(\Gamma)$ we deduce
\[
(\text{new } d) \models \exists x \in \rho
\]

Definition A.18 (Permission Restriction).
\[
S \setminus \mu \overset{\text{def}}{=} \begin{cases} 
\llbracket P \rrbracket_{\mu} & \text{if } S = \llbracket P \rrbracket_\rho \\
\llbracket \llbracket S_1 \setminus \mu \setminus \mu \rrbracket_{\mu} \llbracket S_2 \setminus \mu \rrbracket_{\mu} & \text{if } S = \llbracket S_1 \rrbracket \llbracket S_2 \rrbracket \rrbracket_{\mu} \\
\llbracket (\text{new } c)(T \setminus \mu \setminus \{c\}((\mu \setminus \{c\}) \cap \{c_1, \ldots, c_n\} = \emptyset)) & \text{if } S = \llbracket \text{new } c \rrbracket T
\end{cases}
\]

Proposition A.19. $S \setminus \mu \rightarrow_{\text{err}}$ implies $S \rightarrow_{\text{err}}$

Lemma A.20. $S \setminus \mu \rightarrow_{\text{err}}$ implies $\exists T. S \rightarrow_{\text{err}}$ where $(T \setminus \mu) \equiv (S \setminus \mu)$

Proof. By Proposition A.14 we know,
\[
S \setminus \mu \equiv \llbracket \text{new } d \rrbracket \llbracket c(\vec{c}) \rrbracket_{\mu} = \emptyset
\]
where $\{c_1, \ldots, c_n \cap \{c_1, \ldots, c_m\} = \emptyset
\]
By system structural equivalence, \(\equiv\), the only sub-systems in \(S\) that are abstracted away from \(S \setminus \mu\) in \(\langle\text{new} \vec{d}\rangle \left(\parallel_{i=0}^{n} [c_i; e_i]_{\rho_i} \parallel_{j=0}^{m} [c'_j ? x_j; P_j]_{\mu_j}\right)\) are those of the form \([\text{nil}]_{\rho}\) where \(\rho \subseteq \mu\); the operation made these systems equivalent to \([\text{nil}]_{\emptyset}\) which could then be eliminated through scNu. In \(S\), sub-systems of the form \([\text{nil}]_{\rho}\) can still be eliminated through cDsc and then scNu. (as before), leaving us with the same array of mismatching confined output and input processes found in \(S \setminus \mu\), less the restricted permissions.

**Lemma A.21.** \(S \setminus \mu \rightarrow T \setminus \mu\) implies \(S \rightarrow T\)

**Proof.** By rule induction on \(S \setminus \mu \rightarrow T \setminus \mu\).

**Lemma A.22.** \((S \setminus \mu) \parallel T\) implies \(S \parallel R\) where \(R \setminus \mu \equiv T\)

**Proof.** Follows from Lemma A.21, Lemma A.20 and Proposition A.19.

**Lemma A.23.** \(\Gamma, (S \setminus \mu) \models \phi\) implies \(\Gamma, S \models \phi\)

**Proof.** By induction on the structure of \(\phi\) using Lemma A.22.