Concurrent Self-Explaining Computation

MPI-SWS Technical Report 2013-004 (July 2013)

Roly Perera
University of Edinburgh
rperera@inf.ed.ac.uk

Deepak Garg
MPI-SWS
dg@mpi-sws.org

Umut Acar
Carnegie Mellon University
umut@cs.cmu.edu

Abstract

Self-explaining computation is an approach to program execution in which every value comes with an explanation of how it was computed. The explanation can be used to reverse the computation and to slice the original program relative to any part of the output of interest. As a result, self-explaining computation is a suitable foundation for offline dynamic program analyses such as taint analysis and algorithmic debugging.

Building on prior work in the functional setting, we develop the foundations of concurrent self-explanation for a higher-order process calculus. We represent explanations as traces, which record inter-process synchronisation as well as intra-process functional evaluation. We show that any part of the state of a concurrent computation can be accounted for by a unique minimal slice of the initial configuration. This result is established using a Galois connection describing forward and backward executions of a concurrent program that share a synchronisation structure. As our main practical result, we provide a reverse operational semantics for processes which computes the lower adjoint of the Galois connection.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and debugging—Tracing; D.3.4 [*Programming Languages*]: Processors—Debuggers

Keywords concurrency; reversible computation; program slicing

1 Introduction

We explore concurrent *self-explaining* computation, a paradigm in which the evaluation of a concurrent program yields not just outputs but also an explanation of how each output was computed. This explanation, represented as a trace, can be used to execute backwards to obtain, for any particular part of the output, the least slice of any prior state which is able to compute it. Self-explaining computation has immediate applications in debugging, provenance analysis and other forms of offline dynamic analysis where relying on re-execution as means of recovering computational history is not always practical. Building on prior work on self-explaining computation for sequential programs [12, 13], we develop the mathematical foundation of self-explanation for concurrent computation, including a tracing semantics, and a reverse operational semantics based on traces.

Our formal setting is a higher-order process calculus with an instrumented reduction semantics that records inter-process synchronization events (send-receive events) in a *causality graph*, a form of dependence graph. This simple structure allows us to reverse execution whilst preserving causal consistency with the original execution, usually considered an essential feature of reversible process calculi [4, 5, 9, 14].

More interestingly, if we restrict attention to any specific part of an output (say, because it is the focus of interest when debugging), then there exists a *unique minimal part* of any prior configuration that computes this part exactly. The broad intuition is that computing least dynamic slices is decidable when a language exhibits sufficient *sequentiality* to allow for the existence of the required minima, and non-trivial when it exhibits enough *parallelism* to permit parts of the program to be independent.

Formally, the existence of unique minimal parts in prior configurations is established as a Galois connection. We consider the partial order over configurations induced by erasure (a more erased configuration is smaller) and establish that evaluation, extended in a natural way to partial configurations, preserves meets. Hence, evaluation has a lower adjoint, termed *unevaluation*, whose image on a configuration is the unique minimal slice. We then generalise this result to causally equivalent runs of a concurrent program, forming the foundation of our work. A priori, a Galois connection for concurrency is not particularly plausible, because reduction is non-deterministic. However, by restricting reduction to comply with a given causality graph, we obtain enough determinism to define the Galois connection, yet retain enough internal non-determinism to permit an efficient concurrent implementation.

To *implement* unevaluation, we define a tracing semantics which augments the causality graph with traces of internal functional evaluations and other information about process states which if discarded would be a source of irreversibility. Unevaluation is then implemented as concurrent, but deterministic, *backward execution* along the trace, starting from the output, computing the minimum necessary information at each prior step and merging information where forward evaluation forked. For slicing through functional computation within a process, we use Perera et al.'s work on self-explaining computation for the sequential functional setting [12, 13] with only minor changes.

In summary, our work studies the mathematical foundations of self-explaining concurrency and establishes the existence and computability of least program slices for concurrent systems. We make the following contributions.

- We characterise the existence of unique minimal backward slices of concurrent computations as Galois connections, first on single reductions, and then on causally equivalent runs of a concurrent program (§3).
- We introduce a form of trace suitable for explaining non-deterministic concurrent computations. By design syntactic identity on traces coincides with causal equivalence for a given program (§4.2).

- We define a concurrent reverse execution semantics for processes, based on traces, and prove that it implements the unevaluation lower adjoint (§4.3).

We also improve on [13] in some minor ways: we dispense with both types and environments, neither of which prove essential; and we specify the abstract requirements on pluggable components, showing how to extend our system in a modular way without compromising precision or correctness.

§5 compares self-explaining computation to prior work on dynamic slicing and debugging of concurrent programs, reversible process calculi, and distributed provenance and tracing. §6 summarises our findings and identifies some important future work. The Appendix contains a treatment of mutually recursive functions which improves on earlier work, plus proofs and other technical details omitted from the paper. Colour is useful, but not essential, for reading the paper.

2 Concurrent setting

The language that will form the setting for the rest of the paper is a process calculus with an embedded call-by-value functional language. We describe the functional part first.

2.1 Expressions

Expressions e, defined in Figure 1, include the usual functional constructs, plus *suspended processes*, written $\{P\}$, which can be ignored for the moment. Terms are identified up to renaming of bound identifiers. The only notable feature of the syntax is the form v_x, which represents a value v which was substituted for a variable x; we will use this later to implement a reversible form of substitution.

$$\begin{array}{lrl} \text{Expression} & e & ::= \; () \mid x \mid v_x \mid \lambda x.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \\ & & \quad \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \{\mathsf{inl}\ x_1.e_1; \mathsf{inr}\ x_2.e_2\} \mid \{P\} \\ \text{Value} & u, v & ::= \; c \mid () \mid \lambda x.e \mid (v_1, v_2) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \{P\} \end{array}$$

Figure 1. Expressions and values (processes omitted)

Figure 1 also defines the values u, v of the language. Values include suspended processes $\{P\}$, as well as channels c.

Expression evaluation Expression evaluation is big-step, and defined as the deterministic relation $e \Rightarrow v$ given in Figure 2. Rules for snd and inr are omitted. One important detail is that a process suspension $\{P\}$ evaluates immediately to the suspended process $\{P\}$ as a value; there is no evaluation of P itself, so the expression language remains deterministic.

We write $e\{v/x\}$ for the expression e with v substituted for x. The definition of substitution is mostly standard; we mention only three rules. The variable case has $x\{v/x\} = v_x$, preserving the identifier in the substituted expression; the value case has $u_y\{v/x\} = u_y$, ignoring the variable annotation on the value even when $x = y$. Again the significance of these rules will become apparent later. The suspended process case has $\{P\}\{v/x\} = \{P\{v/x\}\}$. (Values are closed.)

2.2 Processes

The syntax of processes is given in Figure 3, and includes the stop process $\mathbf{0}$, channel creation $\nu x.P$, parallel composition $P_1 \parallel P_2$, asynchronous send $e_1\langle e_2\rangle$ to send the value of e_2 on the channel computed by e_1, receive $e(x).P$ to wait for a value on the channel computed by e, and run e which runs P if e evaluates to the suspended process $\{P\}$.

$\boxed{e \Rightarrow v}$

$$\frac{}{() \Rightarrow ()} \qquad \frac{}{v_x \Rightarrow v} \qquad \frac{}{\lambda x.e \Rightarrow \lambda x.e}$$

$$\frac{e_1 \Rightarrow \lambda x.e_3 \qquad e_2 \Rightarrow v_2 \qquad e_3\{v_2/x\} \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{(e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \frac{e \Rightarrow (v_1, v_2)}{\mathsf{fst}\ e \Rightarrow v_1} \qquad \frac{e \Rightarrow v}{\mathsf{inl}\ e \Rightarrow \mathsf{inl}\ v}$$

$$\frac{e \Rightarrow \mathsf{inl}\ v_1 \qquad e_1\{v_1/x_1\} \Rightarrow v}{\mathsf{case}\ e\ \{\mathsf{inl}\ x_1.e_1; \mathsf{inr}\ x_2.e_2\} \Rightarrow v} \qquad \frac{}{\{P\} \Rightarrow \{P\}}$$

Figure 2. Expression evaluation

$$\begin{array}{lrl} \text{Process} & P & ::= \; \mathbf{0} \mid \nu x.P \mid P_1 \parallel P_2 \mid e_1\langle e_2\rangle \mid e(x).P \mid \mathsf{run}\ e \\ \text{Process id} & \alpha, \beta & ::= \; \ell \mid \alpha.1 \mid \alpha.2 \mid \alpha.\beta \\ \text{Configuration} & C, D & ::= \; [\![P_1]\!]^{\alpha_1}, \ldots, [\![P_n]\!]^{\alpha_n} \end{array}$$

Figure 3. Processes, process ids and configurations

A configuration is a collection of processes with unique *process ids* α. Formally, a configuration is a finite map C from process ids to processes $C(\alpha)$ is called a *configuration*. For a set of process ids Γ, we write $\Gamma \vdash C$ to indicate that C has domain Γ. We often write out a configuration explicitly as $[\![P_1]\!]^{\alpha_1}, \ldots, [\![P_n]\!]^{\alpha_n}$, in which case $\alpha_1, \ldots, \alpha_n$ are assumed to be distinct. Process ids allow syntactically equal processes to be distinguished; this will be important later for ensuring that reverse execution respects the synchronisation structure of the original computation. Forgetting the ids recovers a standard multiset configuration.

Process evaluation. The operational semantics of processes are small-step, and given in Figure 4. The relation $\twoheadrightarrow$ defines a deterministic notion of reduction for configurations C, where C is a redex. If $C \twoheadrightarrow C'$ we say that C *reduces to* C'. We separately specify how to non-deterministically choose a redex from a larger configuration.

The reduction rules are mostly self-explanatory; we discuss only the rules that modify the domain of the configuration, i.e. the set of active process ids. STOP kills α; killed processes are deleted from the configuration. FORK kills α and spawns $\alpha.1$ and $\alpha.2$ to run P_1 and P_2 respectively. JOIN kills α and β, which are waiting to send on c and waiting to receive on c respectively, and spawns $\alpha.\beta$ to run the body of the receiver. For the NEW rule, we also note that terms are identified up to renaming not just of bound identifiers, but also of channels, which we treat as global. Therefore the selection of a fresh c is deterministic; in fact the only non-confluent source of non-determinism in our language is inter-process synchronisation.

To obtain a full concurrent execution semantics, we lift the $\twoheadrightarrow$ relation to operate on an arbitrary configuration $[\![P_1]\!]^{\alpha_1}, \ldots, [\![P_n]\!]^{\alpha_n}$ and to take multiple steps. (Note that the multiset $P_1, \ldots, P_n$ is a typical configuration of the chemical abstract machine [2]. Our system can always be viewed as operating on this multiset, and so structural rewrite rules such as $P_1 \parallel P_2 \rightsquigarrow P_2 \parallel P_1$ are not required.)

First, we introduce the notion of a *causality graph*. A causality graph is used to capture the "causal" or (observable) "happens-before" relation for a concurrent computation [5], and comes in one of two forms. An *atomic* causality graph Δ captures the causal structure of an individual reduction, and has one of the forms given in Figure 5 below; with one exception (corresponding to

$C \twoheadrightarrow C'$

STOP — where:
$[\![\mathbf{0}]\!]^\alpha \twoheadrightarrow \varnothing$

NEW
$[\![\nu x.P]\!]^\alpha \twoheadrightarrow [\![P\{c/x\}]\!]^\alpha$ — c fresh

FORK
$[\![P_1 \parallel P_2]\!]^\alpha \twoheadrightarrow [\![P_1]\!]^{\alpha.1}, [\![P_2]\!]^{\alpha.2}$

SEND-ON
$[\![e_1\langle e_2\rangle]\!]^\alpha \twoheadrightarrow [\![c\langle e_2\rangle]\!]^\alpha$ — $e_1 \Rightarrow c$

SEND-READY
$[\![c\langle e\rangle]\!]^\alpha \twoheadrightarrow [\![c\langle v\rangle]\!]^\alpha$ — $e \Rightarrow v$

RCV-READY
$[\![e(x).P]\!]^\alpha \twoheadrightarrow [\![c(x).P]\!]^\alpha$ — $e \Rightarrow c$

JOIN
$[\![c\langle v\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta \twoheadrightarrow [\![P\{v/x\}]\!]^{\alpha.\beta}$

RUN
$[\![\text{run } e]\!]^\alpha \twoheadrightarrow [\![P]\!]^\alpha$ — $e \Rightarrow \{P\}$

$C \twoheadrightarrow_\Delta C'$

$$\frac{C \to C'}{C \twoheadrightarrow_\Delta C'}\ \mathsf{in}(\Delta) \vdash C', \mathsf{out}(\Delta) \vdash C''$$

$C \to_G C'$

$$\frac{}{C \to_\bullet C} \qquad \frac{C \to_G D \uplus D' \qquad D \twoheadrightarrow_\Delta D''}{C \to_{G \fatsemi \Delta} D \uplus D''}$$

Figure 4. Process evaluation

a STOP reduction), nodes are labelled with process ids. A (general) causality graph G is then a finite graph in which every neighbourhood is an atomic causality graph. The purpose of casuality graphs will become clear as we explain the semantics.

Figure 5. Atomic causality graphs Δ (directed from left to right)

Next, writing $\mathsf{in}(\Delta)$ for the labels on the roots of Δ, and $\mathsf{out}(\Delta)$ for the labels on its leaves, we define, as a notational convenience, a family of relations $\twoheadrightarrow_\Delta$ indexed by atomic causality graphs Δ, which only perform a $\twoheadrightarrow$ reduction when the redex matches Δ. The definition is also given in Figure 4. We then use this to define the family of relations $\to_G$ indexed by general causality graphs given at the bottom of Figure 4. If $C \to_G C'$ we say that *C steps to C' via G*.

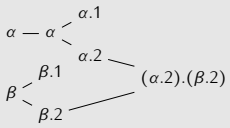
The last rule of Figure 4 extends an existing computation, which we suppose to have the form $C \to_G D \uplus D'$. Here D is a redex non-deterministically chosen from the current configuration, and $\uplus$ is an auxiliary operation which forms the configuration $D \cup D'$ as long as the domains of D and D' are disjoint. The reduction of D yields a reduct D'', which is merged back into what remains of the configuration, and Δ, which is appended to the existing causality graph G. The resulting composite graph is an output of the judgement.

To append Δ to G, we use a monoidal *composition* operator $\fatsemi$ for causality graphs, reminiscent of strategy composition in game semantics, with unit $\bullet$, the empty graph. Composition is parallel, except for matching inputs and outputs, which are connected sequentially.

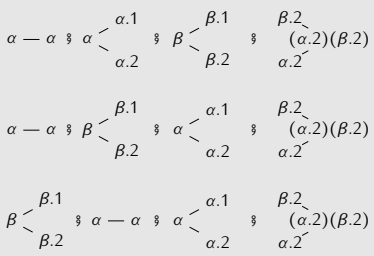
Definition 1 ($G \fatsemi G'$). *If G and G' are causality graphs, define $G \fatsemi G'$ to be the smallest graph G'' satisfying*

1. *G and G' are (isomorphic to) subgraphs of G'';*
2. *every $\alpha \in \mathit{in}(G) \cap \mathit{out}(G')$ labels exactly one node in G''.*

Example 1. *The causality graph*



can be decomposed in the following three ways:



The essential feature of causality graphs is that there are multiple interleavings consistent with the same G, as shown above. This gives us a way to disregard observationally irrelevant nondeterminism and instead deal with sequences of reductions which are equal modulo permutations that preserve causal structure. It is easy to see that the transition system induced by all the possible interleavings (for a fixed G) on an initial state $\Gamma \vdash C$ is confluent, and therefore has a unique terminal state. (Γ should contain only atomic process ids ℓ, so that there is no risk of the ids that arise during execution inadvertently colliding.)

Lemma 1 (Determinism of $\to_G$).
If $C \to_G C'$ and $C \to_G C''$ then $C' = C''$.

This ability to determinise, but not overly determinise, a concurrent program is essential to the technical development that follows. Other sources of observable non-determinism, such as explicit choice operators, can be dealt with by annotating the causality graph with additional information; for simplicity we omit such features from our language.

3 Problem definition

As mentioned in the introduction, Perera et al. [12, 13] formalise a notion of "explanation" for traces of values, based on a novel order-theoretic characterisation of dynamic program slicing. Their notion of explanation is tied to the ability to compute *backwards* from a value to a program that produced that value. Specifically, a trace *explains* a computed value v whenever it can be used to "unevaluate" v back to a partial program, or *program slice*, which is sufficient to compute v using an enriched semantics that allows evaluation of partial programs. Traces thus "explain" when they enable a round trip – from values to programs and back to values – which preserves the output. Unevaluation need not recover *all* of the original program e, but only enough of it to recompute v. Indeed, to underwrite a useful notion of explana-

tion, unevaluation should recover only those parts of e which are *necessary* for the computation of v.

In a concurrent setting, we are not dealing with deterministic computations that run to completion. Rather, the computations of interest are classes of causally equivalent possible evolutions $C \to_G C'$ of a non-deterministic system, and the "explananda" – the objects in need of explanation – are the terminal configurations C'. (We will sometimes refer to C as the "input" configuration and C' as the "output" configuration of the transition.) In §4 we will formalise how a configuration trace "explains" $C \to_G C'$ if it supports a round trip which preserves the terminal configuration, i.e. can be used to compute backwards from the output C' to recover some part, or *slice*, of the input C which is sufficient to compute forwards again to C', utilising any sequence of reductions consistent with G.

In this section we establish the preliminaries for §4. We will see that for such a transition there is always a unique minimal slice of C which suffices to compute C'. In fact there is always a least slice of C which suffices to compute any particular *slice* of C', a generalisation which is needed for compositionality. We shall give these least slices a purely extensional characterisation, considering first a single reduction under the $\twoheadrightarrow_\Delta$ rules, and then extending the analysis to full computations under the $\to_G$ semantics. The basic idea will be to extend the semantics of our concurrent language with an undefined element $\bot$, which is a stub for syntax that has been deleted during slicing, and rules that conservatively propagate $\bot$. Doing so exposes *parallel structure* in the computation, allowing us to reason about the relative independence of parts of the program by replacing parts of it with $\bot$ and observing how $\bot$ propagates. The parallel structure is made manifest by the existence of extremal forward and backward slices.

3.1 Partial configurations

We start by introducing the order structure of partial configurations, or configuration slices, already alluded to, and extending our operational semantics to accommodate partial configurations.

Meet-semilattice of processes. We extend processes with an empty, or *undefined*, process $\bot$, and similarly for expressions and values. From now on we mention only processes in the definitions; analogous notions arise for expressions and values.

$$\begin{array}{lrcl} \text{Expression} & e & ::= & \ldots \mid \bot \\ \text{Value} & u, v & ::= & \ldots \mid \bot \\ \text{Process} & P & ::= & \ldots \mid \bot \end{array}$$

Define $\sqsubseteq$ to be the partial order which has $P \sqsubseteq P'$ whenever we can obtain P from P' by replacing some parts by $\bot$. We say that P is a *prefix*, or *slice*, of P'. Processes extended with $\bot$ form a *meet-semilattice* with $\bot$ as bottom element; the meet $P \sqcap P'$ is the greatest common prefix of P and P'.

Lattice of prefixes. Moreover, the join $P \sqcup P'$ exists whenever P and P' are "compatible", i.e. prefixes of a common process, which we write as $P \uparrow P'$. The existence of such joins means that the down-set $\mathsf{Prefix}(P)$ of prefixes of P forms a finite distributive lattice (hereafter, simply *lattice*) with bottom element $\bot$ and top element P.

Pointwise order on functions. Functions between partial orders are ordered pointwise. In particular given configurations $\Gamma \vdash C$ and $\Gamma \vdash C'$, then $C \sqsubseteq C'$ when $C(\alpha) \sqsubseteq C'(\alpha)$ for every $\alpha \in \Gamma$. The down-set $\mathsf{Prefix}(C)$ also forms a lattice with meet and join given pointwise and the configuration $\bot_\Gamma \triangleq \{\alpha \mapsto \bot \mid \alpha \in \Gamma\}$ as bottom element.

Example 2. *Configurations C, C' are compatible, with the join as given.*

$$\begin{aligned} C &\triangleq [\![\bot\langle(3,\bot)\rangle]\!]^\alpha, [\![c(x).\bot \parallel P_2]\!]^\beta \\ C' &\triangleq [\![c\langle(\bot,4)\rangle]\!]^\alpha, [\![\bot(x).P_1 \parallel \bot]\!]^\beta \\ C \sqcup C' &= [\![c\langle(3,4)\rangle]\!]^\alpha, [\![c(x).P_1 \parallel P_2]\!]^\beta \end{aligned}$$

Semantics of partial configurations. We extend the $\twoheadrightarrow$ relation with the following $\bot$-*propagation* behaviour. (For now on by $\twoheadrightarrow$ we shall mean the extended definition.)

	with side-conditions:
BOT $\bot_\Gamma \twoheadrightarrow \bot_{\Gamma'}$	
JOIN-BOT-CHAN $[\![c\langle v\rangle]\!]^\alpha, [\![c'(x).P]\!]^\beta \twoheadrightarrow [\![\bot]\!]^{\alpha.\beta}$	$c = \bot$ or $c' = \bot$
JOIN-BOT-SEND $[\![\bot]\!]^\alpha, [\![c(x).P]\!]^\beta \twoheadrightarrow [\![\bot]\!]^{\alpha.\beta}$	
JOIN-BOT-RCV $[\![c\langle v\rangle]\!]^\alpha, [\![\bot]\!]^\beta \twoheadrightarrow [\![\bot]\!]^{\alpha.\beta}$	
JOIN $[\![c\langle v\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta \twoheadrightarrow [\![P\{v/x\}]\!]^{\alpha.\beta}$	$c \neq \bot$

Figure 6. Process evaluation ($\bot$-propagation rules)

The BOT rule sends least configurations to least configurations. The JOIN-BOT-CHAN rule permits any sender or receiver to synchronise with a process communicating on the $\bot$ channel, and JOIN-BOT-SEND and JOIN-BOT-RCV permit a synchronisation when either sender or receiver is undefined. These rules help ensure that prefixes of configurations which are not stuck are not themselves stuck (Lemma 2 below), but have the effect of rendering $\twoheadrightarrow$ non-deterministic, so the $\twoheadrightarrow_\Delta$ variant becomes important here as a way of fixing Δ. The JOIN rule is as before except for a side-condition that requires $c \neq \bot$, which is explained in Example 3 below.

Existence of required minima. Our goal is to show that for a given transition $C \to_G C'$ and a given part of the output C', there is a least slice of the input C which, utilising any chain of reductions consistent with G, is large enough to compute that part of C'. We will show that our generalisation above of the $\to_G$ semantics to *prefixes* of C gives rise to input minima of the desired kind. More precisely, it gives rise to a monotonic function $\mathsf{step}_{C,G}$ from $\mathsf{Prefix}(C)$ to $\mathsf{Prefix}(C')$ such that for any $D' \sqsubseteq C'$, the set $S_{D'} \triangleq \{D \mid D \sqsubseteq C \text{ and } \mathsf{step}_{C,G}(D) \sqsupseteq D'\}$ has a least element. Such a function is *stable* in the sense of Berry [1]. If such minima always exist, then clearly there is a process "unevaluation" function, which we call $\mathsf{unstep}_{C,G}$, from $\mathsf{Prefix}(C')$ to $\mathsf{Prefix}(C)$ which takes any $D' \sqsubseteq C'$ to the least element of $S_{D'}$.

In our setting, where the domains are finite lattices, the existence of such minima is equivalent to $\mathsf{step}_{C,G}$ preserving meets, since then $S_{D'}$ is closed under meets, with least element $\bigsqcap S_{D'}$. The function $\mathsf{unstep}_{C,G}$ preserves joins, and together with $\mathsf{step}_{C,G}$ forms a *Galois connection*. Therefore our general strategy will be to generalise each component of evaluation from §2 to prefix lattices, showing that each preserves meets. We first derive a family of meet-preserving functions $\mathsf{reduce}_{C,\Delta}$ from $\mathsf{Prefix}(C)$ to $\mathsf{Prefix}(C')$ for every reduction $C \twoheadrightarrow_\Delta C'$. We then do a similar thing for the $\to_G$ relation, and show how we obtain meet-preserving functions which feature the $\mathsf{reduce}_{C,\Delta}$ functions as components. The central role of Galois connections will become apparent as we go along.

3.2 Least process slices

Galois connection for an individual reduction. First consider the $C \twoheadrightarrow_\Delta C'$ judgement, which is deterministic, but not total. This relation induces a family of meet-preserving functions indexed by its domain.

Definition 2 ($\mathsf{reduce}_{C,\Delta}$ function)**.** *Suppose $C \twoheadrightarrow_\Delta C'$. Then define the function $\mathsf{reduce}_{C,\Delta}$ from $\mathsf{Prefix}(C)$ to $\mathsf{Prefix}(C')$ to be $\twoheadrightarrow_\Delta$ domain-restricted to $\mathsf{Prefix}(C)$.*

Lemma 2. *If $C \twoheadrightarrow_\Delta C'$ then $\mathsf{reduce}_{C,\Delta}$ is total and preserves $\sqcap$.*

Proof. (1) is straightforward case analysis, using the $\bot$-propagation rules when the redex would otherwise get stuck. For (2), see Appendix, §C.1. □

This property depends on expression evaluation and substitution also giving rise to families of meet-preserving functions on prefix lattices, which we return to in §3.3 below. The following example shows how not having the $c \neq \bot$ side-condition on the JOIN rule would violate meet-preservation.

Example 3. *Suppose*

$$C \triangleq [\![c\langle(3,4)\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta \twoheadrightarrow_\Delta [\![P\{(3,4)/x\}]\!]^{\alpha.\beta}$$

for some $c \neq \bot$. Then $D, D' \sqsubseteq C$ below both reduce to $[\![\bot]\!]^{\alpha.\beta}$, but without a $c \neq \bot$ side-condition on the JOIN rule, their meet would reduce to something larger, making the transition highlighted in red:

$$D \triangleq [\![\bot\langle(3,\bot)\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta \twoheadrightarrow_\Delta [\![\bot]\!]^{\alpha.\beta}$$
$$D' \triangleq [\![c\langle(\bot,4)\rangle]\!]^\alpha, [\![\bot(x).P]\!]^\beta \twoheadrightarrow_\Delta [\![\bot]\!]^{\alpha.\beta}$$
$$D \sqcap D' = [\![\bot\langle(\bot,\bot)\rangle]\!]^\alpha, [\![\bot(x).P]\!]^\beta \twoheadrightarrow_\Delta [\![P\{(\bot,\bot)/x\}]\!]^{\alpha.\beta}$$

Instead the JOIN-BOT-CHAN rule should apply.

Meet-preservation guarantees the existence of the minima we want, so we can define the following family of "reverse reduction" functions.

Definition 3. *Suppose $C \twoheadrightarrow_\Delta C'$. Then define the following function from $\mathsf{Prefix}(C')$ to $\mathsf{Prefix}(C)$.*

$$\mathit{unreduce}_{C,\Delta}(D') \triangleq \bigsqcap\{D \mid D \sqsubseteq C \text{ and } \mathit{reduce}_{C,\Delta}(D) \sqsupseteq D'\}$$

This function preserves joins; the pair $(\mathsf{reduce}_{C,\Delta}, \mathsf{unreduce}_{C,\Delta})$ form a *Galois connection*, with $\mathsf{reduce}_{C,\Delta}$ and $\mathsf{unreduce}_{C,\Delta}$ its so-called *upper* and *lower* adjoints.

The pointwise order on functions will be useful for concisely expressing the relationship between $\mathsf{reduce}_{C,\Delta}$ and $\mathsf{unreduce}_{C,\Delta}$. In particular, writing id_e for the identity function on $\mathsf{Prefix}(e)$, we can state that an endo-function $f : \mathsf{Prefix}(e) \to \mathsf{Prefix}(e)$ is *inflationary* (has $f(e') \sqsupseteq e'$ for any $e' \sqsubseteq e$) by simply stating that $f \sqsupseteq \mathsf{id}_e$. Analogously f is *deflationary* iff $f \sqsubseteq \mathsf{id}_e$.

Lemma 3 ($(\mathsf{reduce}_{C,\Delta}, \mathsf{unreduce}_{C,\Delta})$ form a Galois connection)**.**
1. $\mathit{unreduce}_{C,\Delta} \circ \mathit{reduce}_{C,\Delta} \sqsubseteq \mathit{id}_C$
2. $\mathit{reduce}_{C,\Delta} \circ \mathit{unreduce}_{C,\Delta} \sqsupseteq \mathit{id}_{C'}$

We can read Lemma 3 as follows: $\mathsf{unreduce}_{C,\Delta}$ goes back from parts of the reduct to parts of the redex in a necessary and sufficient way, identifying for any particular part of the reduct just that part of the redex required to reconstruct it using $\mathsf{reduce}_{C,\Delta}$.

More precisely, for any $D' \sqsubseteq C'$, let $D \triangleq \mathsf{unreduce}_{C,\Delta}(D')$. Then property (1) of the lemma implies that D is "necessary", i.e. smaller than any slice of C large enough to reduce to D', and property (2) implies that D is "sufficient", i.e. large enough itself to reduce to D'. There is a dual reading of Lemma 3 for $\mathsf{reduce}_{C,\Delta}$, which states that it computes the largest slice of the reduct that it can given only some prefix of the redex.

Example 4. *The following shows a reduction $C \twoheadrightarrow_\Delta C'$ using the JOIN rule. Here the receiving process is a fork which will attempt to send the received value on two further channels. Recall that v_x means that v was previously substituted for x. (We omit variable annotations on the channels.)*

$$[\![c_1\langle(3,4)\rangle]\!]^\alpha, [\![c_1(x).c_2\langle x\rangle \parallel c_3\langle x\rangle]\!]^\beta \twoheadrightarrow_\Delta [\![c_2\langle(3,4)_x\rangle \parallel c_3\langle(3,4)_x\rangle]\!]^{\alpha.\beta}$$

Consider the following two slices of the reduct C':

$$D \triangleq [\![\bot \parallel c_3\langle(3,\bot)_x\rangle]\!]^{\alpha.\beta} \qquad D' \triangleq [\![c_2\langle(\bot,4)_x\rangle \parallel c_3\langle(3,\bot)_x\rangle]\!]^{\alpha.\beta}$$

and the corresponding slices of the redex C obtained by $\mathit{unreduce}_{C,\Delta}$:

$$\mathit{unreduce}_{C,\Delta}(D) = [\![c_1\langle(3,\bot)\rangle]\!]^\alpha, [\![c_1(x).\bot \parallel c_3\langle x\rangle]\!]^\beta$$
$$\mathit{unreduce}_{C,\Delta}(D') = [\![c_1\langle(3,4)\rangle]\!]^\alpha, [\![c_1(x).c_2\langle x\rangle \parallel c_3\langle x\rangle]\!]^\beta$$

The second case illustrates the interaction with substitution: the full pair $(3,4)$ is retained in the sender, since we need to approximate both values of x in D', namely $(\bot,4)$ and $(3,\bot)$. We will see in §4 how the $\mathit{unreduce}_{C,\Delta}$ function can be implemented.

Galois connection for a concurrent computation. Now we define a similar Galois connection for a $\to_G$ step, and observe that Galois connections of the form $(\mathsf{reduce}_{C,\Delta}, \mathsf{unreduce}_{C,\Delta})$ appear as a component.

Definition 4. *Suppose $C \to_G C'$. Then define the Galois connection $(\mathit{step}_{C,G}, \mathit{unstep}_{C,G})$ between $\mathsf{Prefix}(C)$ and $\mathsf{Prefix}(C')$ where $\mathit{step}_{C,G}$ is $\to_G$ domain-restricted to $\mathsf{Prefix}(C)$.*

To see that $\mathsf{step}_{C,G}$ is indeed the upper adjoint of a Galois connection (i.e., that $\mathsf{step}_{C,G}$ is meet preserving), we need only unpack the definition of $\to_G$ and observe that $\mathsf{step}_{C,G}$ satisfies

$$\mathsf{step}_{C,\bullet} = \mathsf{id}_C \tag{1}$$
$$\mathsf{step}_{C,G' \fatsemi \Delta} = (\mathsf{reduce}_{D,\Delta} \uplus \mathsf{id}_{D'}) \circ \mathsf{step}_{C',G'} \tag{2}$$

where in Equation 2, we have $C \to_{G'} D \uplus D'$ with $D \twoheadrightarrow_\Delta D''$. Here $\uplus$ lifts pointwise to functions. Meet-preservation of $\mathsf{step}_{C,G}$ is then a straightforward induction on the size of G using Lemma 2. Then it is equally easy to see that the lower adjoint $\mathsf{unstep}_{C,G}$ satisfies

$$\mathsf{unstep}_{C,\bullet} = \mathsf{id}_C \tag{3}$$
$$\mathsf{unstep}_{C,G' \fatsemi \Delta} = \mathsf{unstep}_{C',G'} \circ (\mathsf{unreduce}_{D,\Delta} \uplus \mathsf{id}_{D'}) \tag{4}$$

In these equations, the decomposition of G into $G' \fatsemi \Delta$ is not unique, and thus there are multiple ways of decomposing $(\mathsf{step}_{C,G}, \mathsf{unstep}_{C,G})$. However they all denote the same Galois connection by Lemma 1, with $\uplus$ acting at the level of functions as parallel composition, permitting some of the sequential compositions to commute.

Definition 4 canonically captures the relationship between forward and backward slices for a particular class of causally equivalent concurrent executions. There is an asymmetry, however: whereas the definition of the forward-slicing function $\mathsf{step}_{C,G}$ is readily interpreted as a non-deterministic but confluent algorithm (expressed as it is in terms of $\to_G$ and $\twoheadrightarrow_\Delta$), its backward-slicing counterpart $\mathsf{unstep}_{C,G}$ lacks an obvious operational interpretation. Although we know that it takes any output slice $D' \sqsubseteq C'$ to the meet of all the input slices $D \sqsubseteq C$ large enough to compute D' using $\mathsf{step}_{C,G}$, there is no obvious efficient procedure for calculating the required meet. In section §4, we show that such a procedure naturally arises as a form of reverse computation. We give a reverse operational semantics for processes, called *process unevaluation*, which utilises a trace of the concurrent computation, and prove that it agrees extensionally with $\mathsf{unstep}_{C,G}$. First, we attend to some other components of our system that the Galois connections just introduced depend on.

3.3 Modular components for reversible concurrency

The development in the previous section relies on the fact that if $C \twoheadrightarrow_\Delta C'$ then $\mathrm{reduce}_{C,\Delta}$ is total and meet preserving (Lemma 2). Since processes contain nested expressions and reduction involves substitution, the proof of this fact relies on similar meet preservation results for expressions and substitutions, which we describe here briefly. The existence of unevaluation functions that compute least slices for expressions and substitutions are corollaries.

Galois connection for an expression evaluation. For expression evaluations, we rely on the approach of [13], which we briefly recapitulate here. For every terminating computation $e \Rightarrow v$, we derive a meet-preserving function eval_e from $\mathsf{Prefix}(e)$ to $\mathsf{Prefix}(v)$ by extending the big-step evaluation relation $e \Rightarrow v$ with the additional $\bot$*-propagation* rules given in Figure 7.

$$\frac{}{\bot \Rightarrow \bot} \qquad \frac{e_1 \Rightarrow \bot}{e_1\ e_2 \Rightarrow \bot} \qquad \frac{e \Rightarrow \bot}{\mathsf{fst}\ e \Rightarrow \bot}$$

$$\frac{e \Rightarrow \bot}{\mathsf{case}\ e\ \{\mathsf{inl}\ x_1.e_1; \mathsf{inr}\ x_2.e_2\} \Rightarrow \bot}$$

Figure 7. Expression evaluation ($\bot$-propagation rules)

The undefined expression $\bot$ evaluates to the undefined value $\bot$; eliminating the value $\bot$ also results in $\bot$. Note that $\bot$ cannot be treated as synonymous with a divergent computation, at least not under a standard sequential call-by-value semantics, because it does not stop parts of the computation which do not depend on it from proceeding. For example if e evaluates to $\bot$ then $\mathsf{inl}\ e$ evaluates to $\mathsf{inl}\ \bot$, capturing the fact that the inner and outer computations are independent. Similarly a case analysis where the scrutinee evaluates to $\mathsf{inl}\ \bot$ does not get stuck, but takes the left branch with x_1 bound to $\bot$. Only under a more parallel operational semantics can $\bot$ as treated here be unified with divergence; we revisit this at the end of the paper.

Definition 5 (eval_e)**.** *Suppose $e \Rightarrow v$. Define eval_e to be $\Rightarrow$ domain-restricted to $\mathsf{Prefix}(e)$.*

Lemma 4. *If $e \Rightarrow v$ then eval_e is total and preserves $\sqcap$.*

Proof. Similar to the one given in [13], using the $\bot$-propagation rules and Lemma 5 below for the substitution cases. □

The existence of a unique function uneval_e, adjoint to eval_e, is immediate.

Corollary 1 (Existence of least expression slices)**.** *If $e \Rightarrow v$ there exists a unique function uneval_e from $\mathsf{Prefix}(v)$ to $\mathsf{Prefix}(e)$ such that $(\mathrm{eval}_e, \mathrm{uneval}_e)$ is a Galois connection.*

Galois connection for a substitution. Substitution easily fits into this approach, since it is total and preserves all meets, not just compatible meets. (Substitution also preserves compatible joins, but here we are only concerned with meet-preservation.)

Lemma 5. $(e_1 \sqcap e_2)\{v_1 \sqcap v_2/x\} = e_1\{v_1/x\} \sqcap e_2\{v_2/x\}$

Then once again, the existence of a family of Galois connections indexed by the domain of substitution is immediate. Each Galois connection is degenerate, in that the lower adjoint is a retraction: $\mathrm{unsubst}_{e,v,x} \circ \mathrm{subst}_{e,v,x} = \mathrm{id}_{(e,v)}$.

Corollary 2. *Suppose $e\{v/x\} = e'$, and write $\mathrm{subst}_{e,v,x}$ for $-\{-/x\}$ domain-restricted to $\mathsf{Prefix}(e, v)$. Then there is a unique function $\mathrm{unsubst}_{e,v,x}$ from $\mathsf{Prefix}(e')$ to $\mathsf{Prefix}(e, v)$ such that $(\mathrm{subst}_{e,v,x}, \mathrm{unsubst}_{e,v,x})$ is a Galois connection.*

Galois connection for a primitive operation. Primitive operations and external modules written in different languages can be safely added to our system, so long as those operations give rise, in the way we have just seen, to families of stable functions when restricted to prefixes of their arguments. The provider of the operation must supply not only an implementation of the operation, but also of the implied lower adjoints. We briefly explain now how the stability requirement constrains the amount of parallel structure that the operation can exhibit with respect to its arguments.

Consider extending our system with a commutative primitive operation with an annihilator, such as $\times$ on natural numbers. One might think it reasonable for $\times$ to satisfy both $0 \times \bot = 0$ and $\bot \times 0 = 0$, but also $\bot \times \bot = \bot$. However such an operation is unstable: there are *two* minimal prefixes of 0×0 which are large enough to compute 0. Another example is Plotkin's *parallel or* [15], which is similarly non-strict in both arguments and therefore unstable. (It was this observation which motivated Berry's work on stable functions.)

In general to be stable an operation must be *sequential*, i.e. consume its arguments in a deterministic order. For an internal operation implemented purely in our language, this sequentiality arises automatically from the operational semantics, but an external operation – because it essentially provides its own semantics – must take care to implement the required sequentiality itself.

4 Implementing reversible concurrency

We now equip our language with a reverse operational semantics called *process unevaluation*. Process unevaluation implements $\mathrm{unstep}_{C,G}$, the unique lower adjoint of the $\bot$-propagating $\mathrm{step}_{C,G}$ forward-slicing function defined in §3, making use of a *trace* recording the history of the computation. Unevaluation is a form of concurrent backward slicing, but will form the basis of our notion of correctness for traces.

The trace of a concurrent computation is called a *configuration trace* (§4.1). Configuration traces are built by tracing versions of the $\twoheadrightarrow$ and $\longrightarrow$ relations, which we write as $\square\!\!\twoheadrightarrow$ and $\square\!\!\longrightarrow$ respectively (§4.2). Utilising the trace, we are able to define deterministic reversing versions of these relations, written $\twoheadleftarrow\!\!\square$ and $\longleftarrow\!\!\square$ (§4.3), which perform backward $\bot$-propagation. Each relation gives rise to a family of total functions on the principal down-sets of its domain, and we show how these assemble algorithmically into a concurrent but deterministic implementation of $\mathrm{unstep}_{C,G}$ for any $C \longrightarrow_G C'$. We omit the details of reverse execution for expressions, as specified by the lower adjoint uneval_e, and instead refer the reader to [13].

4.1 Configuration traces

A configuration trace is simply a causal graph annotated with extra information to allow reverse execution. Thus by construction, all executions of a given program which have the same synchronisation structure have the same trace.

Configuration traces come in two forms, reflecting the structure of causality graphs. (Atomic) *redex traces* R, defined in Figure 8, record the reduction of an individual redex, in particular any process constructor that was eliminated, and any expression that might have been evaluated via a side-condition on the reduction rule. A redex trace has as "inputs" its root process ids, and as "outputs" any holes of the form $\blacksquare^\alpha$. For example the redex trace $[c\langle v\rangle]^\alpha, [c(x).\blacksquare^{\alpha.\beta}]^\beta$ has two inputs α and β and a single output

$\alpha.\beta$. We write $R : \Delta$ to mean that R has the input-output shape specified by Δ.

$$\text{Redex trace} \quad R ::= [\bot.\blacksquare^{\alpha}]^{\alpha} \mid [\mathbf{0}]^{\alpha} \mid [\nu x.\blacksquare^{\alpha}]^{\alpha} \mid [\blacksquare^{\alpha.1} \parallel \blacksquare^{\alpha.2}]^{\alpha} \mid [c\langle v\rangle]^{\alpha}, [c(x).\blacksquare^{\alpha.\beta}]^{\beta} \mid [\bot.\blacksquare^{\alpha.\beta}]^{\beta} \mid [e.\blacksquare^{\alpha}]^{\alpha} \mid [\text{run } e.[\bot]^{\alpha}]^{\alpha}$$

Figure 8. Redex traces (atomic)

Least redex traces record $\bot$-propagation. The redex traces $[\mathbf{0}]^{\alpha}$ and $[\blacksquare^{\alpha.1} \parallel \blacksquare^{\alpha.2}]^{\alpha}$ are the only redex traces of their shape, and are therefore already their own least prefix; for the remaining forms, we introduce special syntax for the least redex trace of that shape. For JOIN reductions, $[\bot]^{\alpha}, [\bot.\blacksquare^{\alpha.\beta}]^{\beta}$ is the least trace form; for all other reductions, $[\bot.\blacksquare^{\alpha}]^{\alpha}$ is the least trace form. To allow a single BOT rule to subsume multiple cases, we write $\bot_{\Delta}$ to denote the least redex trace of shape Δ, with the exception of $[\mathbf{0}]^{\alpha}$, which has a slightly different semantics.

A (general) *configuration trace* T is then any finite graph which has been built by composing redex traces. The composition operator $\fatsemi$ for traces is analogous to the one for causal graphs. We write $T : G$ to mean that T has causal graph G. For any $T : G$ and any $R : \Delta$, the notation $T \fatsemi R$ means the configuration trace with causal graph $G \fatsemi \Delta$ that results from composing T and R are in parallel, except for inputs of R which match outputs of T, which are composed sequentially. The $\blacksquare$ in each output of T is replaced by the information on the corresponding input of R. Example 6 below illustrates.

4.2 Traced evaluation

The trace syntax is best understood by seeing how traces are assembled by the tracing semantics. Lemma 6 relates the tracing semantics to the reference semantics.

Tracing reductions. A $\twoheadrightarrow$ reduction is traced via the traced reduction relation $\boxdot\!\!\twoheadrightarrow$ defined in Figure 9. (The $\boxdot$ at the beginning of the arrow indicates that it is the "tracing" counterpart of $\twoheadrightarrow$.) The judgement $C \boxdot\!\!\twoheadrightarrow R[C']$ states that $C \twoheadrightarrow_{\Delta} C'$, with the redex trace $R : \Delta$ recording the reduction. The BOT rule is the tracing analogue of the BOT rule for $\twoheadrightarrow$; it takes a least input configuration to a least output configuration, using the least trace with the appropriate input-output shape to record the reduction. Again, this rule means traced reduction is non-deterministic (BOT applies for any Δ whose inputs match Γ) and so we sometimes make use of a variant $\boxdot\!\!\twoheadrightarrow_{\Delta}$ which has been determinised by fixing Δ in advance.

For any $T : G$ and any $\text{out}(G) \vdash C$, the notation $T[C]$ denotes any T and C such that the outputs of T are exactly the inputs of C. Such a pair is called a *traced configuration* (or *traced reduct*, when T happens to be a redex trace T). We use a notation for $T[C]$ that plugs the outputs of T with the corresponding inputs of C. The following illustrates.

Example 5.

	"Plugged" notation...	*...for traced configuration $T[C]$*
(a)	$[\bot.[\![P_1 \parallel P_2]\!]^{\alpha}]^{\alpha}$	$([\bot]^{\alpha})[[\![P_1 \parallel P_2]\!]^{\alpha}]$
(b)	$[[\![P_1]\!]^{\alpha.1} \parallel [\![P_2]\!]^{\alpha.2}]^{\alpha}$	$([[\bot]^{\alpha.1} \parallel [\bot]^{\alpha.2}]^{\alpha})[[\![P_1]\!]^{\alpha.1}, [\![P_2]\!]^{\alpha.2}]$
(c)	$[[\mathbf{0}]^{\alpha.1} \parallel [\mathbf{0}]^{\alpha.2}]^{\alpha}$	$([[\mathbf{0}]^{\alpha.1} \parallel [\mathbf{0}]^{\alpha.2}]^{\alpha})[\varnothing]$

In (a) and (b) the trace component T is the least trace of its shape. In (c) the trace has no holes and so the configuration component C is empty.

Tracing concurrent computations. To lift traced reduction to entire concurrent computations, we now define a tracing analogue of the stepping relation $\longrightarrow$, written $\boxdot\!\!\longrightarrow$, given at the bottom of Figure 9. The judgement $C \boxdot\!\!\longrightarrow T[C']$ states that $C \longrightarrow_G C'$ with the unique trace $T : G$ recording the evolution of C to C'. The first rule lifts any configuration to a traced configuration where the trace is empty. The second rule defines how to extend an existing traced computation with a new reduction. First, a redex D is non-deterministically selected from the output configuration of the existing computation. The redex is reduced, obtaining a traced reduct $R[D'']$. The redex trace R is appended to the existing trace, and D'' merged back into what remains of the configuration.

$\boxed{C \boxdot\!\!\twoheadrightarrow R[C']}$

BOT — where: $\Gamma \vdash \Delta : \Gamma'$

$$\bot_{\Gamma} \boxdot\!\!\twoheadrightarrow \bot_{\Delta}[\bot_{\Gamma'}]$$

STOP

$$[\![\mathbf{0}]\!]^{\alpha} \boxdot\!\!\twoheadrightarrow [\mathbf{0}]^{\alpha}$$

NEW

$$[\![\nu x.P]\!]^{\alpha} \boxdot\!\!\twoheadrightarrow [\nu x.[\![P\{c/x\}]\!]^{\alpha}]^{\alpha}$$

FORK

$$[\![P_1 \parallel P_2]\!]^{\alpha} \boxdot\!\!\twoheadrightarrow [[\![P_1]\!]^{\alpha.1} \parallel [\![P_2]\!]^{\alpha.2}]^{\alpha}$$

SEND-ON

$$[\![e_1\langle e_2\rangle]\!]^{\alpha} \boxdot\!\!\twoheadrightarrow [e_1.[\![c\langle e_2\rangle]\!]^{\alpha}]^{\alpha} \qquad e_1 \Rightarrow c$$

SEND-READY

$$[\![c\langle e\rangle]\!]^{\alpha} \boxdot\!\!\twoheadrightarrow [e.[\![c\langle v\rangle]\!]^{\alpha}]^{\alpha} \qquad e \Rightarrow v$$

RCV-READY

$$[\![e(x).P]\!]^{\alpha} \boxdot\!\!\twoheadrightarrow [e.[\![c(x).P]\!]^{\alpha}]^{\alpha} \qquad e \Rightarrow c$$

JOIN

$$[\![c\langle v\rangle]\!]^{\alpha}, [\![c(x).P]\!]^{\beta} \boxdot\!\!\twoheadrightarrow [c\langle v\rangle]^{\alpha}, [c(x).[\![P\{v/x\}]\!]^{\alpha.\beta}]^{\beta} \qquad c \neq \bot$$

JOIN-BOT-CHAN

$$[\![c\langle v\rangle]\!]^{\alpha}, [\![c'(x).P]\!]^{\beta} \boxdot\!\!\twoheadrightarrow [\bot]^{\alpha}, [\bot.[\![\bot]\!]^{\alpha.\beta}]^{\beta} \qquad c = \bot \text{ or } c' = \bot$$

JOIN-BOT-SEND

$$[\![\bot]\!]^{\alpha}, [\![c(x).P]\!]^{\beta} \boxdot\!\!\twoheadrightarrow [\bot]^{\alpha}, [\bot.[\![\bot]\!]^{\alpha.\beta}]^{\beta}$$

JOIN-BOT-RCV

$$[\![c\langle v\rangle]\!]^{\alpha}, [\![\bot]\!]^{\beta} \boxdot\!\!\twoheadrightarrow [\bot]^{\alpha}, [\bot.[\![\bot]\!]^{\alpha.\beta}]^{\beta}$$

RUN

$$[\![\text{run } e]\!]^{\alpha} \boxdot\!\!\twoheadrightarrow [\text{run } e.[\![P]\!]^{\alpha}]^{\alpha} \qquad e \Rightarrow \{P\}$$

$\boxed{C \boxdot\!\!\longrightarrow T[C']}$

$$\frac{}{C \boxdot\!\!\longrightarrow [C]} \qquad \frac{C \boxdot\!\!\longrightarrow T[D \uplus D'] \qquad D \boxdot\!\!\twoheadrightarrow R[D'']}{C \boxdot\!\!\longrightarrow (T \fatsemi R)[D'' \uplus D]}$$

Figure 9. Traced evaluation of processes

Example 6. *Suppose $e \Rightarrow \{P'\}$ and consider how the following derivation extends an existing computation.*

$$\frac{C \boxdot\!\!\longrightarrow [[\![\text{run } e]\!]^{\alpha.1} \parallel [\![P]\!]^{\alpha.2}]^{\alpha} \qquad [\![\text{run } e]\!]^{\alpha.1} \boxdot\!\!\twoheadrightarrow [\text{run } e.[\![P']\!]^{\alpha.1}]^{\alpha.1}}{C \boxdot\!\!\longrightarrow [[\text{run } e.[\![P']\!]^{\alpha.1}]^{\alpha.1} \parallel [\![P]\!]^{\alpha.2}]^{\alpha}}$$

Suppose the input configuration C has run to a state with processes $[\![\text{run } e]\!]^{\alpha.1}, [\![P]\!]^{\alpha.2}$ in its output configuration, with $T \triangleq [\blacksquare^{\alpha.1} \parallel \blacksquare^{\alpha.2}]^{\alpha}$ recording its history. The redex $[\![\text{run } e]\!]^{\alpha.1}$ is chosen non-deterministically. The output of the reduction is reduct $[\![P']\!]^{\alpha.1}$ and trace $R \triangleq [\text{run } e.\blacksquare^{\alpha.1}]^{\alpha.1}$.

$\boxed{R[C] \twoheadleftarrow\!\square\; C'}$

BOT
$R[\bot_{\Gamma'}] \twoheadleftarrow\!\square\; \bot_\Gamma$ $\qquad \Gamma \vdash R : \Gamma'$

STOP
$[\mathbf{0}]^\alpha \twoheadleftarrow\!\square\; [\![\mathbf{0}]\!]^\alpha$

NEW
$[\nu x.[\![\{c/x\}P]\!]^\alpha]^\alpha \twoheadleftarrow\!\square\; [\![\nu x.P]\!]^\alpha$ $\qquad P \neq \bot$

FORK
$[[\![P_1]\!]^{\alpha.1} \parallel [\![P_2]\!]^{\alpha.2}]^\alpha \twoheadleftarrow\!\square\; [\![P_1 \parallel P_2]\!]^\alpha$

SEND-ON
$[e_1.[\![c\langle e_2\rangle]\!]^\alpha]^\alpha \twoheadleftarrow\!\square\; [\![s_1\langle e_2\rangle]\!]^\alpha$ $\qquad c \Leftarrow_{e_1} s_1$

SEND-READY
$[e.[\![c\langle v\rangle]\!]^\alpha]^\alpha \twoheadleftarrow\!\square\; [\![c\langle s\rangle]\!]^\alpha$ $\qquad v \Leftarrow_e s$

RCV-READY
$[e.[\![c(x).P]\!]^\alpha]^\alpha \twoheadleftarrow\!\square\; [\![s(x).P]\!]^\alpha$ $\qquad c \Leftarrow_e s$

JOIN
$[c\langle v\rangle]^\alpha, [c(x).[\![\{u/x\}P]\!]^{\alpha.\beta}]^\beta \twoheadleftarrow\!\square\; [\![c\langle u\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta$ $\qquad P \neq \bot,\ c \neq \bot$ and $u \sqsubseteq v$

RUN
$[\mathrm{run}\ e.[\![P]\!]^\alpha]^\alpha \twoheadleftarrow\!\square\; [\![\mathrm{run}\ s]\!]^\alpha$ $\qquad P \neq \bot$ and $\{P\} \Leftarrow_e s$

$\boxed{T[C] \longleftarrow\!\square\; C'}$

$$\frac{}{[C] \longleftarrow\!\square\; C} \qquad \frac{R[D] \twoheadleftarrow\!\square\; D'' \qquad T[D'' \uplus D'] \longleftarrow\!\square\; C}{(T \fatsemi R)[D \uplus D'] \longleftarrow\!\square\; C}$$

Figure 10. Traced unevaluation of processes

Then R is appended to T yielding $T \fatsemi R = [[\mathrm{run}\ e.\blacksquare^{\alpha.1}]^{\alpha.1} \parallel \blacksquare^{\alpha.2}]^\alpha$, which becomes the trace of the updated configuration $[\![P']\!]^{\alpha.1}, [\![P]\!]^{\alpha.2}$.

Lemma 6 (Agreement with reference semantics).
1. $C \twoheadrightarrow_\Delta C' \iff$ *exists* $R : \Delta$ *with* $C \boxdot\!\twoheadrightarrow R[C']$.
2. $C \rightarrow_G C' \iff$ *exists* $T : G$ *with* $C \boxdot\!\rightarrow T[C']$.

Finally, we take some time to explain the $e.\blacksquare^\alpha$ trace form. (The run $e.\blacksquare^\alpha$, $\nu x.\blacksquare^\alpha$ and $\blacksquare^{\alpha.2} \parallel \blacksquare^{\alpha.2}$ trace forms should be reasonably clear.) The $e.\blacksquare^\alpha$ form only arises during sending and receiving, and records evaluated expressions. By the time a sender is in the form $[\![c\langle v\rangle]\!]^\alpha$, its trace will (via SEND-ON and then SEND-READY) be of the form $[e_1.[e_2.\blacksquare^\alpha]^\alpha]^\alpha$, where e_1 and e_2 record the computation of c and v respectively. Similarly, by the time a receiver is in the form $[\![c(x).P]\!]^\alpha$, its trace will (via RCV-READY) be of the form $[e.\blacksquare^\alpha]^\alpha$, where e records the computation of c. While it is true that the RUN rule also evaluates an expression, it must emit the run constructor into the trace as well as e, because it performs the constructor elimination and the expression evaluation in the same step. The next example illustrates sender and receiver traces.

Example 7. *We trace the execution of a configuration with a single process with id α. The process allocates a channel c, and then forks into two subprocesses. The first computes the factorial of 4 and sends the result on c; the second waits on c for the result and then stops.*

$[\![\nu x.x\langle fact\ 4\rangle \parallel x(y).\mathbf{0}]\!]^\alpha$
$[\nu x.[\![c_x\langle fact\ 4\rangle \parallel c_x(y).\mathbf{0}]\!]^\alpha]^\alpha$ $\qquad$ *(recall that $x\{c/x\} = c_x$)*

$[\nu x.[[\![c_x\langle fact\ 4\rangle]\!]^{\alpha.1} \parallel [\![c_x(y).\mathbf{0}]\!]^{\alpha.2}]^\alpha]^\alpha$
$[\nu x.[[c_x.[\![c\langle fact\ 4\rangle]\!]^{\alpha.1}]^{\alpha.1} \parallel [\![c_x(y).\mathbf{0}]\!]^{\alpha.2}]^\alpha]^\alpha$
$[\nu x.[[c_x.[fact\ 4.[\![c\langle 24\rangle]\!]^{\alpha.1}]^{\alpha.1}]^{\alpha.1} \parallel [\![c_x(y).\mathbf{0}]\!]^{\alpha.2}]^\alpha]^\alpha$
$[\nu x.[[c_x.[fact\ 4.[\![c\langle 24\rangle]\!]^{\alpha.1}]^{\alpha.1}]^{\alpha.1} \parallel [c_x.[\![c(y).\mathbf{0}]\!]^{\alpha.2}]^{\alpha.2}]^\alpha]^\alpha$
$[\nu x.[[c_x.[fact\ 4.[c\langle 24\rangle]^{\alpha.1}]^{\alpha.1}]^{\alpha.1} \parallel [c_x.[c(y).[\![\mathbf{0}]\!]^{\alpha.2}]^{\alpha.2}]^{\alpha.2}]^\alpha]^\alpha$
$[\nu x.[[c_x.[fact\ 4.[c\langle 24\rangle]^{\alpha.1}]^{\alpha.1}]^{\alpha.1} \parallel [c_x.[c(y).[\mathbf{0}]^{\alpha.2}]^{\alpha.2}]^{\alpha.2}]^\alpha]^\alpha$

The name "fact" here should not be read as a variable but rather as shorthand for a variable-annotated value of the form v_{fact}, where v is a recursive function defining factorial.

4.3 Traced unevaluation

We now give a reverse operational semantics for processes called *process unevaluation*, that relies on the trace, and show that it implements the function $\mathrm{unstep}_{C,G}$ for any transition $C \rightarrow_G C'$.

Reversing reductions. We start by defining a deterministic relation $\twoheadleftarrow\!\square$, given in Figure 10, that reverses a traced reduction. For any $R : \Delta$ the judgement $R[C] \twoheadleftarrow\!\square\; C'$ states that the reduct C can *unreduce* to the redex $\mathrm{in}(\Delta) \vdash C'$, consuming the redex trace R in so doing.

The BOT rule implements backward $\bot$-propagation. It says least output configurations map to least input configurations, discarding all the information in the redex trace. The exception is the empty output configuration, which is its own least prefix; this can only be unreduced by the STOP rule. Side-conditions of the form $P \neq \bot$ ensure that the other rules are disjoint from BOT. (The FORK rule overlaps with BOT when P_1 and P_2 are both $\bot$, but in a compatible way.)

The $v \Leftarrow_e s$ side-conditions arise because $\twoheadleftarrow\!\square$ must be able to reverse any expression evaluations it encounters in the trace. Operationally, to reverse $e \Rightarrow v$ requires an implementation of the lower adjoint uneval_e; here we intentionally rely only on its existence (Corollary 1). The expression unevaluation algorithm given in [13], which is also based on traces, can be easily extended to accommodate suspended processes and reused here. To assert that the required lower adjoint exists and is defined for the value we wish to unevaluate, we use the following shorthand, emphasising the symmetry with the corresponding $\boxdot\!\twoheadrightarrow$ side-conditions.

Definition 6 ($\Leftarrow_e$).

$$u \Leftarrow_e s \quad \iff \quad e \Rightarrow v \text{ with } u \sqsubseteq v \text{ and } \mathrm{uneval}_e(u) = s$$

In Lemma 8 below we will verify that the post-conditions of $\boxdot\!\twoheadrightarrow_\Delta$ always satisfy the pre-conditions of $\twoheadleftarrow\!\square$.

To reverse a substitution $e\{v/x\} = e'$, we use the lower adjoint $\mathrm{unsubst}_{e,v,x}$, which exists by Corollary 2. To emphasise the symmetry with substitution, we use an "adjoint pattern-matching" notation $\{u/x\}s$ which matches any $s' \sqsubseteq e'$, and has the effect of binding meta-variables s and u to the the result of $\mathrm{unsubst}_{e,v,x}(s')$. Again we only rely on the existence of the lower adjoint, although at the end of this section, we sketch an operational definition which is given in full in the Appendix, §A.

The following example shows the role of the $\twoheadleftarrow\!\square$ BOT rule in implementing the lower adjoint, which must satisfy the "minimisation" property $\mathrm{unreduce}_{C,\Delta} \circ \mathrm{reduce}_{C,\Delta} \sqsubseteq \mathrm{id}_C$.

Example 8. *Consider the redex $C \triangleq [\![\mathrm{run}\ e]\!]^\alpha$, where $e \Rightarrow \{P\}$, which we reduce as shown below. Now consider the $\bot$-propagation behaviour of $\boxdot\!\twoheadrightarrow_\Delta$.*

$$\begin{array}{lcl} [\![\mathrm{run}\ e]\!]^\alpha & \boxdot\!\twoheadrightarrow_\Delta & [\mathrm{run}\ e.[\![P]\!]^\alpha]^\alpha \\ [\![\bot]\!]^\alpha & \twoheadrightarrow_\Delta & [\![\bot]\!]^\alpha \\ [\mathrm{run}\ e.[\![\bot]\!]^\alpha]^\alpha & \twoheadleftarrow\!\square & [\![\mathrm{run}\ s]\!]^\alpha \end{array}$$

Since $\boxdot\!\twoheadrightarrow_\Delta$ preserves least elements, it maps $[\![\bot]\!]^\alpha$ to $[\![\bot]\!]^\alpha$. However if the RUN rule for $\twoheadleftarrow\!\square$ were applicable here (as shown in red), it would use the

trace to recover the redex $[\![\mathsf{run}\ s]\!]^\alpha$*, using* $\{\bot\} \Leftarrow_e s$*. (Here* $\{\bot\}$ *is a suspended* $\bot$ *process.) Yet* $\mathsf{run}\ s \not\sqsubseteq \bot$*, violating the requirement that the* $\mathit{unreduce}_{C,\Delta} \circ \mathit{reduce}_{C,\Delta}$ *round-trip is deflationary.*

Reversing concurrent computations. We now follow the usual pattern to derive a full concurrent unevaluation relation $\longleftarrow\!\!\square$ from our local notion of unreduction $\twoheadleftarrow\!\!\square$. The informal explanation of $\longleftarrow\!\!\square$, defined at the bottom of Figure 10, is dual to that of $\square\!\!\longrightarrow$. First, a traced reduct $R[D]$ is non-deterministically selected from the traced configuration. The reduct is unreduced to a slice D'' of the original redex, which is spliced back into what remains of the configuration. Then the reverse execution continues with the remaining trace R and the updated configuration. The following example illustrates.

Example 9. *We reverse the step taken in Example 6, where we had* $e \Rightarrow \{P'\}$*. Before reversing, we replace* P' *in the configuration by a smaller (but non-*$\bot$*) process* P''*. Since* $e \Rightarrow \{P'\}$ *and* $P'' \sqsubseteq P'$*, there exists* s *such that* $P'' \Leftarrow_e s$ *by Definition 6, and we can derive*

$$\frac{[\mathsf{run}\ e.[\![P'']\!]^{\alpha.1}]^{\alpha.1} \twoheadleftarrow\!\!\square\ [\![\mathsf{run}\ s]\!]^{\alpha.1} \qquad [[\![\mathsf{run}\ s]\!]^{\alpha.1} \parallel [\![P]\!]^{\alpha.2}]^{\alpha} \longleftarrow\!\!\square\ C'}{[[\mathsf{run}\ e.[\![P'']\!]^{\alpha.1}]^{\alpha.1} \parallel [\![P]\!]^{\alpha.2}]^{\alpha} \longleftarrow\!\!\square\ C'}$$

having non-deterministically chosen $[\mathsf{run}\ e.[\![P'']\!]^{\alpha.1}]^{\alpha.1}$ *as the part of the traced configuration to unreduce. The final state* C' *is a slice of the original configuration* C*.*

The essential property of $\longleftarrow\!\!\square$ is that the $R : \Delta$ which is non-deterministically chosen for unreduction at each step is a *suffix* of the trace $T : G$, implying that Δ is a suffix of G. Thus by construction the portion U of the trace that remains to be unevaluated has a causality graph $G' \sqsubseteq G$, and in particular every "unsynchronisation" is the reversal of a prior synchronisation.

The reverse-evaluation relations just defined give rise to families of functions over configuration prefixes indexed by the domain of the relations. When such a function has as its domain the *codomain* of an upper adjoint identified in §3, we will show that it is in fact the corresponding lower adjoint. Construed algorithmically, these functions constitute an implementation of $\mathsf{unstep}_{C,G}$ for any $C \square\!\!\longrightarrow_G C'$.

Implementing unreduce$_{C,\Delta}$**.** We start by defining a family of $^\dagger\mathsf{unreduce}$ functions indexed by the domain of $\twoheadleftarrow\!\!\square$. The $\dagger$ superscript is to indicate that the functions are a *candidate implementation* of the family of lower adjoints of the same name.

Definition 7 ($^\dagger\mathsf{unreduce}_{R[C]}$ function)**.** *Suppose* $R[C] \twoheadleftarrow\!\!\square\ C'$*. Then define the following function from* $\mathsf{Prefix}(C)$ *to* $\mathsf{Prefix}(C')$*.*

$$^\dagger\mathit{unreduce}_{R[C]} \triangleq \{D \mapsto D' \mid R[D] \twoheadleftarrow\!\!\square\ D'\}$$

From the definition of $\twoheadleftarrow\!\!\square$, it should be clear that $^\dagger\mathsf{unreduce}_{C,\Delta}$ has, as components, lower adjoints of the form uneval_e and $\mathsf{unsubst}_{e,v,x}$; from these we can derive totality and monotonicity.

Lemma 7. $^\dagger\mathit{unreduce}_{R[C]}$ *is total and monotonic.*

The post-conditions of $C \square\!\!\twoheadrightarrow R[C']$ are such that $R[C']$ is in the domain of $\twoheadleftarrow\!\!\square$. In particular, for each $\square\!\!\twoheadrightarrow$ rule which involves expression evaluation, the $\Rightarrow$ side-condition ensures that the $\Leftarrow_e$ side-condition on the corresponding $\twoheadleftarrow\!\!\square$ rule is satisfied. This establishes what is in essence the key *correctness property* for traced reduction: that it produce a trace that enables a reduction to be reversed. In isolation this may not seem like much, but recall that a lower adjoint, when post-composed with the upper adjoint, already has the property of being inflationary on the output. Thus in this lattice-theoretic setting, reversibility is a natural notion of "sufficiency" for traces, and avoids saying anything concrete about their actual structure.

Perera et al. use a similar, but big-step, notion of correctness for expression tracing; there they call it the ability of a trace to *explain* a result. We adapt that notion here to our small-step setting, and show how it relates the correctness of tracing to the suitability of the trace for implementing the lower adjoint.

Definition 8 (Local explanation)**.**
R *locally explains* C *iff* $R[C] \in \mathsf{dom}(\twoheadleftarrow\!\!\square)$.

Lemma 8 (Correctness of $\square\!\!\twoheadrightarrow$)**.**
If $C \square\!\!\twoheadrightarrow R[C']$ *then* R *locally explains* C'*.*

Theorem 1 (Implementation of $\mathsf{unreduce}_{C,\Delta}$)**.**
If $C \square\!\!\twoheadrightarrow R[C']$ *with* $R : \Delta$ *then* $^\dagger\mathit{unreduce}_{R[C']} = \mathit{unreduce}_{C,\Delta}$*.*

Proof. Because of the uniqueness of the lower adjoint, it suffices to show that $(\mathsf{reduce}_C, {}^\dagger\mathsf{unreduce}_{R[C']})$ is a Galois connection. See Appendix, §C.2. □

Implementing unstep$_{C,G}$**.** Now we take a similar approach to the $\longleftarrow\!\!\square$ relation, and show that we obtain an implementation of $\mathsf{unstep}_{C,G}$ for any $C \square\!\!\longrightarrow_G C'$, in which implementations of the form $^\dagger\mathsf{unreduce}_{R[C]}$ appear as components.

Definition 9 ($^\dagger\mathsf{unstep}_{T,C}$ function)**.** *Suppose* $T[C] \longleftarrow\!\!\square\ C'$*. Then define the following function from* $\mathsf{Prefix}(C)$ *to* $\mathsf{Prefix}(C')$*.*

$$^\dagger\mathit{unstep}_{T[C]} \triangleq \{D \mapsto D' \mid T[D] \longleftarrow\!\!\square\ D'\}$$

By unpacking the definition of $\longleftarrow\!\!\square$, it is easy to see that these functions satisfy

$$^\dagger\mathsf{unstep}_{[C]} = \mathsf{id}_C \tag{5}$$

$$^\dagger\mathsf{unstep}_{(T \fatsemi R)[D \uplus D']} = {}^\dagger\mathsf{unstep}_{T[D'' \uplus D']} \circ ({}^\dagger\mathsf{unreduce}_{R[D]} \uplus \mathsf{id}_{D'}) \tag{6}$$

where in Equation 6 we have $(T \fatsemi R)[D \uplus D'] \longleftarrow\!\!\square\ D'' \uplus D$ and $R[C] \twoheadleftarrow\!\!\square\ D''$. As in Definition 4 earlier, the decomposition $T \fatsemi R$ is non-deterministic and thus there are multiple implementations of $^\dagger\mathsf{unstep}_{T,C}$, one for each possible interleaving. We show that all implementations are observationally equivalent by simply showing that they all implement the same lower adjoint, which is a straightforward induction on the size of the trace using Equations 5 and 6 and Theorem 1.

Theorem 2 (Implementation of $\mathsf{unstep}_{C,G}$)**.**
If $C \square\!\!\longrightarrow T[C']$ *and* $T : G$ *then* $^\dagger\mathit{unstep}_{T,C'} = \mathit{unstep}_{C,G}$*.*

Finally, given Theorem 2, the correctness of traced evaluation is immediate: it produces traced configurations which "explain themselves", in that the trace explains the configuration.

Definition 10 (Explanation)**.** T explains C *iff* $T[C] \in \mathsf{dom}(\longleftarrow\!\!\square)$.

Corollary 3 (Correctness of $\square\!\!\longrightarrow$)**.**
If $C \square\!\!\longrightarrow T[C']$ *then* T *explains* C'*.*

Implementing unsubst$_{e,v,x}$**.** We close this section with a brief discussion on substitution. Although the existence of "unsubstitution" functions is trivial, their computation is somewhat less so, relying on the variable-annotated values of the form v_x, which can be thought of as traces of substitutions $x\{v/x\}$. Since variables x

are non-linear there will in general be multiple values of the form v_x in the scope of an unsubstitution.

The idea behind a procedure for unsubstitution for x is to use the variable annotations to locate all such values, and then compute their join using $\sqcup$. Clearly, unsubstitution is only defined when these various uses are compatible, i.e. for any u_x and v_x in the scope of the unevaluation, $u \uparrow v$ and therefore $u \sqcup v$ is defined. This is always the case when an unsubstitution is applied to a prefix of a substituted expression, since all occurrences of x were initially substituted with the same value v_x. Example 4 in §3 illustrated the behaviour of unsubstitution; in the interests of space we omit the definition, and refer the interested reader to the Appendix, §A.

5 Related work

Reversible process calculi. Interest in reversible process calculi has grown recently, with applications including speculative execution, debugging, transactions, and other distributed protocols that require backtracking. A key challenge is to permit backwards execution to leverage concurrency, whilst ensuring *causal consistency*, the property that every state reached during backwards execution is computable by a prefix of the forward execution.

The key difference between existing reversible calculi and ours is that we are able to execute *partial* configurations – forwards and backwards – in order to compute extremal slices. Beyond this obvious difference, there are minor differences in emphasis and technique. Danos and Krivine's *reversible CCS* (RCCS) [5] was early work in this area, recording fork and join actions in thread-local memories and using them for synchronisation during reverse execution, guaranteeing causal consistency. We achieve causal consistency somewhat more directly, since forward execution literally *constructs* a causality graph, and backward execution proceeds by (non-deterministically but confluently) consuming suffixes of that graph.

Lanese et al. [9] extend the RCCS approach to the higher-order pi (HOπ) calculus in a language they call $\rho\pi$ ("reversible pi"). A chief concern of theirs is supporting structural congruences, such as associativity of parallel composition, which do not hold in RCCS. A difference between their approach and ours is that we define composition of traces in such a way that it abstracts over causally irrelevant interleavings, so that syntactic equivalence of traces coincides with causal equivalence of executions. Lanese et al. have to define a non-trivial causal equivalence relation for traces.

Phillips and Ulidowski propose a method for deriving a "reversing" process calculus from any calculus definable using structured operational semantics of a certain kind [14]. They show how to reformulate each operator of the language into a reversible counterpart. Instead of thread-local memories, they use a trace-based approach, similar to ours, retaining process syntax whose elimination would be a source of irreversibility, such as parallel composition, and tagging this residual syntax with synchronisation information. Again the chief difference is that reverse computation is "on the nose", rather than up to a Galois connection.

In recent work, Cristescu et al. [4] develop a compositional semantics for the reversible pi calculus, focusing in particular on name mobility. In our calculus, we forgo pi-calculus style names and use global channels bound by lexically scoped identifiers. Thus we side-step the complexity of "undoing" scope extrusion during backwards execution, at a possible loss in expressivity.

Concurrent dynamic slicing and debugging. An early example of concurrent dynamic slicing is the work of Duesterwald et al., who consider a language with synchronous message-passing [6]. They give a notion of correctness with respect to a slicing criterion, but find that computing least slices is undecidable.

Inspired by Cheng's influential graph-theoretic formulation of concurrent slicing [3], most subsequent work has, not unreasonably, recast dynamic slicing as a dependency-graph reachability problem. (Indeed our unevaluation semantics for traces does essentially the same thing.) However, in the literature to date there is a notable lack of correctness and minimality properties for systems for concurrent dynamic slicing. For example Goswami and Mall consider a language with shared-memory concurrency [7], and Mohapatra et al. tackle slicing for concurrent Java [11], but both present only algorithms, with no formal guarantees. Tallam et al. also develop an approach based on dependency graphs, but again offer only algorithms and empirical results [18]. Moreover in most of this work, the slicing criteria are typically restricted to the (entire) values of particular variables, rather than arbitrary parts of configurations or values.

These limitations makes existing dynamic slicing approaches unsuitable for many important applications, such as offline analysis and provenance tracking, which need to work safely and accurately within a portion of a larger computation. Most other work on concurrent slicing is static rather than dynamic, for which the applications are rather different. However least slices are usually undecidable in the static setting.

Other debugging solutions for concurrency also make use of dependency/causal graphs. For example Kahlon and Wang's *concurrent trace program* format [8] orders events from the same thread by their execution order, and events from different threads via the causal relations implied by fork-join, as happens with our causality graphs. The goal is essentially the same: to eliminate redundant interleavings from consideration.

Distributed provenance and tracing. Souilah et al. introduce a provenance-tracking semantics for distributed systems [17] which is somewhat related to our approach. Every value in their system accumulates a *provenance record* specifying where it came from, including how it was passed between agents. This provenance information can be used by an agent as a form of guarded choice, in order to decide whether to accept a value from a particular sender. Applications include authentication, auditing, and collaborative software. This is an interesting twist on our philosophy, where at the application level "explanations" are completely hidden.

Llorens et al. define an instrumented semantics for CSP which produces a trace (which they call a "track") alongside the output [10]. Their contribution consists of the tracing semantics and a proof of correctness; they leave developing applications such as slicing and debugging for future work. An interesting point of comparison is in the correctness criterion for traces. In their system, a "track" is correct if it faithfully captures the reduction sequence; in ours, if it can be used to unevaluate the final configuration. In fact these amount to much the same thing: in our system, by construction, the unevaluation semantics must reverse exactly the steps recorded in the trace of the forward computation, although not necessarily in the exact reverse order.

6 Conclusion

The key observation underpinning this paper is that computing least dynamic slices is decidable when a language exhibits sufficient *sequentiality* to allow for the existence of the required minima, and non-trivial when it exhibits enough *parallelism* to permit the relative independence of parts of the program. To show that this is possible even in the presence of non-determinism, and moreover that such slices can be calculated without foregoing the

benefits of a concurrent implementation, are two of our main contributions.

To the best of our knowledge, the order-theoretic "problem definition" we presented in §3 is the first purely extensional account of least dynamic slices for concurrency. Often in dynamic slicing, the notion of sufficiency is tied to a particular dependency graph or technique for calculating them (e.g. [16]). Our account is tied to a modest $\perp$-propagating extension of the semantics; we intuitively construe this as a "deterministically parallel" evaluation scheme, where every forward computation is informationally maximal, producing as much output as it can even in the presence of blocking or pending sub-computations.

We close by mentioning one important topic for future work. Like programs, traces may be ordered under erasure. Unevaluation yields a natural criterion for *sufficiency* of a trace with respect to some part of the output: we have defined a trace to *explain* a sliced output if and only if the trace lies in the domain of the unevaluation function for that output. So one future plan is to use a variant of backwards execution to calculate a *least explanation* of a given part of the output, the smallest trace still able to unevaluate it.

References

[1] G. Berry. Stable models of typed λ-calculi. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*, pages 72–89, London, UK, 1978. Springer-Verlag.

[2] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM.

[3] J. Cheng. Slicing concurrent programs: A graph-theoretical approach, 1993.

[4] I. D. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible pi-calculus. In *Proceedings of Twenty-Eighth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2013)*, June 2013.

[5] V. Danos and J. Krivine. Reversible communicating systems. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer Berlin Heidelberg, 2004.

[6] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 497–511, London, UK, 1993. Springer-Verlag.

[7] D. Goswami and R. Mall. Dynamic slicing of concurrent programs. In M. Valero, V. Prasanna, and S. Vajapeyam, editors, *High Performance Computing – HiPC 2000*, volume 1970 of *Lecture Notes in Computer Science*, pages 15–26. Springer, Berlin / Heidelberg, 2000.

[8] V. Kahlon and C. Wang. Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 434–449, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 478–493, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] M. Llorens, J. Oliver, J. Silva, and S. Tamarit. A tracking semantics for CSP. In *Proceedings of the 10th international conference on Mathematics of program construction*, MPC'10, pages 248–270, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] D. Mohapatra, R. Mall, and R. Kumar. An efficient technique for dynamic slicing of concurrent Java programs. In S. Manandhar, J. Austin, U. Desai, Y. Oyanagi, and A. Talukder, editors, *Applied Computing*, volume 3285 of *Lecture Notes in Computer Science*, pages 255–262. Springer, Berlin / Heidelberg, 2004.

[12] R. Perera. *Interactive Functional Programming*. PhD thesis, University of Birmingham, Birmingham, UK, July 2013. `http://etheses.bham.ac.uk/4209/`.

[13] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *ICFP '12: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, 2012.

[14] I. Phillips and I. Ulidowski. Reversing algebraic process calculi. In *FOSSACS '06*, Lecture Notes in Computer Science, pages 246–260. Springer, 2006.

[15] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[16] J. Silva and O. Chitil. Combining algorithmic debugging and program slicing. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 157–166, New York, NY, USA, 2006. ACM.

[17] I. Souilah, A. Francalanza, and V. Sassone. A formal model of provenance in distributed systems. In *First workshop on on Theory and practice of provenance*, TAPP'09, pages 1:1–1:11, Berkeley, CA, USA, 2009. USENIX Association.

[18] S. Tallam, C. Tian, and R. Gupta. Dynamic slicing of multithreaded programs for race detection. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pages 97–106. IEEE, 2008.

Appendix

A Unsubstitution

Definition 11 ($^\dagger\mathrm{unsubst}_x$). *Suppose $e\{v/x\} = e'$. Then define the following function $^\dagger unsubst_{e,v,x}$ from $\mathrm{Prefix}(e')$ to $\mathrm{Prefix}(e, v)$:*

$$^\dagger unsubst_{e,v,x}(s') \triangleq (\theta_{\{x\}}(s'), \phi_{\{x\}}(s'))$$

where the auxiliary partial functions $\theta_\Gamma(-)$ and $\phi_\Gamma(-)$ are defined in Figure 11.

In Figure 11 we abuse notation somewhat and write $\Gamma \setminus x$ to mean $\Gamma \setminus \{x\}$ and $\Gamma \setminus \overrightarrow{g}$ to mean $\Gamma \setminus \{g_1, \ldots, g_n\}$. So that $\phi_\Gamma(-)$ is well-defined (albeit partial), when the right-hand side of any equation takes a join $u \sqcup v$, the reader should assume an implicit guard on the left-hand of the equation asserting $u \uparrow v$.

This function implements the lower adjoint $\mathrm{unsubst}_{e,v,x}$ identified in §3.3.

Lemma 9 (Implementation of $\mathrm{unsubst}_{e,v,x}$). *Suppose $e\{v/x\} = e'$. Then $^\dagger unsubst_{e,v,x} = unsubst_{e,v,x}$.*

B Mutual recursion

The functional language presented in Perera et al. supports mutual recursion, but only as an explicit encoding via additional functional arguments. In a more realistic language, one would prefer to introduce blocks of mutually recursive functions, without the need to pass additional arguments. Here we show how to extend the expression language introduced in §2 with letrec-style mutual recursion.

To add mutual recursion to our language, we introduce a letrec form which defines a collection $\overrightarrow{gx.e}$ of mutually recursive functions, where the identifiers in $\overrightarrow{g}$ are distinct. A functional value additionally carries the collection $\overrightarrow{gx.e}$ of function definitions with which it was mutually defined. The new syntax and evaluation rules are given in Figure 12.

$$\begin{array}{lrl} \text{Expression} & e & ::= \quad \ldots \mid \text{letrec } \overrightarrow{gx.e} \text{ in } e' \\ \text{Value} & u, v & ::= \quad \ldots \mid \langle \overrightarrow{gy.e}, \lambda x.e' \rangle \end{array}$$

$$\boxed{e \Rightarrow v}$$

$$\ldots \quad \frac{e_1 \Rightarrow \langle \overrightarrow{gy.e}, \lambda x.e_3 \rangle \qquad e_2 \Rightarrow v_2 \qquad e_3\{\overrightarrow{gy.e}/\overrightarrow{g}\}\{v_2/x\} \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

$$\frac{e_1\{\overrightarrow{gx.e}/\overrightarrow{g}\}, e_1 \Rightarrow v}{\text{letrec } \overrightarrow{gx.e} \text{ in } e_1 \Rightarrow v}$$

Figure 12. Mutually recursive functions

A simultaneous form of substitution $e'\{\overrightarrow{gx.e}/\overrightarrow{g}\}$ is used for recursive definitions:

Definition 12.

$$e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} \triangleq e'\{\langle \overrightarrow{gx.e}, \lambda x_1.e_1 \rangle / g_1\} \ldots \{\langle \overrightarrow{gx.e}, \lambda x_n.e_n \rangle / g_n\}$$

Simultaneous substitution preserves meets when domain-restricted to prefix-lattices, and induces a family of Galois connections.

Lemma 10.
$(e_1\{\overrightarrow{gx.e_1}/\overrightarrow{g}\}) \sqcap (e_2\{\overrightarrow{gx.e_2}/\overrightarrow{g}\}) = (e_1 \sqcap e_2)\{\overrightarrow{gx.e_1 \sqcap e_2}/\overrightarrow{g}\}$

Corollary 4. *Suppose $e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} = e''$. Overloading $subst_{e',\overrightarrow{gx.e}}$ to mean $-\{-/\overrightarrow{g}\}$ domain-restricted to $\mathrm{Prefix}(e', \overrightarrow{gx.e})$, there exists a unique monotonic function $unsubst_{e',\overrightarrow{gx.e}}$ from $\mathrm{Prefix}(e'')$ to $\mathrm{Prefix}(e', \overrightarrow{gx.e})$ satisfying*

1. $unsubst_{e',\overrightarrow{gx.e}} \circ subst_{e',\overrightarrow{gx.e}} = id_{(e',\overrightarrow{gx.e})}$
2. $subst_{e',\overrightarrow{gx.e}} \circ unsubst_{e',\overrightarrow{gx.e}} \sqsupseteq id_{e''}$

The lower adjoint forms can be implemented as follows. The goal for simultaneous unsubstitution with respect to a set of simultaneous function names $\overrightarrow{g}$ in e' must recover a set of recursive definitions $\overrightarrow{gx.e}$, where each recursive definition is the least upper bound of all the uses of that function that occur in e'. "Uses" include not only direct uses of the function body e_i for a functional value $\langle \overrightarrow{gx.s}, \lambda x_i.e_i \rangle_{g_i}$ but also indirect uses of the set of recursive functions with which g_i was mutually defined. First we define the following partial function.

Definition 13. *For any sequence of variables $\overrightarrow{g}$, define the partial function $^\dagger unsubst_{\overrightarrow{g}}$ to take an expression e' to the expression s and recursive definitions $\overrightarrow{gx.e}$ such that*

$$g_i x_i.e_i = (g_i y_i.s_i') \sqcup (\bigsqcup \overrightarrow{\delta})_i$$

where

$$e' = \{\langle \delta_n, \lambda y_n.s_n' \rangle / g_n\} \ldots \{\langle \delta_1, \lambda y_1.s_1' \rangle / g_1\} s$$

The partiality arises because both the joins and the individual unsubstitutions of each g_n are not necessarily defined. But suitably restricted, this definition does indeed give the required lower adjoints.

Lemma 11 ($^\dagger\mathrm{unsubst}_{e',\overrightarrow{gx.e}} = \mathrm{unsubst}_{e',\overrightarrow{gx.e}}$). *Suppose $e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} = e''$, and write $^\dagger unsubst_{e',\overrightarrow{gx.e}}$ for $^\dagger unsubst_{\overrightarrow{g}}$ domain-restricted to $\mathrm{Prefix}(e'')$. Then $^\dagger unsubst_{e',\overrightarrow{gx.e}} = unsubst_{e',\overrightarrow{gx.e}}$.*

Proof. See §C.3. □

To illustrate Definition 13, we again adopt an "adjoint pattern-matching" notation so that, given a substitution $e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} = e''$, the pattern $\{\overrightarrow{gx.s}/\overrightarrow{g}\}s'$ matches any expression $s'' \sqsubseteq e''$, with $(s', \overrightarrow{gx.s}) = {}^\dagger\mathrm{unsubst}_{e',\overrightarrow{gx.e}}(s'')$.

Example 10. *Mutual recursion example.*

C Proofs

Here we give longer proofs omitted from the main body of the paper.

C.1 Proof of Lemma 2

Proof. Suppose $C \twoheadrightarrow_\Delta C'$ where $\Gamma \vdash \Delta\Gamma'$, and consider any D, $D' \sqsubseteq C$. If $D = \bot_\Gamma$ then $D \twoheadrightarrow \bot_{\Gamma'}$. But then $D \sqcap D' \twoheadrightarrow \bot_{\Gamma'}$. If $D' = \bot_\Gamma$ a similar argument applies by commutativity. Otherwise we proceed by case analysis, considering only the cases when $D, D' \neq \bot_\Gamma$, using Lemma 4 for the cases involving expression evaluation, and Lemma 5 for the cases involving substitution.

Case STOP. The conclusion is immediate since $D = [\![\mathbf{0}]\!]^\alpha = D'$.

Case NEW. We have

$$[\![\nu x.P]\!]^\alpha \twoheadrightarrow [\![P\{c/x\}]\!]^\alpha$$

$$[\![\nu x.P']\!]^\alpha \twoheadrightarrow [\![P'\{c/x\}]\!]^\alpha$$

$$\theta_\Gamma(\bot) = \bot$$
$$\theta_\Gamma(v_y) = \begin{cases} y & \text{if } x = y \\ v_y & \text{otherwise} \end{cases}$$
$$\theta_\Gamma(()) = ()$$
$$\theta_\Gamma(\lambda x.e) = \lambda x.\theta_{\Gamma\setminus x}(e)$$
$$\theta_\Gamma(e_1\ e_2) = \theta_\Gamma(e_1)\ \theta_\Gamma(e_2)$$
$$\theta_\Gamma(\text{letrec } \overrightarrow{gx.\hat{e}} \text{ in } e') = \text{letrec } \overrightarrow{gx.\hat{e}} \text{ in } \theta_{\Gamma\setminus\overrightarrow{g}}(e')$$
$$\theta_\Gamma((e_1, e_2)) = (\theta_\Gamma(e_1), \theta_\Gamma(e_2))$$
$$\theta_\Gamma(\text{fst } e) = \text{fst } \theta_\Gamma(e)$$
$$\theta_\Gamma(\text{inl } e) = \text{inl } \theta_\Gamma(e)$$
$$\theta_\Gamma(\text{case } e \{\text{inl } x_1.e_1; \text{inr } x_2.e_2\}) = \text{case } \theta_\Gamma(e) \{\text{inl } x_1.\theta_{\Gamma\setminus x_1}(e_1); \text{inr } x_2.\theta_{\Gamma\setminus x_2}(e_2)\}$$

$$\phi_\Gamma(\bot) = \bot$$
$$\phi_{\{x\}}(v_y) = \begin{cases} v & \text{if } x = y \\ \bot & \text{otherwise} \end{cases}$$
$$\phi_\Gamma(()) = \bot$$
$$\phi_\Gamma(\lambda x.e) = \phi_{\Gamma\setminus x}(e)$$
$$\phi_\Gamma(e_1\ e_2) = \phi_\Gamma(e_1) \sqcup \phi_\Gamma(e_2)$$
$$\phi_\Gamma(\text{letrec } \overrightarrow{gx.\hat{e}} \text{ in } e') = \phi_{\Gamma\setminus\overrightarrow{g}}(e')$$
$$\phi_\Gamma((e_1, e_2)) = \phi_\Gamma(e_1) \sqcup \phi_\Gamma(e_2)$$
$$\phi_\Gamma(\text{fst } e) = \phi_\Gamma(e)$$
$$\phi_\Gamma(\text{inl } e) = \phi_\Gamma(e)$$
$$\phi_\Gamma(\text{case } e \{\text{inl } x_1.e_1; \text{inr } x_2.e_2\}) = \phi_\Gamma(e) \sqcup \phi_{\Gamma\setminus x_1}(e_1) \sqcup \phi_{\Gamma\setminus x_2}(e_2)$$

$$\theta_\Gamma(\bot) = \bot$$
$$\theta_\Gamma(\mathbf{0}) = \mathbf{0}$$
$$\theta_\Gamma(\text{run } e) = \text{run } \theta_\Gamma(e)$$
$$\theta_\Gamma(P_1 \parallel P_2) = \theta_\Gamma(P_1) \parallel \theta_\Gamma(P_2)$$
$$\theta_\Gamma(\nu x.P) = \nu x.\theta_{\Gamma\setminus x}(P)$$
$$\theta_\Gamma(e_1\langle e_2\rangle) = \theta_\Gamma(e_1)\langle\theta_\Gamma(e_2)\rangle$$
$$\theta_\Gamma(e(x).P) = \theta_\Gamma(e)(x).\theta_{\Gamma\setminus x}(P)$$

$$\phi_\Gamma(\bot) = \bot$$
$$\phi_\Gamma(\mathbf{0}) = \bot$$
$$\phi_\Gamma(\text{run } e) = \phi_\Gamma(e)$$
$$\phi_\Gamma(P_1 \parallel P_2) = \phi_\Gamma(P_1) \sqcup \phi_\Gamma(P_2)$$
$$\phi_\Gamma(\nu x.P) = \phi_{\Gamma\setminus x}(P)$$
$$\phi_\Gamma(e_1\langle e_2\rangle) = \phi_\Gamma(e_1) \sqcup \phi_\Gamma(e_2)$$
$$\phi_\Gamma(e(x).P) = \phi_\Gamma(e) \sqcup \phi_{\Gamma\setminus x}(P)$$

Figure 11. Expression $\theta_\Gamma(-)$ and value $\phi_\Gamma(-)$ components of unsubstitution with respect to Γ, where $\Gamma = \varnothing$ or $\Gamma = \{x\}$.

where $D = [\![\nu x.P]\!]^\alpha$ and $D' = [\![\nu x.P']\!]^\alpha$. By congruence $D \sqcap D' = [\![\nu x.P \sqcap P']\!]^\alpha$, and then by the definition of $\twoheadrightarrow$

$$[\![\nu x.P \sqcap P']\!]^\alpha \twoheadrightarrow [\![(P \sqcap P')\{c/x\}]\!]^\alpha$$

Then $[\![(P \sqcap P')\{c/x\}]\!]^\alpha = [\![P\{c/x\}]\!]^\alpha \sqcap [\![P'\{c/x\}]\!]^\alpha$ by Lemma 5 and congruence.

Case FORK. We have

$$[\![P_1 \parallel P_2]\!]^\alpha \twoheadrightarrow [\![P_1]\!]^{\alpha.1}, [P_2]^{\alpha.2}$$

$$[\![P_1' \parallel P_2']\!]^\alpha \twoheadrightarrow [\![P_1']\!]^{\alpha.1}, [P_2']^{\alpha.2}$$

where $D = [\![P_1 \parallel P_2]\!]^\alpha$ and $D' = [\![P_1' \parallel P_2']\!]^\alpha$. By congruence $D \sqcap D' = [\![P_1 \sqcap P_1' \parallel P_2 \sqcap P_2']\!]^\alpha$, and then by the definition of $\twoheadrightarrow$

$$[\![P_1 \sqcap P_1' \parallel P_2 \sqcap P_2']\!]^\alpha \twoheadrightarrow [\![P_1 \sqcap P_1']\!]^{\alpha.1}, [P_2 \sqcap P_2']^{\alpha.2}$$

Then we have $[\![P_1 \sqcap P_1']\!]^{\alpha.1}, [\![P_2 \sqcap P_2']\!]^{\alpha.2} = ([\![P_1]\!]^{\alpha.1}, [\![P_2]\!]^{\alpha.2}) \sqcap ([\![P_1']\!]^{\alpha.1}, [\![P_2']\!]^{\alpha.2})$ by congruence.

Case SEND-ON. We have

$$[\![e_1\langle e_2\rangle]\!]^\alpha \twoheadrightarrow [\![c\langle e_2\rangle]\!]^\alpha$$

$$[\![e_1'\langle e_2'\rangle]\!]^\alpha \twoheadrightarrow [\![c'\langle e_2'\rangle]\!]^\alpha$$

where $D = [\![e_1\langle e_2\rangle]\!]^\alpha$ and $D' = [\![e_1'\langle e_2'\rangle]\!]^\alpha$, with $e_1 \Rightarrow c$ and $e_1' \Rightarrow c'$. By congruence $D \sqcap D' = [\![(e_1 \sqcap e_1')\langle e_2 \sqcap e_2'\rangle]\!]^\alpha$, and then by the definition of $\twoheadrightarrow$

$$[\![(e_1 \sqcap e_1')\langle e_2 \sqcap e_2'\rangle]\!]^\alpha \twoheadrightarrow [\![(c \sqcap c')\langle e_2 \sqcap e_2'\rangle]\!]^\alpha$$

with $e_1 \sqcap e_1' \Rightarrow c \sqcap c'$ by Lemma 4. Then $[\![(c \sqcap c')\langle e_2 \sqcap e_2'\rangle]\!]^\alpha = [\![c\langle e_2\rangle]\!]^\alpha \sqcap [\![c'\langle e_2'\rangle]\!]^\alpha$ by congruence.

Case SEND-READY. We have

$$[\![c\langle e_2\rangle]\!]^\alpha \twoheadrightarrow [\![c\langle v\rangle]\!]^\alpha$$

$$[\![c'\langle e_2'\rangle]\!]^\alpha \twoheadrightarrow [\![c'\langle v'\rangle]\!]^\alpha$$

where $D = [\![c\langle e_2\rangle]\!]^\alpha$ and $D' = [\![c'\langle e_2'\rangle]\!]^\alpha$, with $e_2 \Rightarrow v$ and $e_2' \Rightarrow v'$. By congruence $D \sqcap D' = [\![(c \sqcap c')\langle e_2 \sqcap e_2'\rangle]\!]^\alpha$, and then by the definition of $\twoheadrightarrow$

$$[\![(c \sqcap c')\langle e_2 \sqcap e_2'\rangle]\!]^\alpha \twoheadrightarrow [\![(c \sqcap c')\langle v \sqcap v'\rangle]\!]^\alpha$$

with $e_2 \sqcap e_2' \Rightarrow v \sqcap v'$ by Lemma 4. Then $[\![(c \sqcap c')\langle v \sqcap v'\rangle]\!]^\alpha = [\![c\langle v\rangle]\!]^\alpha \sqcap [\![c'\langle v'\rangle]\!]^\alpha$ by congruence.

Case RCV-READY. We have

$$[\![e(x).P]\!]^\alpha \twoheadrightarrow [\![c(x).P]\!]^\alpha$$

$$[\![e'(x).P']\!]^\alpha \twoheadrightarrow [\![c'(x).P']\!]^\alpha$$

where $D = [\![e(x).P]\!]^\alpha$ and $D' = [\![e'(x).P']\!]^\alpha$, with $e \Rightarrow c$ and $e' \Rightarrow c'$. By congruence $D \sqcap D' = [\![(e \sqcap e')(x).P \sqcap P']\!]^\alpha$, and then by the definition of $\twoheadrightarrow$

$$[\![(e \sqcap e')(x).P \sqcap P']\!]^\alpha \twoheadrightarrow [\![(c \sqcap c')(x).P \sqcap P']\!]^\alpha$$

with $e \sqcap e' \Rightarrow c \sqcap c'$ by Lemma 4. Then $[\![(c \sqcap c')(x).P \sqcap P']\!]^\alpha = [\![c(x).P]\!]^\alpha \sqcap [\![c'(x).P']\!]^\alpha$ by congruence.

Cases JOIN-BOT-SEND ***and*** JOIN-BOT-RCV. If $D = [\![\bot]\!]^\alpha, [\![c(x).P]\!]^\beta$ or $D = [\![c\langle v\rangle]\!]^\alpha, [\![\bot]\!]^\beta$ then $D \twoheadrightarrow [\![\bot]\!]^{\alpha.\beta}$, and we reason similarly to when $D = \bot$, since we also have $D \sqcap D' \twoheadrightarrow [\![\bot]\!]^{\alpha.\beta}$. If $D' = [\![\bot]\!]^\alpha, [\![c(x).P]\!]^\beta$ or $D' = [\![c\langle v\rangle]\!]^\alpha, [\![\bot]\!]^\beta$ a similar argument applies by commutativity.

Case JOIN-BOT-CHAN. Otherwise, $D = [\![c\langle v\rangle]\!]^\alpha, [\![c'(x).P]\!]^\beta$ and $D' = [\![c''\langle v'\rangle]\!]^\alpha, [\![c'''(x).P']\!]^\beta$. If $c = \bot$, then D is stuck in virtue of the side-condition on the JOIN rule. But then $D \sqcap D' = [\![\bot\langle v \sqcap v'\rangle]\!]^\alpha, [\![(c' \sqcap c''')(x).P \sqcap P']\!]^\beta$, which is stuck for the same reason. If $c' = \bot$, then D is again stuck in virtue of the side-condition. But then $D \sqcap D' = [\![(c \sqcap c'')\langle v \sqcap v'\rangle]\!]^\alpha, [\![\bot(x).P \sqcap P']\!]^\beta$, which is stuck for the same reason. In either case $\text{step}(D) \sqcap \text{step}(D') = \bot = \text{step}(D \sqcap D')$. If $c'' = \bot$ or $c''' = \bot$ a similar argument applies by commutativity.

Case JOIN. We have

$$[\![c\langle v\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta \twoheadrightarrow [\![P\{v/x\}]\!]^{\alpha.\beta}$$

$$[\![c\langle v'\rangle]\!]^\alpha, [\![c(x).P']\!]^\beta \twoheadrightarrow [\![P'\{v'/x\}]\!]^{\alpha.\beta}$$

where $c \neq \bot$, and also $D = [\![c\langle v\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta$ and $D' = [\![c\langle v'\rangle]\!]^\alpha, [\![c(x).P']\!]^\beta$. Then $D \sqcap D' = [\![c\langle v \sqcap v'\rangle]\!]^\alpha, [\![(c)(x).P \sqcap P']\!]^\beta$ by congruence, and by the definition of $\twoheadrightarrow$

$$[\![c\langle v \sqcap v'\rangle]\!]^\alpha, [\![c(x).P \sqcap P']\!]^\beta \twoheadrightarrow [\![(P \sqcap P')\{(v \sqcap v')/x\}]\!]^{\alpha.\beta}$$

Then $[\![(P \sqcap P')\{(v \sqcap v')/x\}]\!]^{\alpha.\beta} = [\![P\{v/x\}]\!]^{\alpha.\beta} \sqcap [\![P'\{v'/x\}]\!]^{\alpha.\beta}$ by Lemma 5.

Case RUN. We have

$$[\![\mathsf{run}\ e]\!]^\alpha \twoheadrightarrow [\![P]\!]^\alpha$$

$$[\![\mathsf{run}\ e']\!]^\alpha \twoheadrightarrow [\![P']\!]^\alpha$$

where $D = [\![\mathsf{run}\ e]\!]^\alpha$ and $D' = [\![\mathsf{run}\ e']\!]^\alpha$, with $e \Rightarrow \{P\}$ and $e' \Rightarrow \{P'\}$. By congruence $D \sqcap D' = [\![\mathsf{run}\ e \sqcap e']\!]^\alpha$, and then by the definition of $\twoheadrightarrow$

$$[\![\mathsf{run}\ e \sqcap e']\!]^\alpha \twoheadrightarrow [\![P \sqcap P']\!]^\alpha$$

with $e \sqcap e' \Rightarrow \{P\} \sqcap \{P'\}$ by Lemma 4. Then $[\![P \sqcap P']\!]^\alpha = [\![P]\!]^\alpha \sqcap [\![P']\!]^\alpha$ by congruence.

□

C.2 Proof of Theorem 1

Proof. Suppose $\Gamma \vdash C$ and $\Gamma \vdash R : \Gamma'$ with $C \ \square\!\!\twoheadrightarrow R[C']$.

(1) $\boldsymbol{\mathsf{reduce}_C \circ {}^\dagger\mathsf{unreduce}_{R[C']} \sqsupseteq \mathsf{id}_{C'}}$. Consider any $D' \sqsubseteq C'$; we proceed by case analysis on $R[D'] \twoheadleftarrow\!\!\square\ D$.

Case BOT.

$$R[\bot_{\Gamma'}] \twoheadleftarrow\!\!\square\ \bot_\Gamma$$

The conclusion is immediate.

Case STOP.

$$[\![\mathbf{0}]\!]^\alpha \twoheadleftarrow\!\!\square\ [\mathbf{0}]^\alpha$$

By the definition of $\twoheadrightarrow$

$$[\mathbf{0}]^\alpha \twoheadrightarrow [\![\mathbf{0}]\!]^\alpha$$

Case NEW.

$$[\nu x.[\![P]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\square\ [\![\nu x.P']\!]^\alpha$$

where $P = \{c'/x\}P'$. By the definition of $\twoheadrightarrow$

$$[\![\nu x.P']\!]^\alpha \twoheadrightarrow [\![P'\{c/x\}]\!]^\alpha$$

with $P'\{c/x\} \sqsupseteq P$ by Corollary 2.

Case FORK.

$$[[\![P_1]\!]^{\alpha.1} \parallel [\![P_2]\!]^{\alpha.2}]^\alpha \twoheadleftarrow\!\!\square\ [\![P_1 \parallel P_2]\!]^\alpha$$

By the definition of $\twoheadrightarrow$

$$[\![P_1 \parallel P_2]\!]^\alpha \twoheadrightarrow [\![P_1]\!]^{\alpha.1}, [\![P_2]\!]^{\alpha.2}$$

Case SEND-ON.

$$[e_1.[\![c\langle e_2\rangle]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\square\ [\![s_1\langle e_2\rangle]\!]^\alpha$$

with $c \Leftarrow_{e_1} s_1$. Since $s_1 \sqsubseteq e_1$, by the definition of $\twoheadrightarrow$

$$[\![s_1\langle e_2\rangle]\!]^\alpha \twoheadrightarrow [\![c'\langle e_2\rangle]\!]^\alpha$$

with $s_1 \Rightarrow c'$. Then $c' \sqsupseteq c$ by Corollary 1.

Case SEND-READY.

$$[e_2.[\![c\langle v\rangle]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\square\ [\![c\langle s_2\rangle]\!]^\alpha$$

with $v \Leftarrow_{e_2} s_2$. Since $s_2 \sqsubseteq e_2$, by the definition of $\twoheadrightarrow$

$$[\![c\langle s_2\rangle]\!]^\alpha \twoheadrightarrow [\![c\langle v'\rangle]\!]^\alpha$$

with $s_2 \Rightarrow v'$. Then $v' \sqsupseteq v$ by Corollary 1.

Case RCV-READY.

$$[e.[\![c(x).P]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\square\ [\![s(x).P]\!]^\alpha$$

with $c \Leftarrow_e s$. Since $s \sqsubseteq e$, by the definition of $\twoheadrightarrow$

$$[\![s(x).P]\!]^\alpha \twoheadrightarrow [\![c'(x).P]\!]^\alpha$$

with $s \Rightarrow c'$. Then $c' \sqsupseteq c$ by Corollary 1.

Case JOIN.

$$[c\langle v\rangle]^\alpha, [c(x).[\![P]\!]^{\alpha.\beta}]^\beta \twoheadleftarrow\!\!\square\ [\![c\langle u\rangle]\!]^\alpha, [\![c(x).P']\!]^\beta$$

where $P = \{u/x\}P'$ and $c \neq \bot$. Since $c \neq \bot$, by the definition of $\twoheadrightarrow$

$$[\![c\langle u\rangle]\!]^\alpha, [\![c(x).P']\!]^\beta \twoheadrightarrow [\![P'\{u/x\}]\!]^{\alpha.\beta}$$

with $P'\{u/x\} \sqsupseteq P$ by Corollary 2.

Case RUN.

$$[\mathsf{run}\ e.[\![P]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\square\ [\![\mathsf{run}\ s]\!]^\alpha$$

with $\{P\} \Leftarrow_e s$. Since $s \sqsubseteq e$, by the definition of $\twoheadrightarrow$

$$[\![\mathsf{run}\ s]\!]^\alpha \twoheadrightarrow [\![P']\!]^\alpha$$

with $s \Rightarrow \{P'\}$. Then $\{P'\} \sqsupseteq \{P\}$ by Corollary 1 and so $P' \sqsupseteq P$.

(2) $\boldsymbol{{}^\dagger\mathsf{unreduce}_{R[C']} \circ \mathsf{reduce}_C \sqsubseteq \mathsf{id}_C}$. Consider any $D \sqsubseteq C$. If D is stuck then $\mathrm{reduce}_C(D) = \bot_{\Delta'}$ by definition and then ${}^\dagger\mathrm{unreduce}_{R[C']}(\bot_{\Delta'}) = \bot_\Delta$ also by definition. Otherwise we proceed by case analysis on $D \ \square\!\!\twoheadrightarrow R[D']$.

Case BOT.

$$\bot_\Gamma \ \square\!\!\twoheadrightarrow \bot_\Delta[\bot_{\Gamma'}]$$

By the definition of $\twoheadleftarrow\!\!\square$

$$\bot_\Delta[\bot_{\Gamma'}] \twoheadleftarrow\!\!\square\ \bot_\Gamma$$

Case STOP.

$$[\mathbf{0}]^\alpha \ \square\!\!\twoheadrightarrow [\![\mathbf{0}]\!]^\alpha$$

By the definition of $\twoheadleftarrow\!\!\square$

$$[\![\mathbf{0}]\!]^\alpha \twoheadleftarrow\!\!\square\ [\mathbf{0}]^\alpha$$

Case NEW.

$$[\![\nu x.P]\!]^\alpha \ \square\!\!\twoheadrightarrow [\nu x.[\![P\{c/x\}]\!]^\alpha]^\alpha$$

Note that $P\{c/x\} = \{c/x\}P$ by Corollary 2. Then by the definition of $\twoheadleftarrow\!\!\square$

$$[\nu x.[\![\{c/x\}P]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\square\ [\![\nu x.P]\!]^\alpha$$

Case FORK.

$$[\![P_1 \parallel P_2]\!]^\alpha \ \square\!\!\twoheadrightarrow [[\![P_1]\!]^{\alpha.1} \parallel [\![P_2]\!]^{\alpha.2}]^\alpha$$

By the definition of $\twoheadleftarrow\!\!\square$

$$[[\![P_1]\!]^{\alpha.1} \parallel [\![P_2]\!]^{\alpha.2}]^\alpha \twoheadleftarrow\!\!\square\ [\![P_1 \parallel P_2]\!]^\alpha$$

Case SEND-ON.

$$[\![e_1\langle e_2\rangle]\!]^\alpha \ \square\!\!\twoheadrightarrow [e_1.[\![c\langle e_2\rangle]\!]^\alpha]^\alpha$$

with $e_1 \Rightarrow c$. Since $c \in \mathrm{dom}(\mathrm{uneval}_{e_1})$, by the definition of $\twoheadleftarrow\!\!\square$

$$[e_1.[\![c\langle e_2\rangle]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\square\ [\![s_1\langle e_2\rangle]\!]^\alpha$$

with $c \Leftarrow_{e_1} s_1$. Then $s_1 \sqsubseteq e_1$ by Corollary 1.

Case SEND-READY**.**

$$[\![c\langle e_2\rangle]\!]^\alpha \boxdot\!\!\twoheadrightarrow [e_2.[\![c\langle v\rangle]\!]^\alpha]^\alpha$$

with $e_2 \Rightarrow v$. Since $v \in \mathsf{dom}(\mathsf{uneval}_{e_2})$, by the definition of $\boxdot\!\!\twoheadrightarrow$

$$[e_2.[\![c\langle v\rangle]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\boxdot [\![c\langle s_2\rangle]\!]^\alpha$$

with $v \Leftarrow_{e_2} s_2$. Then $s_2 \sqsubseteq e_2$ by Corollary 1.

Case RCV-READY**.**

$$[\![e(x).P]\!]^\alpha \boxdot\!\!\twoheadrightarrow [e.[\![c(x).P]\!]^\alpha]^\alpha$$

with $e \Rightarrow c$. Since $c \in \mathsf{dom}(\mathsf{uneval}_e)$, by the definition of $\twoheadleftarrow\!\!\boxdot$

$$[e.[\![c(x).P]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\boxdot [\![s(x).P]\!]^\alpha$$

with $c \Leftarrow_e s$. Then $s \sqsubseteq e$ by Corollary 1.

Case JOIN**.**

$$[\![c\langle v\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta \boxdot\!\!\twoheadrightarrow [c\langle v\rangle]^\alpha, [c(x).[\![P\{v/x\}]\!]^{\alpha.\beta}]^\beta$$

Note that $P\{v/x\} = \{v/x\}P$ by Corollary 2. By the definition of $\twoheadleftarrow\!\!\boxdot$

$$[c\langle v\rangle]^\alpha, [c(x).[\![P\{v/x\}]\!]^{\alpha.\beta}]^\beta \twoheadleftarrow\!\!\boxdot [\![c\langle v\rangle]\!]^\alpha, [\![c(x).P]\!]^\beta$$

Case JOIN-BOT-CHAN**,** JOIN-BOT-SEND ***and*** JOIN-BOT-RCV**.** If we have any of the following traced reductions

$$[\![c\langle v\rangle]\!]^\alpha, [\![c'(x).P]\!]^\beta \boxdot\!\!\twoheadrightarrow [\bot]^\alpha, [\bot.[\![\bot]\!]^{\alpha.\beta}]^\beta$$

$$[\![\bot]\!]^\alpha, [\![c'(x).P]\!]^\beta \boxdot\!\!\twoheadrightarrow [\bot]^\alpha, [\bot.[\![\bot]\!]^{\alpha.\beta}]^\beta$$

$$[\![c\langle v\rangle]\!]^\alpha, [\![\bot]\!]^\beta \boxdot\!\!\twoheadrightarrow [\bot]^\alpha, [\bot.[\![\bot]\!]^{\alpha.\beta}]^\beta$$

then by the definition of $\twoheadleftarrow\!\!\boxdot$

$$[\bot]^\alpha, [\bot.[\![\bot]\!]^{\alpha.\beta}]^\beta \twoheadleftarrow\!\!\boxdot [\![\bot]\!]^\alpha, [\![\bot]\!]^\beta$$

Case RUN**.**

$$[\![\mathsf{run}\ e]\!]^\alpha \boxdot\!\!\twoheadrightarrow [\mathsf{run}\ e.[\![P]\!]^\alpha]^\alpha$$

where $e \Rightarrow \{P\}$. Since $\{P\} \in \mathsf{dom}(\mathsf{uneval}_e)$, by the definition of $\twoheadleftarrow\!\!\boxdot$

$$[\mathsf{run}\ e.[\![P]\!]^\alpha]^\alpha \twoheadleftarrow\!\!\boxdot [\![\mathsf{run}\ s]\!]^\alpha$$

with $\{P\} \Leftarrow_e s$. Then $s \sqsubseteq e$ by Corollary 1. □

C.3 Proof of Lemma 11

Proof. First we show ${}^\dagger\mathsf{unsubst}_{e',\overrightarrow{gx.e}} \circ \mathsf{subst}_{e',\overrightarrow{gx.e}} = \mathsf{id}_{(e',\overrightarrow{gx.e})}$. Suppose $e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} = \{\overrightarrow{gy.s}/\overrightarrow{g}\}s'$. We want to show that $(s', \overrightarrow{gy.s}) = (e', \overrightarrow{gx.e})$. We have:

$$\begin{aligned}
&e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} \\
&= e'\{\langle\overrightarrow{gx.e}, \lambda x_1.e_1\rangle/g_1\}\ldots\{\langle\overrightarrow{gx.e}, \lambda x_n.e_n\rangle/g_n\} && \text{Definition 12} \\
&= \{\langle\overrightarrow{gx.e}, \lambda x_n.e_n\rangle/g_n\}\ldots\{\langle\overrightarrow{gx.e}, \lambda x_1.e_1\rangle/g_1\}e' && \text{Corollary 2}
\end{aligned}$$

By Definition 13 we have $s' = e'$, and $g_i y_i.s_i = g_i x_i.e_i \sqcup (\overrightarrow{gx.e})_i = g_i x_i.e_i$ for every i.

Now we show $\mathsf{subst}_{e',\overrightarrow{gx.e}} \circ {}^\dagger\mathsf{unsubst}_{e',\overrightarrow{gx.e}} \sqsupseteq \mathsf{id}_{e''}$. Suppose $e'' = \{\overrightarrow{gx.e}/\overrightarrow{g}\}e'$. We want $e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} \sqsupseteq e''$. By Definition 13, we have $e'' = \{\langle\delta_n, \lambda x_n.s'_n\rangle/g_n\}\ldots\{\langle\delta_1, \lambda x_1.s'_1\rangle/g_1\}e'$, where $g_i x_i.e_i = (g_i x_i.s'_i) \sqcup (\bigsqcup \overrightarrow{\delta})_i$. By Corollary 2, which also implies monotonicity, $e'\{\langle\delta_1, \lambda x_1.s'_1\rangle/g_1\}\ldots\{\langle\delta_n, \lambda x_n.s'_n\rangle/g_n\} \sqsupseteq e''$. Since $\overrightarrow{gx.e} \sqsupseteq \overrightarrow{\delta_i}$ and $e_i \sqsupseteq s'_i$ for every i, we also have $e'\{\langle\overrightarrow{gx.e}, \lambda x_1.e_1\rangle/g_1\}\ldots\{\langle\overrightarrow{gx.e}, \lambda x_n.e_n\rangle/g_n\} \sqsupseteq e''$ by monotonicity. Then $e'\{\overrightarrow{gx.e}/\overrightarrow{g}\} \sqsupseteq e''$ by Definition 12. □