Causally consistent dynamic slicing

Roly Perera[1,2], Deepak Garg[3], and James Cheney[1]

1 **Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, UK**
`rperera@inf.ed.ac.uk`, `jcheney@inf.ed.ac.uk`
2 **School of Computing Science, University of Glasgow, Glasgow, UK**
`roly.perera@glasgow.ac.uk`
3 **Max Planck Institute for Software Systems, Saarbrücken, Germany**
`dg@mpi-sws.org`

Abstract

We offer a lattice-theoretic account of the problem of dynamic slicing for π-calculus, building on prior work in the sequential setting. For any particular run of a concurrent program, we exhibit a Galois connection relating forward and backward slices of the initial and terminal configurations. We prove that, up to lattice isomorphism, the same Galois connection arises for any causally equivalent execution, allowing an efficient concurrent implementation of slicing via a standard interleaving semantics. Our approach has been formalised in the dependently-typed programming language Agda.

1998 ACM Subject Classification D.1.3 Concurrent Programming; D.2.5 Testing and debugging

Keywords and phrases π-calculus; dynamic slicing; causal equivalence; Galois connection

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Dynamic slicing, due originally to Weiser [20], is a dynamic analysis technique with applications in debugging, security and provenance tracking. The basic goal is to identify a sub-program, or *program slice*, that may affect an outcome of interest called the *slicing criterion*, such as the value of a variable. Dynamic slicing in concurrent settings is often represented as a graph reachability problem, thanks to influential work by Cheng [4]. However, most prior work on dynamic slicing for concurrency does not yield minimum slices, nor allow particularly flexible slicing criteria, such as arbitrary parts of configurations. Systems work on concurrent slicing [10, 15, 19] tends to be largely informal.

Perera et al [16] developed an approach where backward dynamic slicing is treated as a kind of (abstract) reverse execution or "rewind" and forward slicing as a kind of (abstract) re-execution or "replay". Forward and backward slices are related by a Galois connection, ensuring the existence of minimal slices. This idea is straightforward in the sequential setting of the earlier work. However, generalising it to concurrent programs is non-trivial. Suppose we run a concurrent computation, discover a bug, and then wish to compute a dynamic slice. It would likely be prohibitively slow, or impractical for other reasons, to insist that the slice be computed using the exact interleaving of the original run. On the other hand, computing the slice using a brand-new concurrent execution may make different non-deterministic choices, producing a slice of a different computation than the one intended.

Intuitively, any execution which exhibits the same causal structure should be adequate for computing the slice, and any practical approach to concurrent slicing should take advantage of this. Cristescu et al [5] make a similar observation about reversible concurrency, arguing

	Scheduler node 1	*Scheduler node 2*	A_1	A_2
	$\mathbf{a_1}.c_1.(b_1.\overline{c_2}.\overline{r_1} + \overline{c_2}.b_1.\overline{r_1})$	$\overline{c_1}.a_2.c_2.(b_2.\overline{c_1}.\overline{r_2} + \overline{c_1}.b_2.\overline{r_2})$	$\overline{\mathbf{a_1}}.\overline{b_1}.\overline{p_1}$	$\overline{a_2}.\overline{b_1}.\overline{p_2}$
⟶	$\mathbf{c_1}.(b_1.\overline{c_2}.\overline{r_1} + \overline{c_2}.b_1.\overline{r_1})$	$\overline{\mathbf{c_1}}.a_2.c_2.(b_2.\overline{c_1}.\overline{r_2} + \overline{c_1}.b_2.\overline{r_2})$	$\overline{b_1}.\overline{p_1}$	$\overline{a_2}.\overline{b_1}.\overline{p_2}$
⟶	$b_1.\overline{c_2}.\overline{r_1} + \overline{c_2}.b_1.\overline{r_1}$	$\mathbf{a_2}.c_2.(b_2.\overline{c_1}.\overline{r_2} + \overline{c_1}.b_2.\overline{r_2})$	$\overline{b_1}.\overline{p_1}$	$\overline{\mathbf{a_2}}.\overline{b_1}.\overline{p_2}$
⟶	$\mathbf{b_1}.\overline{c_2}.\overline{r_1} + \overline{c_2}.b_1.\overline{r_1}$	$c_2.(b_2.\overline{c_1}.\overline{r_2} + \overline{c_1}.b_2.\overline{r_2})$	$\overline{b_1}.\overline{p_1}$	$\overline{\mathbf{b_1}}.\overline{p_2}$
⟶	$\overline{\mathbf{c_2}}.\overline{r_1}$	$\mathbf{c_2}.(b_2.\overline{c_1}.\overline{r_2} + \overline{c_1}.b_2.\overline{r_2})$	$\overline{b_1}.\overline{p_1}$	$\overline{a_2}.\overline{b_1}.\overline{p_2}$
⟶	$a_1.c_1.(b_1.\overline{c_2}.\overline{r_1} + \overline{c_2}.b_1.\overline{r_1})$	$b_2.\overline{c_1}.\overline{r_2} + \overline{c_1}.b_2.\overline{r_2}$	$\overline{b_1}.\overline{p_1}$	$\overline{a_2}.\overline{b_1}.\overline{p_2}$

Figure 1 Stuck configuration, overlaid with backward slice with respect to final state of node 1

that the most liberal notion of reversibility is one that just respects causality: an action can be undone precisely after all the actions that causally depend on it have also been undone.

In this paper we give a formal account of dynamic slicing for π-calculus, and show that any causally equivalent execution generates precisely the same slicing information. We do this by formalising dynamic slicing with respect to a particular execution $\tilde{t}$, and then proving that slicing with respect to any causally equivalent computation $\tilde{u}$ yields the same slice, after a unique "rewiring" which interprets the path witnessing $\tilde{t} \simeq \tilde{u}$ as a lattice isomorphism relating the two slices. The isomorphism is constructive, rewriting one slice into the other: this allows non-deterministic metadata (e.g. memory addresses or transaction ids) in the slicing execution to be aligned with the corresponding metadata in the original run. To achieve this we build on an earlier "proof-relevant" formalisation of causal equivalence for π-calculus in Agda [17]. As long as causality is respected, an implementation of our system can safely use any technique (e.g. redex trails, proved transitions, or thread-local memories) to implement rewind and replay.

Example: scheduler with non-compliant task. While dynamic slicing cannot automatically isolate bugs, it can hide irrelevant detail and yield compact provenance-like explanations of troublesome parts of configurations. As an example we consider Milner's scheduler implementation [14, p. 65]. The scheduler controls the repeated performance of a set of n tasks, executed by agents $A_1, \ldots, A_n$. Agent A_i sends the message a_i (*announce*) to the scheduler to start its task, and message b_i (*break*) to end its task. The scheduler ensures that the actions a_i occur cyclically starting with a_1, and that for each i the actions a_i and b_i alternate, starting with a_i. Although started sequentially, once started the tasks are free to execute in parallel.

Figure 1 shows five transitions of a two-node scheduler, with the redex selected at each step highlighted in bold. The parts of the configuration which contribute to the final state of node 1 are in black; the grey parts are discarded by our backward-slicing algorithm. Assume prefixing binds more tightly than either $\cdot \mid \cdot$ or $+$. To save space, we omit the ν-binders defining the various names, and write $x.\mathbf{0}$ simply as x. The names r_1, r_2, p_1 and p_2 are used to make recursive calls [14, p. 94]: a recursive procedure is implemented as a server which waits for an invocation request, spawns a new copy of the procedure body, and then returns to the wait state. Here we omit the server definitions, and simply replace a successful invocation by the spawned body; thus in the final step of Figure 1, after the synchronisation on c_2 the invocation $\overline{r_1}$ is replaced by a fresh copy of the initial state of scheduler node 1.

The final state of Figure 1 has no redexes, and so is stuck. The slice helps highlight the fact that by the time we come to start the second loop of scheduler 1, the task was terminated by message $\overline{b_1}$ from A_2, before any such message could be sent by A_1. We can understand the slice of the initial configuration (computed by "rewinding", or backward-slicing) as *sufficient* to explain the slice of the stuck configuration by noting that the former is able to

compute the latter by "replay", or forward-slicing. In other words, writing a sliced part of the configuration as $\square$, and pretending the holes $\square$ are sub-computations which diverge, we can derive

$$\mathbf{a_1}.c_1.(b_1.\square + \square) \mid \overline{c_1}.a_2.\square \mid \overline{\mathbf{a_1}}.\square \mid \overline{a_2}.\overline{b_1}.\overline{p_2} \quad \longrightarrow^* \quad \overline{a_2}.\square$$

without diverging. The slice on the left may of course choose to take the right-hand branch of the choice instead. But if we constrain the replay of the sliced program to follow the causal structure of the original unsliced run – to take the same branches of internal choices, and have the same synchronisation structure – then it will indeed evolve to the slice on the right. This illustrates the correctness property for backward slicing, which is that forward-slicing its result must recompute (at least) the slicing criterion.

For this example, the tasks are entirely atomic and so fixing the outcome of + has the effect of making the computation completely sequential. Less trivial systems usually have multiple mays they can evolve, even once the causal structure is fixed. A confluence lemma typically formalises the observational equivalence of two causally equivalent runs. However, a key observation made in [17] is that requiring causally equivalent runs to reach exactly the same state is too restrictive for π-calculus, in particular because of name extrusion. As we discuss in § 3, two causally unrelated extrusion-rendezvous lead to states which differ in the relative position of two ν-binders, reflecting the two possible orderings of the rendezvous. Although technically unobservable to the program, interleaving-sensitive metadata, such as memory locations in a debugger or transaction ids in a financial application, may be important for domain-specific reasons. In these situations being able to robustly translate between the target states of the two executions may be useful.

Summary of contributions. § 2 defines the core forward and backward dynamic slicing operations for π-calculus transitions and sequences of transitions (traces). We prove that they are related by a Galois connection, showing that backward and forward slicing, as defined, are minimal and maximal with respect to each other. § 3 extends this framework to show that the Galois connections for causally equivalent traces compute the same slices up to lattice isomorphism. § 4 discusses related work and § 5 offers closing thoughts and prospects for follow-up work. Appendix A summarises the Agda module structure and required libraries; the source code can be found at `https://github.com/rolyp/concurrent-slicing`, release `0.1`.

2 Galois connections for slicing π-calculus programs

To summarise informally, our approach is to interpret, functorially, every transition diagram in the π-calculus into the category of lattices and Galois connections. For example the interpretation of the transition diagram on the left is the commutative diagram on the right:

where $\downarrow P$ means the lattice of slices of P, and $\overline{\mathsf{step}}_t : \downarrow P \longrightarrow \downarrow Q$ is a *Galois connection*, a kind of generalised order isomorphism. An order isomorphism between posets A and B is a pair of monotone functions $f : A \longrightarrow B$ and $g : B \longrightarrow A$ such that $f \circ g = \mathsf{id}_B$ and $g \circ f = \mathsf{id}_A$. Galois connections require only $f \circ g \geq \mathsf{id}_B$ and $g \circ f \leq \mathsf{id}_A$ where $\leq$ means the pointwise order. Galois connections are closed under composition.

The relationship to slicing is that these properties can be unpacked into statements of sufficiency and minimality: for example $f \circ g \geq \mathsf{id}_B$ means g (backward-slicing) is "sufficient" in that f (forward-slicing) is able to use the result of g to restore the slicing criterion, and $g \circ f \leq \mathsf{id}_A$ means g is "minimal" in that it computes the smallest slice with that property. One can dualise these statements to make similar observations about f.

We omit a treatment of structural congruence from our approach, but note that it slots easily into the framework, generating lattice isomorphisms in a manner similar to the "bound braid" relation $\rtimes$ discussed in § 3, Definition 12.

2.1 Lattices of slices

The syntax of names, processes and actions is given in Figure 2. Slices are represented syntactically, via the $\square$ notation introduced informally in § 1. Our formalisation employs de Bruijn indices [7], an approach with well-known strengths and weaknesses compared to other approaches to names such as higher-order abstract syntax or nominal calculi.

Name	x, y, z	$::=$	$\square$	erased	Process	P, Q, R, S	$::=$	$\square$	erased
			$0 \mid 1 \mid \cdots$					$\mathbf{0}$	inactive
Action	a	$::=$	$\square$	erased				$\underline{x}.P$	input
			$\underline{x}$	input				$\overline{x}\langle y\rangle.P$	output
			$\overline{x}\langle y\rangle$	output				$P + Q$	choice
			$\overline{x}$	bound output				$P \mid Q$	parallel
			τ	silent				νP	restriction
								$!P$	replication

Figure 2 Syntax of names, processes and actions

Names. The erased name $\square$ gives rise to a (trivial) partial order $\leq$ over names, namely the least partial order containing $\square \leq x$ for any x. The set of *slices* of x is written $\downarrow x$ and defined to be $\{y \mid y \leq x\}$; because names are atomic $\downarrow x$ is simply the two-element set $\{\square, x\}$. The set $\downarrow x$ is a finite lattice with meet and join operations $\sqcap$ and $\sqcup$, and top and bottom elements x and $\square$ respectively. The meet and join are related to the partial order by $x \leq y \iff x \sqcup y = y \iff x \sqcap y = x$. Lattices are closed under component-wise products, justifying the notation $\downarrow(x, y)$ for $\downarrow x \times \downarrow y$.

Processes. The $\leq$ relation and $\downarrow\cdot$ operation extend to processes, via names which may be $\square$, and a special undefined process also written $\square$. A slice of P is simply P with some sub-processes replaced by $\square$. The relation $\leq$ is the least compatible partial order which has $\square$ as least element; all process constructors both preserve and reflect $\leq$, so we assume an equivalent inductive definition of $\leq$ when convenient. A process has a closing context Γ enumerating its free variables; in the untyped de Bruijn setting Γ is just a natural number. Often it is convenient to conflate Γ with a set of that cardinality.

Actions. An action a labels a transition (Figure 3 below), and is either a *bound* or *non-bound*. A bound action b is of the form $\underline{x}$ or $\overline{x}$ and opens a process with respect to x, taking it from Γ to $\Gamma + 1$. A non-bound action c of the form $\overline{x}\langle y\rangle$ or τ and preserves the free variables of the process. The $\leq$ relation and $\downarrow\cdot$ operation extend to actions via $\square$ names, plus a special undefined action also written $\square$.

Renamings. In the lattice setting, a renaming $\rho : \Gamma \longrightarrow \Gamma'$ is any function from Γ to $\Gamma' \uplus \{\square\}$; we also allow σ to range over renamings. Renaming application $\rho^* P$ is extended with the equation $\rho^* \square = \square$. The $\leq$ relation and $\downarrow\cdot$ operation apply pointwise.

Labelled transition semantics. The late-style labelled transition semantics is given in

$$\frac{}{\underline{x}.P \xrightarrow{\underline{x}} P} \qquad \frac{}{\overline{x}\langle y\rangle.P \xrightarrow{\overline{x}\langle y\rangle} P} \qquad \frac{P \xrightarrow{a} R}{P + Q \xrightarrow{a} R} \qquad \frac{P \xrightarrow{c} R}{P \mid Q \xrightarrow{c} R \mid Q}$$

$$(*)\ \frac{P \xrightarrow{b} R}{P \mid Q \xrightarrow{b} R \mid \mathsf{push}^* Q} \qquad (\S)\ \frac{P \xrightarrow{\underline{x}} R \qquad Q \xrightarrow{\overline{x}\langle y\rangle} S}{P \mid Q \xrightarrow{\tau} (\mathsf{pop}\ y)^* R \mid S} \qquad \frac{P \xrightarrow{\overline{(x+1)}\langle 0\rangle} R}{\nu P \xrightarrow{\overline{x}} R}$$

$$\frac{P \xrightarrow{\underline{x}} R \qquad Q \xrightarrow{\overline{x}} S}{P \mid Q \xrightarrow{\tau} \nu(R \mid S)} \qquad (\dagger)\ \frac{P \xrightarrow{\mathsf{push}^* c} R}{\nu P \xrightarrow{c} \nu R} \qquad (\ddagger)\ \frac{P \xrightarrow{\mathsf{push}^* b} R}{\nu P \xrightarrow{b} \nu(\mathsf{swap}^* R)} \qquad \frac{P \mid !P \xrightarrow{a} R}{!P \xrightarrow{a} R}$$

Figure 3 Labelled transition relation $P \xrightarrow{a} R$ (symmetric variants omitted)

Figure 3, and is distinguished only by its adaptation to the de Bruijn setting. The primary reference for a de Bruijn formulation of π-calculus is [11]; the consequences such an approach are explored in some depth in [17]. One pleasing consequence of a de Bruijn approach is that the usual side-conditions associated with transition rules can be operationalised via renamings. We briefly explain this, along with other uses of renamings in the transition rules, and refer the interested reader to these earlier works for more details. Definition 1 defines the renamings used in Figure 3 and Definition 2 the application $\rho^* a$ of ρ to an action a.

▶ **Definition 1** (push, pop, and swap).

$$\begin{array}{lcl} \mathsf{push}_\Gamma & : & \Gamma \longrightarrow \Gamma + 1 \\ \mathsf{push}\ x & = & x + 1 \end{array} \qquad \begin{array}{lcl} \mathsf{pop}_\Gamma\ y & : & \Gamma + 1 \longrightarrow \Gamma \\ \mathsf{pop}\ y\ 0 & = & y \\ \mathsf{pop}\ y\ (x+1) & = & x \end{array} \qquad \begin{array}{lcl} \mathsf{swap}_\Gamma & : & \Gamma + 2 \longrightarrow \Gamma + 2 \\ \mathsf{swap}\ 0 & = & 1 \\ \mathsf{swap}\ 1 & = & 0 \\ \mathsf{swap}\ (x+2) & = & x + 2 \end{array}$$

▶ **Definition 2** (Action renaming). Define the following lifting of a renaming to actions.

$$\begin{array}{lcll} \cdot^* & & : & (\Gamma \longrightarrow \Gamma') \longrightarrow \mathsf{Action}\ \Gamma \longrightarrow \mathsf{Action}\ \Gamma' \\ \rho^* & \square & = & \square \\ \rho^* & \underline{x} & = & \underline{\rho x} \\ \rho^* & \overline{x} & = & \overline{\rho x} \\ \rho^* & \tau & = & \tau \\ \rho^* & \overline{x}\langle y\rangle & = & \overline{\rho x}\langle \rho y\rangle \end{array}$$

- push occurs in the transition rule which propagates a bound action through a parallel composition $P \mid Q$ (rule $(*)$ in Figure 3), and rewires Q so that the name 0 is reserved. The effect is to ensure that the binder being propagated by P is not free in Q.
- push also occurs in the rules which propagate an action through a ν-binder (rules $(\dagger)$ and $(\ddagger)$), where it is applied to the action being propagated using the function defined in Definition 2. This ensures the action does not mention the binder it is propagating through. The use of $\cdot + 1$ in the name extrusion rule can be interpreted similarly.
- pop y is used in the event of a successful synchronisation (rule $(\S)$), and undoes the effect of push, substituting the communicated name y for index 0.
- swap occurs in the rule which propagates a bound action through a ν-binder (rule $(\dagger)$) and has no counterpart outside of the de Bruijn setting. As a propagating binder passes through another binder, their relative position in the syntax is exchanged, and so to preserve naming R is rewired with a "braid" that swaps 0 and 1.

Although its use in the operational semantics is unique to the de Bruijn setting, swap will also play an important role when we consider the relationship between slices of causally

equivalent traces (§ 3 below), where it captures how the relative position of binders changes between different (but causally equivalent) interleavings.

2.2 Galois connections for slicing

We now compositionally assemble a Galois connection for each component of execution, starting with renamings, and then proceeding to individual transitions and entire traces, which relates forward and backward slices of the initial and terminal state.

Slicing renamings. The application ρx of a renaming to a name, and the lifting $\rho^* P$ of that operation to a process give rise to the Galois connections defined here.

▶ **Definition 3** (Galois connection for ρx)**.** Suppose $\rho : \Gamma \longrightarrow \Gamma'$ and $x \in \Gamma$. Define the following pair of monotone functions between $\downarrow(\rho, x)$ and $\downarrow(\rho x)$.

$$\begin{array}{llcl} \mathsf{app}_{\rho,x} & & : & \downarrow(\rho, x) \longrightarrow \downarrow(\rho x) \\ \mathsf{app}_{\rho,x} & (\sigma, \Box) & = & \Box \\ \mathsf{app}_{\rho,x} & (\sigma, x) & = & \sigma x \end{array} \qquad \begin{array}{llcl} \mathsf{unapp}_{\rho,x} & & : & \downarrow(\rho x) \longrightarrow \downarrow(\rho, x) \\ \mathsf{unapp}_{\rho,x} & u & = & (x \mapsto_\rho u, \rho_x^{-1} u) \end{array}$$

where

$$\begin{array}{llcll} x \mapsto_\rho \cdot & & : & \downarrow(\rho x) \longrightarrow \downarrow\rho & \\ (x \mapsto_\rho u) & x & = & u & \\ (x \mapsto_\rho u) & y & = & \Box & (\text{if } y \neq x) \end{array} \qquad \begin{array}{llcll} \rho_x^{-1} & & : & \downarrow(\rho x) \longrightarrow \downarrow x & \\ \rho_x^{-1} & \Box & = & \Box & \\ \rho_x^{-1} & u & = & x & (\text{if } u \neq \Box) \end{array}$$

It is convenient to decompose $\mathsf{unapp}_{\rho,x}$ into two components: $x \mapsto_\rho u$ denotes the least slice of ρ which maps x to u, and $\rho_x^{-1} u$ denotes the least slice of x such that $\rho x = u$.

▶ **Lemma 4.** *($\mathsf{app}_{\rho,x}$, $\mathsf{unapp}_{\rho,x}$) is a Galois connection.*

1. $\mathsf{app}_{\rho,x} \circ \mathsf{unapp}_{\rho,x} \geq \mathsf{id}_{\rho x}$
2. $\mathsf{unapp}_{\rho,x} \circ \mathsf{app}_{\rho,x} \leq \mathsf{id}_{\rho,x}$

▶ **Definition 5** (Galois connection for a renaming $\rho^* P$)**.**
Suppose $\rho : \Gamma \longrightarrow \Gamma'$ and $\Gamma \vdash P$. Define monotone functions between $\downarrow(\rho, P)$ and $\downarrow(\rho^* P)$ by structural recursion on $\downarrow P$, using the following equations. Here $\Box_\rho$ denotes the least slice of ρ, namely the renaming which maps every $x \in \Gamma$ to $\Box$.

$$\begin{array}{lll} \mathsf{ren}_{\rho,P} & : & \downarrow(\rho, P) \longrightarrow \downarrow(\rho^* P) \\ \mathsf{ren}_{\rho,P}\ (\sigma, \Box) & = & \Box \\ \mathsf{ren}_{\rho,\mathbf{0}}\ (\sigma, \mathbf{0}) & = & \mathbf{0} \\ \mathsf{ren}_{\rho,\underline{x}.P}\ (\sigma, \underline{u}.R) & = & \underline{x'}.\mathsf{ren}_{\rho+1,P}\ (\sigma, R) \ \textit{where}\ x' = \mathsf{app}_{\rho,x}(\sigma, u) \\ \mathsf{ren}_{\rho,\overline{x}\langle y\rangle.P}\ (\sigma, \overline{u}\langle v\rangle.R) & = & \overline{x'}\langle y'\rangle.\mathsf{ren}_{\rho,P}\ (\sigma, R) \ \textit{where}\ x' = \mathsf{app}_{\rho,x}(\sigma, u)\ \textit{and}\ y' = \mathsf{app}_{\rho,y}(\sigma, v) \\ \mathsf{ren}_{\rho,P+Q}\ (\sigma, R + S) & = & \mathsf{ren}_{\rho,P}\ (\sigma, R) + \mathsf{ren}_{\rho,Q}\ (\sigma, S) \\ \mathsf{ren}_{\rho,P|Q}\ (\sigma, R \mid S) & = & \mathsf{ren}_{\rho,P}\ (\sigma, R) \mid \mathsf{ren}_{\rho,Q}\ (\sigma, S) \\ \mathsf{ren}_{\rho,\nu P}\ (\sigma, \nu R) & = & \nu(\mathsf{ren}_{\rho+1,P}\ (\sigma + 1, R) \\ \mathsf{ren}_{\rho,!P}\ (\sigma, !R) & = & !(\mathsf{ren}_{\rho,P}\ (\sigma, R)) \end{array}$$

$$\begin{array}{lll} \mathsf{unren}_{\rho,P} & : & \downarrow(\rho^* P) \longrightarrow \downarrow(\rho, P) \\ \mathsf{unren}_{\rho,P}\ \Box & = & (\Box_\rho, \Box) \\ \mathsf{unren}_{\rho,\mathbf{0}}\ \mathbf{0} & = & (\Box_\rho, \mathbf{0}) \\ \mathsf{unren}_{\rho,\underline{x}.P}\ \underline{u}.R & = & (\rho' \sqcup (x \mapsto_\rho u), \underline{\rho_x^{-1} u}.P') \ \textit{where}\ \mathsf{unren}_{\rho+1,P}\ R = (\rho' + 1, P') \\ \mathsf{unren}_{\rho,\overline{x}\langle y\rangle.P}\ \overline{u}\langle v\rangle.R & = & (\rho' \sqcup (x \mapsto_\rho u) \sqcup (y \mapsto_\rho v), \overline{\rho_x^{-1} u}\langle \rho_y^{-1} v\rangle.P') \ \textit{where}\ \mathsf{unren}_{\rho,P}\ R = (\rho', P') \\ \mathsf{unren}_{\rho,P+Q}\ (R + S) & = & (\rho_1 \sqcup \rho_2, P' + Q') \ \textit{where}\ \mathsf{unren}_{\rho,P}\ R = (\rho_1, P')\ \text{and}\ \mathsf{unren}_{\rho,Q}\ S = (\rho_2, Q') \\ \mathsf{unren}_{\rho,P|Q}\ (R \mid S) & = & (\rho_1 \sqcup \rho_2, P' \mid Q') \ \textit{where}\ \mathsf{unren}_{\rho,P}\ R = (\rho_1, P')\ \text{and}\ \mathsf{unren}_{\rho,Q}\ S = (\rho_2, Q') \\ \mathsf{unren}_{\rho,\nu P}\ \nu R & = & (\rho', \nu P') \ \textit{where}\ \mathsf{unren}_{\rho+1,P}\ R = (\rho' + 1, P') \\ \mathsf{unren}_{\rho,!P}\ !R & = & (\rho', !P') \ \textit{where}\ \mathsf{unren}_{\rho,P}\ R = (\rho', P') \end{array}$$

▶ **Lemma 6.** *($\mathsf{ren}_{\rho,P}$, $\mathsf{unren}_{\rho,P}$) is a Galois connection.*

$$\frac{}{\Box_P \xrightarrow{\Box_a} \Box_{P'}} \qquad \frac{}{\underline{u_x}.R_P \xrightarrow{u_x} R_P} \qquad \frac{}{\overline{u_x}\langle v_y\rangle.R_P \xrightarrow{\overline{u_x}\langle v_y\rangle} R_P} \qquad \frac{R_P \xrightarrow{a'_a} R'_{P'}}{R_P + S_Q \xrightarrow{a'_a} R'_{P'}}$$

$$\frac{R_P \xrightarrow{c'_c} R'_{P'}}{R_P \mid S_Q \xrightarrow{c'_c} R'_{P'} \mid S_Q} \qquad \frac{R_P \xrightarrow{b'_b} R'_{P'}}{R_P \mid S_Q \xrightarrow{b'_b} R'_{P'} \mid \mathsf{push}^* S_Q} \qquad \frac{R_P \xrightarrow{\underline{x}} R'_{P'} \qquad S_Q \xrightarrow{\overline{x}\langle y_z\rangle} S'_{Q'}}{R_P \mid S_Q \xrightarrow{\tau} (\mathsf{pop}\ y_z)^* R'_{P'} \mid S'_{Q'}}$$

$$\frac{R_P \xrightarrow{a_{\underline{x}}} R'_{P'} \qquad S_Q \xrightarrow{a'_{\overline{x}\langle z\rangle}} S'_{Q'}}{R_P \mid S_Q \xrightarrow{\Box_\tau} \Box_{(\mathsf{pop}\ z)^* P' \mid Q'}} \qquad \frac{R_P \xrightarrow{\overline{(x+1)}\langle 0\rangle} R'_{P'}}{\nu R_P \xrightarrow{\overline{x}} R'_{P'}} \qquad \frac{R_P \xrightarrow{a_{\overline{(x+1)}\langle 0\rangle}} R'_{P'}}{\nu R_P \xrightarrow{\Box_{\overline{x}}} \Box_{P'}}$$

$$\frac{R_P \xrightarrow{\underline{x}} R'_{P'} \qquad S_Q \xrightarrow{\overline{x}} S'_{Q'}}{R_P \mid S_Q \xrightarrow{\tau} \nu(R'_{P'} \mid S'_{Q'})} \qquad \frac{R_P \xrightarrow{a_{\underline{x}}} R'_{P'} \qquad S_Q \xrightarrow{a'_{\overline{x}}} S'_{Q'}}{R_P \mid S_Q \xrightarrow{\Box_\tau} \Box_{\nu(P' \mid Q')}} \qquad \frac{R_P \xrightarrow{\mathsf{push}^* c'_c} R'_{P'}}{\nu R_P \xrightarrow{c'_c} \nu R'_{P'}}$$

$$\frac{R_P \xrightarrow{c'_{\mathsf{push}^* c}} R'_{P'}}{\nu R_P \xrightarrow{\Box_c} \Box_{\nu P'}} \qquad \frac{R_P \xrightarrow{\mathsf{push}^* b'_b} R'_{P'}}{\nu R_P \xrightarrow{b'_b} \nu(\mathsf{swap}^* R'_{P'})} \qquad \frac{R_P \xrightarrow{b'_{\mathsf{push}^* b}} R'_{P'}}{\nu R_P \xrightarrow{\Box_b} \Box_{\nu(\mathsf{swap}^* P')}} \qquad \frac{R_P \mid !R_P \xrightarrow{a'_a} R'_{P'}}{!R_P \xrightarrow{a'_a} R'_{P'}}$$

Figure 4 Forward slicing judgement $R_P \xrightarrow{a'_a} R'_{P'}$

1. $ren_{\rho,P} \circ unren_{\rho,P} \geq id_{\rho^* P}$
2. $unren_{\rho,P} \circ ren_{\rho,P} \leq id_{\rho,P}$

Proof. In each case by induction on P, using Lemma 4 and the invertibility of $\cdot + 1$. ◀

Slicing transitions. Transitions also lift to the lattice setting, in the form of Galois connections defined by structural recursion over the proof that $t : P \xrightarrow{a} P'$. Figures 4 and 5 define the forward and backward slicing judgements. As presented the rules are non-deterministic, but the reader should consider there to be implicit determinising side-conditions which prefer earlier rules over subsequent rules.

The judgement $R_P \xrightarrow{a'_a} R'_{P'}$ asserts that there is a "replay" transition from $R \leq P$ to $(a', R') \leq (a, P)$. In this case R is the input to the judgement and (a', R') is the output. The judgement $R'_P \xleftarrow{a'_a} R_{P'}$ asserts that there is an "rewind" transition from $(a', R) \leq (a, P')$ to $R' \leq P$. In this case (a', R) is the input and R' the output. When writing R_P where $R \leq P$ we exploit the preservation and reflection of $\leq$ by all constructors in the notation, writing $\nu(R_P \mid S_Q)$ for $\nu(R \mid S)_{\nu(P|Q)}$, for example.

As a further notational convenience, we permit the renaming application operator * to appear in a pattern-matching (input) position of the backward-slicing judgement, indicating a use of the lower adjoint unren. Specifically, given a renaming application $\rho^* P$, the pattern $\sigma^* P'$ matches any slice R of $\rho^* P$ such that $\mathsf{unren}_{\rho,P}(R) = (\sigma, P')$.

▶ **Definition 7** (Galois connection for a transition)**.** Suppose $t : P \xrightarrow{a} P'$. Define the following pair of monotone functions between $\downarrow P$ to $\downarrow(a, P')$.

$$\begin{array}{lcl} \mathsf{step}_t & : & \downarrow P \longrightarrow \downarrow(a, P') \\ \mathsf{step}_t\ R & = & (a', R')\ \textit{where}\ R_P \xrightarrow{a'_a} R'_{P'} \end{array} \qquad \begin{array}{lcl} \mathsf{unstep}_t & : & \downarrow(a, P') \longrightarrow \downarrow P \\ \mathsf{unstep}_t\ (R, a') & = & R'\ \textit{where}\ R'_P \xleftarrow{a'_a} R_{P'} \end{array}$$

We omit the proofs that these equations indeed define total, deterministic, monotone relations.

▶ **Theorem 8** ($(\mathsf{step}_t, \mathsf{unstep}_t)$ is a Galois connection)**.**

1. $step_t \circ unstep_t \geq id_{a,P'}$
2. $unstep_t \circ step_t \leq id_P$

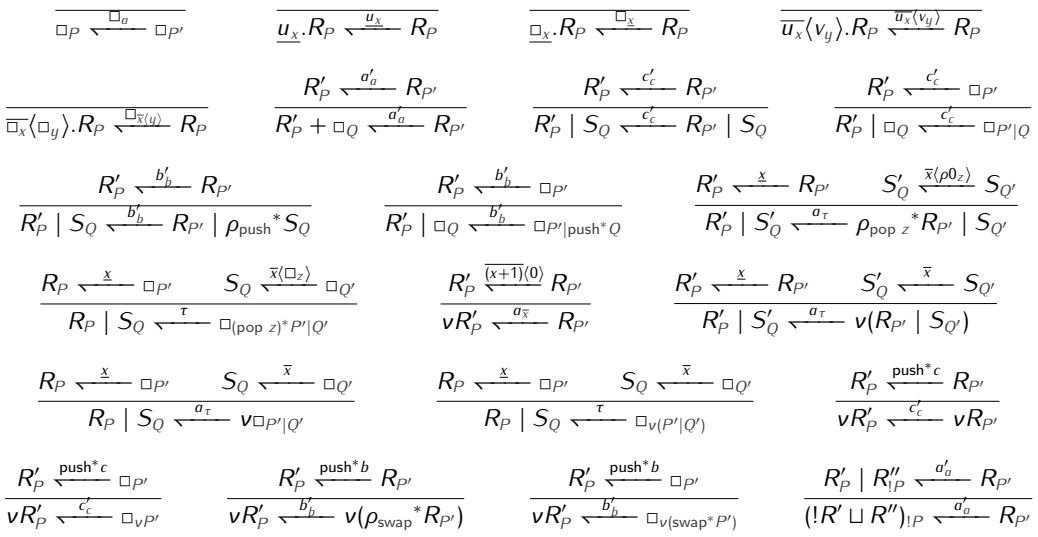


Figure 5 Backward slicing judgement $R'_P \xleftarrow{a'_a} R_{P'}$

Proof. By induction on $t : P \xrightarrow{a'} P'$, using Lemma 6 for the cases involving renaming. ◀

Slicing traces. Finally we extend slicing to entire runs of a π-calculus program. A sequence of transitions $\tilde{t}$ is called a *trace*; the empty trace at P is written ε_P, and the composition of a transition $t : P \xrightarrow{a} R$ and trace $\tilde{t} : R \xrightarrow{\tilde{a}} S$ is written $t \cdot \tilde{t} : P \xrightarrow{a \cdot \tilde{a}} S$ where actions are composable whenever their source and target contexts match.

▶ **Definition 9** (Galois connection for a trace $\tilde{t}$)**.** Suppose $\tilde{t} : P \xrightarrow{\tilde{a}} P'$. Define the following pair of monotone functions between $\downarrow P$ and $\downarrow P'$, which use variants of step_t and unstep_t which discard the action slice (going forward) and which use $\square$ as the action slice (going backward).

$$
\begin{array}{lcll}
\mathsf{fwd}_{\tilde{t}} & : & \downarrow P \longrightarrow \downarrow P' & \\
\mathsf{fwd}_{\varepsilon_P} & = & \mathsf{id}_{\downarrow P} & \\
\mathsf{fwd}_{t \cdot \tilde{t}}\ \square & = & \square & \\
\mathsf{fwd}_{t \cdot \tilde{t}}\ R & = & \mathsf{fwd}_{\tilde{t}}\ (\mathsf{step}'_t\ R) & (R \neq \square)
\end{array}
\qquad
\begin{array}{lcll}
\mathsf{bwd}_{\tilde{t}} & : & \downarrow P' \longrightarrow \downarrow P & \\
\mathsf{bwd}_{\varepsilon_{P'}} & = & \mathsf{id}_{\downarrow P'} & \\
\mathsf{bwd}_{t \cdot \tilde{t}}\ \square & = & \square & \\
\mathsf{bwd}_{t \cdot \tilde{t}}\ R & = & \mathsf{unstep}'_t\ (\mathsf{bwd}_{\tilde{t}}\ R) & (R \neq \square)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{step}'_t & : & \downarrow \mathbf{P} \longrightarrow \downarrow \mathbf{P}' \\
\mathsf{step}'_t\ R & = & R'\ \textit{where}\ \mathsf{step}_t\ R = (a', R')
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{unstep}'_t & : & \downarrow P' \longrightarrow \downarrow P \\
\mathsf{unstep}'_t\ R' & = & \mathsf{unstep}_t\ (\square, R')
\end{array}
$$

At the empty trace ε_P the Galois connection is simply the identity on $\downarrow P$. Otherwise, we recurse into the structure of the trace $t \cdot \tilde{t}$, composing the Galois connection for the single transition t with the Galois connection for the tail of the computation $\tilde{t}$.

▶ **Theorem 10** ($(\mathsf{fwd}_{\tilde{t}}, \mathsf{bwd}_{\tilde{t}})$ is a Galois connection)**.**

1. $fwd_{\tilde{t}} \circ bwd_{\tilde{t}} \geq id_{P'}$
2. $bwd_{\tilde{t}} \circ fwd_{\tilde{t}} \leq id_P$

Note that the trace used to define forward and backward slicing for a computation is not an auxiliary data structure recording the computation, such as a redex trail or memory, but simply the proof term witnessing $P \xrightarrow{\tilde{a}} P'$.

3 Slicing and causal equivalence

In this section, we show that when dynamic slicing a π-calculus program, slicing with respect to any causally equivalent execution yields essentially the same slice. "Essentially the same" here means modulo lattice isomorphism. In other words slicing discards the same quantity of information regardless of which interleaving is chosen to do the slicing.

Proof-relevant causal equivalence. Causally equivalent computations are generated by transitions which share a start state, but which are independent. Following Lévy [13], we call such transitions *concurrent*, written $t \frown t'$. We illustrate this idea, and the non-trivial relationship that it induces between terminal states, by way of example. For the full definition of concurrency for π-calculus, we refer the interested reader to [17] or to the Agda definition[1]. For the sake of familiarity the example uses regular names instead of de Bruijn indices.

Example. Consider the process $P_0 \stackrel{\text{def}}{=} (\nu yz)\ (\overline{x}\langle y\rangle.P) \mid \overline{x}\langle z\rangle.Q$ for some unspecified processes P and Q. This process can take *two* transitions, which we will call t and t'. Transition $t : P_0 \xrightarrow{\overline{x}(y)} P_1$ extrudes y on the channel x:

$$P_0 \xrightarrow{\overline{x}(y)} (\nu z)\ P \mid \overline{x}\langle z\rangle.Q \stackrel{\text{def}}{=} P_1$$

whereas transition $t' : P_0 \xrightarrow{\overline{x}(z)} P'_1$ extrudes z, also on the channel x:

$$P_0 \xrightarrow{\overline{x}(z)} (\nu y)\ (\overline{x}\langle y\rangle.P) \mid Q \stackrel{\text{def}}{=} P'_1$$

In both cases the output actions are bound, representing the extruding binder. Moreover, t and t' are *concurrent*, written $t \frown t'$, meaning they can be executed in either order. Having taken t, one can *mutatis mutandis* take t', and vice versa. Concurrency is an irreflexive and symmetric relation defined over transitions which are *coinitial* (have the same source state).

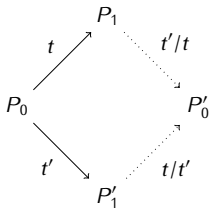
The qualification is needed because t' will need to be adjusted to operate on the target state of t, if t is the transition which happens first. If t' happens first then t will need to be adjusted to operate on the target state of t'. The adjusted version of t' is called the *residual* of t' after t, and is written t'/t. In this case t'/t can still extrude z:

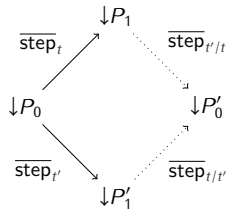
$$P_1 = (\nu z)\ P \mid \overline{x}\langle z\rangle.Q \xrightarrow{\overline{x}(z)} P \mid Q \stackrel{\text{def}}{=} P'_0$$

whereas the residual t/t' can still extrude y:

$$P'_1 = (\nu y)\ (\overline{x}\langle y\rangle.P) \mid Q \xrightarrow{\overline{x}(y)} P \mid Q = P'_0$$

The independence of t and t' is confirmed by the fact that $t \cdot t'/t$ and $t' \cdot t/t'$ are *cofinal* (share a target state), as shown on the left below.





We say that the traces $\tilde{t} \stackrel{\text{def}}{=} t \cdot t'/t$ and $\tilde{u} \stackrel{\text{def}}{=} t' \cdot t/t'$ are *causally equivalent*, written $\tilde{t} \simeq \tilde{u}$. The two interleavings are also equivalent from a slicing point of view, in the sense that the

[1] https://github.com/rolyp/proof-relevant-pi/blob/master/Transition/Concur.agda

right-hand square commutes (Theorem 16 below). Here $\overline{\mathsf{step}}_t$ denotes the Galois connection $(\mathsf{step}_t, \mathsf{unstep}_t)$.

However [17], which formalised causal equivalence for π-calculus, showed that causally equivalence traces do not always reach exactly the same state, but only the same state up to some permutation of the binders in the resulting processes. This will become clear if we consider another process $Q_0 \stackrel{\text{def}}{=} (x(y').R) \mid x(z').S$ able to synchronise with both of the extrusions raised by P_0 and consider the two different ways that $P_0 \mid Q_0$ can evolve.

First note that Q_0 can also take two independent transitions: $u : Q_0 \xrightarrow{x(y')} R \mid x(z').S \stackrel{\text{def}}{=} Q_1$ inputs on x and binds the received name to y'; and $u' : Q_0 \xrightarrow{x(z')} (x(y').R) \mid S \stackrel{\text{def}}{=} Q_1'$ also inputs on x and binds the received name to z'. (Assume z is not free in the left-hand side of Q_0 and that y is not free in the right-hand side.) The respective residuals $Q_1 = R \mid x(z').S \xrightarrow{x(z')} R \mid S \stackrel{\text{def}}{=} Q_0'$ and $Q_1' = (x(y').R) \mid S \xrightarrow{x(y')} R \mid S = Q_0'$ again converge on the same state Q_0', leading to a diamond for Q_0 similar to the one for P_0 above.

The subtlety arises when we put P_0 and Q_0 into parallel composition, since now we have two concurrent synchronisation possibilities. For clarity we give the derivations, which we call s and s':

$$\frac{t : P_0 \xrightarrow{\overline{x}(y)} P_1 \qquad u : Q_0 \xrightarrow{x(y')} Q_1}{s : P_0 \mid Q_0 \xrightarrow{\tau} (\nu y)\, P_1 \mid Q_1\{y/y'\}} \qquad \frac{t' : P_0 \xrightarrow{\overline{x}(z)} P_1' \qquad u' : Q_0 \xrightarrow{x(z')} Q_1'}{s' : P_0 \mid Q_0 \xrightarrow{\tau} (\nu z)\, P_1' \mid Q_1'\{z/z'\}}$$

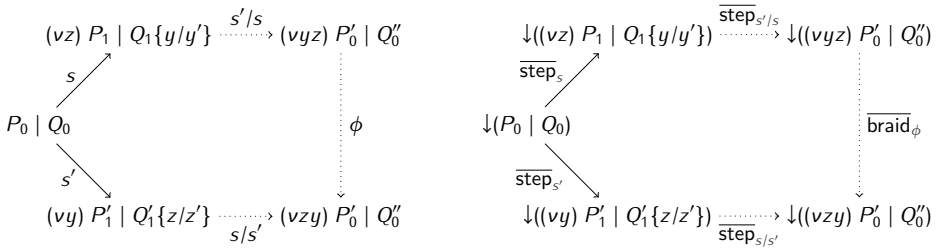
The labelled transition system is closed under renamings; thus the residual u'/u has an image in the renaming $\cdot\{y/y'\}$, and u/u' has an image in the renaming $\cdot\{z/z'\}$, allowing us to derive composite residual s'/s:

$$\frac{t'/t : P_1 \xrightarrow{\overline{x}(z)} P_0' \qquad \dfrac{u'/u : Q_1 \xrightarrow{x(z')} Q_0'}{(u'/u)\{y/y'\} : Q_1\{y/y'\} \xrightarrow{x(z')} Q_0'\{y/y'\}}}{s'/s : (\nu y)\, P_1 \mid Q_1\{y/y'\} \xrightarrow{\tau} (\nu yz)\, P_0' \mid Q_0'\{y/y'\}\{z/z'\}}$$

By similar reasoning we can derive s/s':

$$s/s' : (\nu z)\, P_1' \mid Q_1'\{y/y'\} \xrightarrow{\tau} (\nu zy)\, P_0' \mid Q_0'\{z/z'\}\{y/y'\}$$

By side-conditions on the transition rules the renamings $\cdot\{y/y'\}$ and $\cdot\{z/z'\}$ commute and so $Q_0'\{y/y'\}\{z/z'\} \stackrel{\text{def}}{=} Q_0'' = Q_0'\{z/z'\}\{y/y'\}$. However, the positions of binders y and z are transposed in the terminal states of s'/s and s/s'. Instead of the usual diamond shape, we have the pentagon on the left below, where ϕ is a *braid* representing the transposition of the binders. Lifted to slices, ϕ becomes the unique isomorphism $\overline{\mathsf{braid}}_\phi$ relating slices of the terminal states, as shown in the commutative diagram on the right:



In the de Bruijn setting, a braid like ϕ does not relate two processes of the form $(\nu yz)\, R$ and $(\nu yz)\, R$ but rather two processes of the form $\nu\nu R$ and $\nu\nu(\mathsf{swap}^* R)$: the transposition of the (nameless) binders is represented by the transposition of the roles of indices 0 and 1 in the body of the innermost binder.

▶ **Definition 11** (Bound braid $P \rtimes R$)**.** Inductively define the symmetric relation $P \rtimes R$ using the rules below.

$$\nu\nu\text{-swap}_P \ \frac{}{\nu\nu P \rtimes \nu\nu P'} \ P = \mathsf{swap}^* P' \qquad \cdot + Q \ \frac{P \rtimes R}{P + Q \rtimes R + Q} \qquad P + \cdot \ \frac{Q \rtimes S}{P + Q \rtimes P + S}$$

$$\cdot \mid Q \ \frac{P \rtimes R}{P \mid Q \rtimes R \mid Q} \qquad P \mid \cdot \ \frac{Q \rtimes S}{P \mid Q \rtimes P \mid S} \qquad \nu\cdot \ \frac{P \rtimes R}{\nu P \rtimes \nu R} \qquad !\cdot \ \frac{P \rtimes R}{!P \rtimes !R}$$

Following [17], we adopt a compact term-like notation for $\rtimes$ proofs, using the rule names which occur to the left of each rule in Definition 11. For the extrusion example above, ϕ (in de Bruijn indices notation) would be a leaf case of the form $\nu\nu\text{-swap}_{\cdot\mid\cdot}$.

▶ **Definition 12** (Lattice isomorphism for bound braid)**.** Suppose $\phi : Q \rtimes Q'$. Define the following pair of monotone functions between $\downarrow Q$ and $\downarrow Q'$ by structural recursion on ϕ.

braid_ϕ		:	$\downarrow Q \longrightarrow \downarrow Q'$	$\mathsf{unbraid}_\phi$		:	$\downarrow Q' \longrightarrow \downarrow Q$
$\mathsf{braid}_{\nu\nu\text{-swap}_P}$	$(\nu\nu R)$	=	$\nu\nu(\mathsf{ren}_{\mathsf{swap},P}(R))$	$\mathsf{unbraid}_{\nu\nu\text{-swap}_P}$	$(\nu\nu R)$	=	$\nu\nu(\mathsf{ren}_{\mathsf{swap},P}(R))$
$\mathsf{braid}_{\phi+S}$	$(R+S)$	=	$\mathsf{braid}_\phi\ R + S$	$\mathsf{unbraid}_{\phi+S}$	$(R+S)$	=	$\mathsf{unbraid}_\phi\ R + S$
$\mathsf{braid}_{R+\psi}$	$(R+S)$	=	$R + \mathsf{braid}_\psi\ S$	$\mathsf{unbraid}_{R+\psi}$	$(R+S)$	=	$R + \mathsf{unbraid}_\psi\ S$
$\mathsf{braid}_{\phi\mid S}$	$(R \mid S)$	=	$\mathsf{braid}_\phi\ R \mid S$	$\mathsf{unbraid}_{\phi\mid S}$	$(R \mid S)$	=	$\mathsf{unbraid}_\phi\ R \mid S$
$\mathsf{braid}_{R\mid\psi}$	$(R \mid S)$	=	$R \mid \mathsf{braid}_\psi\ S$	$\mathsf{unbraid}_{R\mid\psi}$	$(R \mid S)$	=	$R \mid \mathsf{unbraid}_\psi\ S$
$\mathsf{braid}_{\nu\phi}$	(νR)	=	$\nu(\mathsf{braid}_\phi\ R)$	$\mathsf{unbraid}_{\nu\phi}$	(νR)	=	$\nu(\mathsf{unbraid}_\phi\ R)$
$\mathsf{braid}_{!\phi}$	$(!R)$	=	$!(\mathsf{braid}_\phi\ R)$	$\mathsf{unbraid}_{!\phi}$	$(!R)$	=	$!(\mathsf{unbraid}_\phi\ R)$

▶ **Lemma 13.**

1. $\mathit{braid}_\phi \circ \mathit{unbraid}_\phi = \mathit{id}_{\downarrow Q'}$
2. $\mathit{unbraid}_\phi \circ \mathit{braid}_\phi = \mathit{id}_{\downarrow Q}$

Proof. Induction on ϕ. In the base case use the idempotence of swap lifted to lattices. ◀

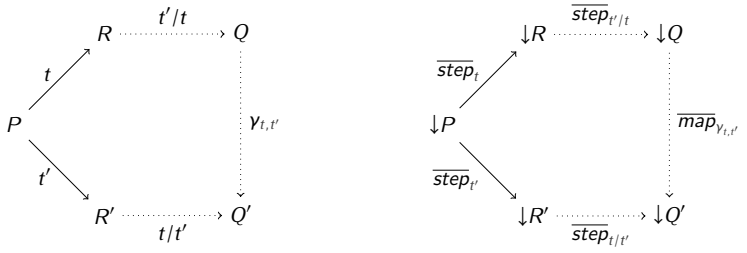
▶ **Definition 14** (Lattice isomorphism for cofinality map)**.** Suppose $t \smile t'$ with $\mathsf{tgt}(t'/t) = Q$ and $\mathsf{tgt}(t/t') = Q'$. By Theorem 1 of [17], there exists a unique $\gamma_{t,t'}$ witnessing $Q = Q'$, $Q \asymp Q'$ or $Q \rtimes Q'$. Define the following pair of monotone functions between $\downarrow Q$ and $\downarrow Q'$.

$\mathsf{map}_{\gamma_{t,t'}}$	:	$\downarrow Q \longrightarrow \downarrow Q'$	$\mathsf{unmap}_{\gamma_{t,t'}}$	:	$\downarrow Q' \longrightarrow \downarrow Q$
$\mathsf{map}_{Q=Q'}$	=	$\mathsf{id}_{\downarrow Q}$	$\mathsf{unmap}_{Q=Q'}$	=	$\mathsf{id}_{\downarrow Q}$
$\mathsf{map}_{Q \asymp Q'}$	=	$\mathsf{ren}_{\mathsf{swap},Q}$	$\mathsf{unmap}_{Q \asymp Q'}$	=	$\mathsf{unren}_{\mathsf{swap},Q}$
$\mathsf{map}_{\phi:Q \rtimes Q'}$	=	braid_ϕ	$\mathsf{unmap}_{\phi:Q \rtimes Q'}$	=	$\mathsf{unbraid}_\phi$

▶ **Lemma 15.**

1. $\mathit{map}_{\gamma_{t,t'}} \circ \mathit{unmap}_{\gamma_{t,t'}} = \mathit{id}_{\downarrow Q'}$
2. $\mathit{unmap}_{\gamma_{t,t'}} \circ \mathit{map}_{\gamma_{t,t'}} = \mathit{id}_{\downarrow Q}$

▶ **Theorem 16.** *Suppose $t \smile t'$ as on the left. Then the pentagon on the right commutes.*



Lattice isomorphism for arbitrary causal equivalence. Concurrent transitions $t \smile t'$ induce an "atom" of causal equivalence, $t \cdot t'/t \simeq t' \cdot t/t'$. The full relation is generated by closing under the trace constructors (for horizontal composition) and transitivity (for vertical composition). In [17] this yields a composite form of cofinality map γ_α where $\alpha : \tilde{t} \simeq \tilde{u}$ is an arbitrary causal equivalence. We omit further discussion for reasons of space, but note that γ_α is built by composing and translating (by contexts) atomic cofinality maps, and so gives rise, by composition of isomorphisms, to a lattice isomorphism between $\downarrow\mathsf{tgt}(\tilde{t})$ and $\downarrow\mathsf{tgt}(\tilde{u})$.

4 Related work

Reversible process calculi. Reversible process calculi have recently been used for speculative execution, debugging, transactions, and distributed protocols that require backtracking. A key challenge is to permit backwards execution to leverage concurrency whilst ensuring causal consistency. In contrast to our work, reversible calculi focus on mechanisms for reversibility, such as the thread-local memories used by Danos and Krivine's reversible CCS [6], Lanese et al's $\rho\pi$ [12], and Cristescu et al's reversible π-calculus [5]. We intentionally remain agnostic about implementation strategy, whilst providing a formal guarantee that causally consistent rewind and replay are suitable foundation for any implementation.

Concurrent program slicing. An early example of concurrent dynamic slicing is Duesterwald et al, who consider a language with synchronous message-passing [9]. They give a notion of correctness with respect to a slicing criterion, but find that computing least slices is undecidable. Following Cheng's [4], most subsequent work has recast dynamic slicing as a dependency-graph reachability problem; we take a different approach, namely to slice with respect to a particular execution but show how to map the result to the slice that would be obtained from another execution with the same dependency structure.

Moreover, there is a notable lack of correctness and minimality properties for systems for concurrent dynamic slicing. Goswami and Mall consider a language with shared-memory concurrency [10], and Mohapatra et al tackle slicing for concurrent Java [15], but both present only algorithms, with no formal guarantees. Tallam et al develop an approach based on dependency graphs, but again offer only algorithms and empirical results [19]. Finally, most prior work restricts the slicing criteria to the (entire) values of particular variables, rather than arbitrary parts of configurations.

Provenance and slicing. Our interest in slicing [16, 3] arises in part due to connections with provenance and dependency-tracking [2], and recent applications of provenance to security [1]. Others have also considered provenance models in concurrency calculi, including Souliah et al [18] and Dezani-Ciancaglini et al [8]. We have has focused on formalising slicing for π-calculus using Galois connections; further study is needed to relate this model to provenance and security.

5 Conclusion

The main contribution of this paper is to extend our previous approach to slicing based on Galois connections to a concurrent setting, and show that the resulting notion of slicing for the π-calculus is invariant under causal equivalence of traces. Because of the complexity of the underlying definitions, we have built upon a previous formalization of causal equivalence for the π-calculus in Agda [17]. Although this makes the resulting definitions less transparent (as usual with de Bruijn indices), this is a small price to pay for definitions that are correct by construction. This paper provides a foundation for future development of rigorous provenance

tracing or dynamic slicing techniques for practical concurrent programs, which we plan to investigate in future work.

References

1. Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. *Journal of Computer Security*, 21:919–969, 2013. Full version of a POST 2012 paper.
2. James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.
3. James Cheney, Amal Ahmed, and Umut A. Acar. Database queries that explain their work. In *PPDP 2014*, pages 271–282, 2014.
4. Jingde Cheng. Slicing concurrent programs: A graph-theoretical approach. In *Automated and Algorithmic Debugging*, number 749 in LNCS, pages 223–240. Springer-Verlag, 1993.
5. Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible pi-calculus. In *LICS*, pages 388–397, June 2013.
6. Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004. doi:10.1007/978-3-540-28644-8_19.
7. N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.
8. Mariangiola Dezani-Ciancaglini, Ross Horne, and Vladimiro Sassone. Tracing where and who provenance in linked data: A calculus. *Theor. Comput. Sci.*, 464:113–129, 2012.
9. Evelyn Duesterwald, Rajiv Gupta, and Mary Lou 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.
10. D. Goswami and R. Mall. Dynamic slicing of concurrent programs. In Mateo Valero, Viktor K. Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing – HiPC 2000*, volume 1970 of *LNCS*, pages 15–26. Springer, Berlin / Heidelberg, 2000. doi:10.1007/3-540-44467-X_2.
11. Daniel Hirschkoff. Handling substitutions explicitly in the pi-calculus. In *Proceedings of the Second International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs*, 1999.
12. Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversing higher-order pi. In *CONCUR*, pages 478–493. Springer-Verlag, 2010.
13. Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, New York, NY, USA, 1980.
14. Robin Milner. *Communicating and mobile systems: the π calculus*. Cambridge University Press, Cambridge, UK, 1999.
15. D.P. Mohapatra, Rajib Mall, and Rajeev Kumar. An efficient technique for dynamic slicing of concurrent Java programs. In Suresh Manandhar, Jim Austin, Uday Desai, Yoshio Oyanagi, and AsokeK. Talukder, editors, *Applied Computing*, volume 3285 of *LNCS*, pages 255–262. Springer, Berlin / Heidelberg, 2004. doi:10.1007/978-3-540-30176-9_33.
16. Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *ICFP*, pages 365–376, New York, NY, USA, 2012. ACM. doi:10.1145/2364527.2364579.
17. Roly Perera and James Cheney. Proof-relevant pi-calculus, 2016. Submitted. http://arxiv.org/abs/1604.04575.

18 Issam Souilah, Adrian Francalanza, and Vladimiro Sassone. A formal model of provenance in distributed systems. In *TAPP 2009*, Berkeley, CA, USA, 2009. USENIX Association.

19 Sriraman Tallam, Chen Tian, and Rajiv 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. doi:10.1109/ICSM.2008.4658058.

20 Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

A Agda module structure

Figure 6 summarises the module structure of the Agda formalisation. The module structure of the auxiliary libraries is described in [17]. All repositories can be found at the URL `https://github.com/rolyp`.

Auxiliary libraries	
`agda-stdlib-ext 0.0.3`	Extensions to Agda library
`proof-relevant-pi 0.3`	Concurrent transitions, residuals and causal equivalence
Core modules	
`Action.Lattice`	Action slices $a' \in \downarrow a$
`Braiding.Proc.Lattice`	Bound braid prefixes and slicing functions braid_ϕ and $\mathsf{unbraid}_\phi$
`ConcurrentSlicing`	Include everything; compile to build project
`ConcurrentSlicingCommon`	Common imports from standard library
`Example`	Milner's scheduler example
`Example.Helper`	Utility functions for examples
`Lattice`	Lattice typeclass
`Lattice.Product`	Component-wise product of lattices
`Name.Lattice`	Name slices $y \in \downarrow x$
`Proc.Lattice`	Process slices $P' \in \downarrow P$
`Proc.Ren.Lattice`	Slicing functions $\mathsf{ren}_{\rho,P}$ and $\mathsf{unren}_{\rho,P}$
`Ren.Lattice`	Renaming slices $\sigma \in \downarrow \rho$ and slicing functions $\mathsf{app}_{\rho,x}$ and $\mathsf{unapp}_{\rho,x}$
`Ren.Lattice.Properties`	Additional properties relating to renaming slices
`Transition.Lattice`	Slicing functions step_t and unstep_{gt}
`Transition.Seq.Lattice`	Slicing functions $\mathsf{fwd}_{\vec{t}}$ and $\mathsf{bwd}_{\vec{t}}$
Common sub-modules	
`.GaloisConnection`	Galois connection between lattices defined in parent module

Figure 6 `concurrent-slicing` module overview, release `0.1`