

© 2019 Kevin Liao

A CALCULUS FOR COMPOSABLE, COMPUTATIONAL CRYPTOGRAPHY

BY

KEVIN LIAO

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Andrew Miller

## ABSTRACT

The universal composability (UC) framework is the established standard for analyzing cryptographic protocols in a modular way, such that security is preserved under concurrent composition with arbitrary other protocols. However, although UC is widely used for on-paper proofs, prior attempts at systemizing it have fallen short, either by using a symbolic model (thereby ruling out computational reduction proofs), or by limiting its expressiveness.

In this thesis, we lay the groundwork for building a concrete, executable implementation of the UC framework. Our main contribution is a process calculus, dubbed the Interactive Lambda Calculus (ILC). ILC faithfully captures the computational model underlying UC—interactive Turing machines (ITMs)—by adapting ITMs to a subset of the  $\pi$ -calculus through an affine typing discipline. In other words, *well-typed ILC programs are expressible as ITMs*. In turn, ILC’s strong confluence property enables reasoning about cryptographic security reductions. We use ILC to develop a simplified implementation of UC called SaUCy.

*To my family, for their love and support.*

## ACKNOWLEDGMENTS

This thesis would not have been possible without the support of my co-conspirators. Thanks to my advisor, Professor Andrew Miller, who not only introduced me to this nice confluence of cryptography and programming languages, but also taught me the great value of catchy acronyms in research; thanks to programming languages extraordinaire, Matthew A. Hammer, who guided me through the (often impenetrable) field of PL-research. Also, thanks to my many friends in the Security and Privacy Research at Illinois (SPRAI) group, especially everyone in the Decentralized Systems Lab (DSL), for enriching my two years at Illinois—our collaborations and memes were priceless.

## TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION . . . . .	1
CHAPTER 2	PRELIMINARIES . . . . .	2
2.1	Simulation-Based Security . . . . .	2
2.2	Universal Composition . . . . .	3
2.3	Execution Model . . . . .	3
CHAPTER 3	INTERACTIVE LAMBDA CALCULUS . . . . .	5
3.1	Syntax . . . . .	5
3.2	Static Semantics . . . . .	8
3.3	Dynamic Semantics . . . . .	12
3.4	Metatheory . . . . .	12
CHAPTER 4	SAUCY . . . . .	19
4.1	Probabilistic Polynomial Time in ILC . . . . .	19
4.2	SaUCy Execution Model . . . . .	20
4.3	Defining UC Security in ILC . . . . .	23
4.4	A Composition Theorem in SaUCy . . . . .	24
4.5	Instantiating UC Commitments . . . . .	26
4.6	Reentrancy in SaUCy . . . . .	34
CHAPTER 5	RELATED WORK . . . . .	36
5.1	Process Calculi . . . . .	36
5.2	Tools for Cryptographic Analysis . . . . .	37
5.3	Variations of Universal Composability . . . . .	37
CHAPTER 6	CONCLUSION . . . . .	39
APPENDIX A	INTERACTIVE LAMBDA CALCULUS . . . . .	40
A.1	Algorithmic Typing Rules . . . . .	40
A.2	Type Soundness . . . . .	42
A.3	Confluence . . . . .	56
REFERENCES	. . . . .	59

## CHAPTER 1: INTRODUCTION

In cryptography, a proof of security in the simulation-based universal composability (UC) framework is considered the gold standard for demonstrating that a protocol “does its job securely” [1]. In particular, a UC-secure protocol enjoys the strongest notion of compositionality—it maintains all security properties even when run *concurrently* with arbitrary other protocol instances. This is in contrast with weaker property-based notions that only guarantee security in a standalone setting [2] or under sequential composition [3]. Thus, the benefit of using UC is *modularity*—it supports analyzing complex protocols by composing simpler building blocks. However, the cost of using UC is that security proofs tend to be quite complicated. We believe that applying a formal systemization to UC can help simplify its use, bring new clarity, and provide useful tooling. We envision a future where modularity of cryptographic protocol composition translates to modular implementation as well.

Reviewing prior efforts of applying formal techniques to cryptography, we find they run up against challenges when importing the existing body of UC theory. Either they do not support computational reasoning (which considers issues of probability and computational complexity) [4], do not support message-passing concurrency for distributed protocols [5], or are too expressive (allow for expressing nondeterminism with no computational interpretation) [6].

Our observation is that these approaches diverge from UC at a low level: UC is defined atop the underlying (concurrent) computational model of *interactive Turing machines* (ITMs). The significance of ITMs is that they have a clear computational interpretation, so it is straightforward to relate execution traces to a probabilistic polynomial time computation, as is necessary for cryptographic reduction proofs. The presence of (non-probabilistic) nondeterminism in alternative models of concurrency would frustrate such reduction proofs. ITMs sidestep this issue by having a deterministic (modulo random coin tosses), “single-threaded” execution semantics. That is, processes pass control from one to another each time a message is sent so that *exactly one process is active at any given time*, and, moreover, *the order of activations is fully determined*.

In this thesis, we take up the challenge of faithfully capturing these idioms by designing a new process calculus called the Interactive Lambda Calculus (ILC), which adapts ITMs to a subset of the  $\pi$ -calculus [7] through an affine typing discipline. In other words, *well-typed ILC programs are expressible as ITMs*. We then use ILC to build a concrete, executable implementation of a simplified UC framework, dubbed SaUCy.

## CHAPTER 2: PRELIMINARIES

This chapter provides sufficient background on the UC framework needed to understand this thesis. For a more in-depth treatment of the simulation-based security paradigm (upon which the UC framework is based), we refer readers to Lindell’s tutorial [8]. For a comprehensive treatment of UC, we refer readers to Canetti’s paper [1].

### 2.1 SIMULATION-BASED SECURITY

Security proofs in the UC framework follow the simulation-based security paradigm, also known as the real/ideal paradigm [3]. In UC, an environment  $\mathcal{Z}$  hands inputs to protocol parties and receives their outputs. Intuitively,  $\mathcal{Z}$  represents the external environment consisting of arbitrary protocol executions that may be running concurrently with the protocol in consideration. In the real world, honest parties execute a protocol  $\pi$  over a network controlled by an adversary  $\mathcal{A}$ , who can corrupt (and thus control) parties while freely communicating with  $\mathcal{Z}$ . In the ideal world, honest parties and an incorruptible, trusted party  $\mathcal{F}$  called an *ideal functionality* together execute an idealized protocol  $\rho$ . More specifically, the honest parties of  $\rho$  are “dummy parties” who directly forward their inputs to  $\mathcal{F}$ ; the ideal functionality  $\mathcal{F}$  internally performs the desired cryptographic task and generates outputs for the honest parties. Intuitively, the ideal protocol  $\rho$  is secure by construction, and the security requirements of the task are captured via the instructions of the ideal functionality  $\mathcal{F}$ .

Informally, we say that a protocol  $\pi$  *securely realizes* an ideal functionality  $\mathcal{F}$  if the real world is as secure as the ideal world. The proof follows a standard rhythm: For every adversary  $\mathcal{A}$  attacking the real world, we need to show there exists a simulator  $\mathcal{S}$  that performs an equivalent attack on the ideal world. As there are no meaningful attacks on the ideal world ( $\mathcal{F}$  is secure by construction), it follows that there are no meaningful attacks on the real world ( $\pi$  is secure). This can be stated a bit more precisely as follows.

**Definition 2.1** (UC-realizes). A protocol  $\pi$  UC-realizes an ideal functionality  $\mathcal{F}$  if for every probabilistic polynomial time (p.p.t.) adversary  $\mathcal{A}$  there exists a p.p.t simulator  $\mathcal{S}$  such that no p.p.t. environment  $\mathcal{Z}$  can distinguish the real world (with  $\pi$  and  $\mathcal{A}$ ) from the ideal world (with  $\mathcal{F}$  and  $\mathcal{S}$ ).

A few remarks are in order. First, in the context of computational security, we are only interested in computationally bounded adversaries, i.e., adversaries whose running time is polynomial in some security parameter. Second, we should be a bit more precise about



what it means for the real world and ideal world to be indistinguishable to an environment. Let  $\text{EXEC}_{\pi, \mathcal{A}, \mathcal{Z}}(z)$  denote the random variable (over the local random choices of all involved machines) describing the output of an execution of  $\pi$  with environment  $\mathcal{Z}$  and adversary  $\mathcal{S}$ , on input  $z \in \{0, 1\}^*$ . Let  $\text{EXEC}_{\pi, \mathcal{A}, \mathcal{Z}}$  denote the ensemble  $\{\text{EXEC}_{\pi, \mathcal{A}, \mathcal{Z}}(z)\}_{z \in \{0, 1\}^*}$ . By “real world and ideal world being indistinguishable,” we mean that the ensembles  $\text{EXEC}_{\pi, \mathcal{A}, \mathcal{Z}}$  and  $\text{EXEC}_{\rho, \mathcal{S}, \mathcal{Z}}$  are indistinguishable, where  $\rho$  is the ideal protocol for  $\mathcal{F}$ . That is, for any input, the probability that  $\mathcal{Z}$  outputs 1 in the real world differs by at most a negligible amount from the probability that  $\mathcal{Z}$  outputs 1 in the ideal world.

## 2.2 UNIVERSAL COMPOSITION

The UC framework comes with a composition theorem, which provides composability guarantees for protocols proven secure in the framework. Let  $\pi_1$  be a protocol that securely realizes a functionality  $\mathcal{F}_1$ . If a protocol  $\pi_2$ , using the functionality  $\mathcal{F}_1$  as a subroutine, securely realizes a functionality  $\mathcal{F}_2$ , then the composed protocol  $\pi_2^{\pi_1/\mathcal{F}_1}$ , where calls to  $\mathcal{F}_1$  are replaced by calls to  $\pi_1$ , securely realizes  $\mathcal{F}_2$ . Therefore, it suffices to analyze the security of the simpler protocol  $\pi_2$  in the  $\mathcal{F}_1$ -hybrid model, where the parties run  $\pi_2$  with access to the ideal functionality  $\mathcal{F}_1$ . Thus,  $\pi_2$  behaves just like the ideal functionality  $\mathcal{F}_2$ , even when composed concurrently with an arbitrary protocol  $\pi_1$  and when using the protocol as a building block for more advanced protocols.

## 2.3 EXECUTION MODEL

The entities taking part in protocol executions (protocol machines, functionalities, adversaries, and environments) are described as interactive Turing machines (ITMs). The execution of a system of ITMs is initiated by the environment, which provides input to and obtains output from the protocol machines, and also communicates with the adversary. The adversary has access to the ideal functionalities in the hybrid models, in some models it also serves as a network among the protocol machines. Throughout the execution, ITMs are activated one-by-one, where the exact order of the activations depends on the considered model. Processes pass control from one to another each time a message is sent so that *exactly one process is active at any given time*, and, moreover, *the order of activations is fully determined*. This gives ITMs a clear computational interpretation, which is necessary for the above proofs (in particular, cryptographic reductions) to go through.

One might wonder whether this simplistic, seemingly sequential model of computation ad-

equately represents concurrent computation. Traditional models of concurrent computation permit arbitrary (nondeterministic) interleavings of local, atomic operations performed by processes (subject to causality constraints). In this setting, the granularity of atomic events is a key factor in determining the level of concurrency under consideration, and thus the expressive power of the model. In the UC setting, instead of determining the granularity of atomic events in advance and then considering all possible interleavings of these events, ITMs let processes themselves determine both the granularity of atomic events (by deciding when to send a message to another machine) and the specific interleaving of events (by deciding which machine to send a message to). Arbitrary and adversarial interleaving of events is captured by including in the system explicit adversarial processes that represent the variability in timing and ordering of events. Moreover, the ability to restrict attention to computationally bounded scheduling of events—as opposed to fully nondeterministic scheduling—is crucial for establishing computational security guarantees.

## CHAPTER 3: INTERACTIVE LAMBDA CALCULUS

Why do we need another process calculus in the first place? Where do existing ones fall short? On the one hand, process calculi such as the  $\pi$ -calculus [7] and its cryptography-oriented variants [6, 9, 10] are not a good fit to ITMs, since they permit non-confluent reductions by design (i.e., non-probabilistic nondeterminism). On the other hand, various other calculi that do enjoy confluence are overly restrictive, only allowing for fixed or two-party communications [11, 4, 12].

ILC fills this gap by adapting ITMs to a subset of the  $\pi$ -calculus through an affine typing discipline. That is, resources designated as affine are non-duplicable. To maintain that only one process is active (can write) at any given time, processes implicitly pass around an affine *write token*  $\textcircled{w}$  by virtue of where they perform read and write effects: When process  $A$  writes to process  $B$ , process  $A$  “spends” the write token and process  $B$  “earns” the write token. Moreover, to maintain that the order of activations is fully determined, the read endpoints of channels are affine resources, and so each write operation corresponds to a single, unique read operation. Together, these give ILC its central metatheoretic property of *confluence*.

The importance of confluence is that the only nondeterminism in an ILC program is due to random coin tosses taken by processes, which have a well-defined distribution. Additionally, any apparent concurrency hazards, such as adversarial scheduling of messages in an asynchronous network, are due to an explicit adversary process rather than uncertainty built into the model itself. This eliminates non-probabilistic nondeterminism, and so ILC programs are amenable to the reasoning patterns necessary for establishing computational security guarantees.

Next, we go through ILC in full, presenting its syntax, static semantics, dynamic semantics, and metatheory.

### 3.1 SYNTAX

The syntax of types is given in Figure 3.1. Types (written  $U, V$ ) are bifurcated into unrestricted types (written  $A, B$ ) and affine types (written  $X, Y$ ).

A subset of the unrestricted types are sendable types (written  $S, T$ ), i.e., the types of values that can be sent over channels. This restriction ensures that channels model network channels, which send only data. The sendable types include unit ( $\mathbf{1}$ ), products ( $S \times T$ ), and sums ( $S + T$ ).

All types	$U, V ::= A$	Unrestricted type
	$  X$	Affine type
Sendable types	$S, T ::= \mathbb{1}$	Unit type
	$  S \times T$	Sendable product
	$  S + T$	Sendable sum
Unrestricted types	$A, B ::= S$	Sendable type
	$  \mathbf{Wr} S$	Write endpoint
	$  A \times B$	Unrestricted product
	$  A + B$	Unrestricted sum
	$  A \rightarrow_{\infty} U$	Unrestricted arrow
	$  A \rightarrow_{\mathbf{w}} U$	Unrestricted write arrow
Affine types	$X, Y ::= !A$	Bang type
	$  \mathbf{Rd} S$	Read endpoint
	$  X \otimes Y$	Affine product
	$  X \oplus Y$	Affine sum
	$  X \rightarrow_1 U$	Affine arrow
Syntax labels	$\ell ::= \pi$	Multiplicity
	$  \mathbf{w}$	Write
Multiplicity labels	$\pi ::= 1$	Affine
	$  \infty$	Unrestricted
Unrestricted typings	$\Gamma ::= \cdot$	Empty
	$  \Gamma, x : A$	Unrestricted extension
Affine typings	$\Delta ::= \cdot$	Empty
	$  \Delta, x : X$	Affine extension
	$  \Delta, \textcircled{\mathbf{w}}$	Write token extension

Figure 3.1: Type syntax.

The unrestricted types include the sendable types, write endpoint types ( $\mathbf{Wr} S$ ), products ( $A \times B$ ), sums ( $A + B$ ), arrows ( $A \rightarrow_{\infty} U$  or simply  $A \rightarrow U$ ), and write arrows ( $A \rightarrow_{\mathbf{w}} U$ ). Jumping ahead slightly, write arrows specify unrestricted abstractions for which the write token can be moved into the affine context of the abstraction body during  $\beta$ -reduction.

The affine types include bang types ( $!A$ ), read endpoint types ( $\mathbf{Rd} S$ ), products ( $X \otimes Y$ ), sums ( $X \oplus Y$ ), and arrows ( $X \rightarrow_1 U$  or simply  $X \multimap U$ ). The bang type operator lifts an unrestricted value of type  $A$  into an affine type  $!A$ . Note that the write token  $\textcircled{\mathbf{w}}$  lives in the affine context, though it cannot be bound to any variable. Instead, it flows around implicitly by virtue of where read and write effects are performed (more details will be explained in short order).

For concision, certain syntactic forms are parameterized by a multiplicity  $\pi$  to distinguish between the unrestricted ( $\infty$ ) and affine (1) counterparts; other syntactic forms are param-

Values	$v ::=$	$()$	Unit
		$  (v_1, v_2)_\ell$	Pair
		$  \mathbf{inj}_\ell^1(v)$	Left injection
		$  \mathbf{inj}_\ell^2(v)$	Right injection
		$  \lambda_\ell x. e$	Function
		$  c$	Channel
		$  !v$	Banged value
Channel endpoints	$c ::=$	$\mathbf{Read}(d)$	Read endpoint
		$  \mathbf{Write}(d)$	Write endpoint
Channel names	$d ::=$	$\dots$	Names

Figure 3.2: Value syntax.

eterized by a syntax label  $\ell$ , which includes the multiplicity labels and the write label  $\mathbf{w}$  (related to write effects). On introduction and elimination forms for functions (abstraction, application, and fixed points), the label  $\mathbf{w}$  denotes variants that move around the write token as explained above. On introduction and elimination forms for products and sums, the label  $\mathbf{w}$  denotes the sendable variants.

The syntax of values is given in Figure 3.2. Values in ILC (written  $v$ ) include unit, pairs, sums, lambda expressions, channel endpoints (written  $c$ ), and banged values. We distinguish between the names of channel endpoints— $\mathbf{Read}(d)$  and  $\mathbf{Write}(d)$ —and the channel  $d$  itself that binds them.

The syntax of expressions is given in Figure 3.3. ILC supports a fairly standard feature set of expressions. Bang-typed values have introduction form  $!e$  and elimination form  $\mathbf{j}e$ . The more interesting expressions are those related to communication and concurrency:

- *Restriction*:  $\nu(x_1, x_2).e$  binds a read endpoint  $x_1$  and a corresponding write endpoint  $x_2$  in  $e$ .
- *Write*:  $\mathbf{wr}(e_1, e_2)$  sends the value that  $e_1$  evaluates to on the write endpoint that  $e_2$  evaluates to.
- *Read*:  $\mathbf{rd}(e_1, x.e_2)$  reads a value from the read endpoint that  $e_1$  evaluates to and binds the value-endpoint pair as  $x$  in  $e_2$ .
- *Choice*:  $\mathbf{ch}(e_1, x_1.e_3, e_2, x_2.e_4)$  allows a process to continue as either  $e_3$  or  $e_4$  based on some initial read event on one of the read endpoints that  $e_1$  and  $e_2$  evaluate to. The value read over the channel and the two read endpoints are rebound in a 3-tuple as  $x_1$  in  $e_3$  or  $x_2$  in  $e_4$ . Here, we show only binary choice, but it can be generalized to the  $n$ -ary case.

Expressions	$e ::= x$	Variable
	$()$	Unit
	$(e_1, e_2)_\ell$	Pair introduction
	$\text{inj}_\ell^i(e)$	Injection introduction
	$\text{split}_\ell(e_1, x_1.x_2.e_2)$	Pair elimination
	$\text{case}_\ell(e, x_1.e_1, x_2.e_2)$	Injection elimination
	$\lambda_\ell x. e$	Function abstraction
	$(e_1 e_2)_\ell$	Function application
	$\text{fix}_\ell(x.e)$	Fixpoint
	$\text{let}_\pi(e_1, x.e_2)$	Let binding
	$!e$	Bang introduction
	$\text{j} e$	Bang elimination
	$\nu(x_1, x_2). e$	Channel restriction
	$\text{wr}(e_1, e_2)$	Write operation
	$\text{rd}(e_1, x.e_2)$	Read operation
	$\text{ch}(e_1, x_1.e_3, e_2, x_2.e_4)$	External choice
	$e_1 \mid\triangleright e_2$	Fork operation

Figure 3.3: Expression syntax.

- *Fork*:  $e_1 \mid\triangleright e_2$  spawns a child process  $e_1$  and continues as  $e_2$ .

### 3.2 STATIC SEMANTICS

ILC terms have either an unrestricted type, meaning they can be freely copied, or an affine type, meaning they can be used at most once. Affine typing serves a special purpose, namely, to ensure that ILC processes have a determined sequence of activations, as is required in ITMs. This is achieved through the following type-level invariants:

- *Only one process is active at any given time.* Processes implicitly pass around an affine “write token”  $\textcircled{w}$  by virtue of where they perform read and write effects. In order for process  $A$  to write to process  $B$ , process  $A$  must first own the write token. Because the write token is unique, at most one process owns the write token (“is active” or “can write”) at any given time. When process  $B$  reads the message from  $A$ , process  $B$  earns the write token, thereby conserving its uniqueness and now allowing process  $B$  to write to some other process.
- *The order of activations is deterministic.* Each channel (or “tape” in ITM parlance) has a read endpoint and a write endpoint. The read endpoint is an affine resource, and so it

is owned by at most one process. This ensures that each write operation corresponds to a single, unique read operation.

Intuitively, the first invariant rules out the possibility of write nondeterminism. Consider the case in which two processes are trying to execute writes in parallel, which would lead to a race condition. This does not typecheck, since the affine write token belongs to at most one process. One might justifiably wonder why write endpoints are unrestricted and read endpoints are affine. Note that if two processes are trying to write in parallel, the two write endpoints need not be the same, so making write endpoints affine would not help our case in eliminating write nondeterminism.

Dually, the second invariant rules out the possibility of read nondeterminism. Consider the case in which two processes  $A$  and  $B$  are listening on the same read endpoint. If a process  $C$  writes on the corresponding write endpoint, which of  $A$  or  $B$  (or both) gets activation? If only one of them is activated, then we have a source of nondeterminism. If both are activated, now  $A$  and  $B$  both own write tokens, violating its affinity. In any case, this does not typecheck since read endpoints are affine resources, making it impossible for two processes  $A$  and  $B$  to listen on the same read endpoint. Together, these invariants ensure that processes have a determined sequence of activations as desired.

The typing rules of ILC are collected in Figures 3.4 and 3.5. An algorithmic version of the rules appears in Appendix A.1. We read the typing judgement  $\Delta; \Gamma \vdash e : U$  as “under affine context  $\Delta$  and unrestricted context  $\Gamma$ , expression  $e$  has type  $U$ .” In full detail, the typing judgement also includes a typing context  $\Psi$ , which maps channel names  $d$  to sendable types  $S$ . However, it is only used in two special rules for typing channel endpoints that do not arise for source level programs, but will be needed to typecheck a running program that has performed channel allocation:

$$\frac{\Psi(d) = S}{\Psi; \Delta; \Gamma \vdash \text{Read}(d) : \text{Rd } S} \text{ rdend} \qquad \frac{\Psi(d) = S}{\Psi; \Delta; \Gamma \vdash \text{Write}(d) : \text{Wr } S} \text{ wrend}$$

This pair of rules establish the canonical forms for the types of channel endpoints,  $\text{Rd } S$  and  $\text{Wr } S$ . We use the metavariable  $c$  to range over these two canonical forms.

The typing rules for the functional fragment of ILC are fairly standard, except that they now have unrestricted and affine variants (and for some, sendable variants).

$$\frac{\cdot; \Gamma, x : A \vdash e : U}{\Delta; \Gamma \vdash \lambda_\infty x. e : A \rightarrow_\infty U} \text{ uabs}$$

The rule for unrestricted abstraction (uabs) extends the unrestricted context  $\Gamma$  with  $x : A$  before checking the body  $e$  of the abstraction. Notice that because unrestricted abstractions can be duplicated, the body must be affinely closed (cannot contain free affine variables).

$$\frac{\textcircled{w}; \Gamma, x : A \vdash e : U}{\Delta; \Gamma \vdash \lambda_w x. e : A \rightarrow_w U} \text{uabs}$$

The rule for write abstraction (wabs) is similar to uabs. The only difference is that wabs extends the affine context with the write token before checking the body  $e$  of the abstraction. Dually, the write application rule (wapp) stipulates that a process must own the write token in order to apply a write abstraction.

$$\frac{\Delta, x : X; \Gamma \vdash e : U}{\Delta; \Gamma \vdash \lambda_1 x. e : X \rightarrow_1 U} \text{aabs}$$

The rule for affine abstraction (aabs) is analogous to uabs, but notice that the body *need not* be affinely closed, since affine abstractions cannot be duplicated. It turns out that most affine functions we write *are* affinely closed, and so such a function  $f : X \multimap U$  can be made into an unrestricted function  $g : A \rightarrow X \multimap U$  by adding a leading unrestricted argument.

$$\frac{\Delta; \Gamma \vdash e : A}{\Delta; \Gamma \vdash !e : !A} \text{bang}$$

The bang rule turns an unrestrictedly typed expression  $e : A$  into an affinely typed expression  $e : !A$ . Dually, the gnab rule turns an affinely typed expression  $e : !A$  into an unrestrictedly typed expression  $e : A$ .

$$\frac{\Delta_1; \Gamma \vdash e_1 : U \quad \Delta_2; \Gamma \vdash e_2 : V}{\Delta_1, \Delta_2; \Gamma \vdash e_1 \mid\triangleright e_2 : V} \text{fork}$$

The fork expression  $e_1 \mid\triangleright e_2$  spawns a child process  $e_1$  and continues as  $e_2$ . Its typing rule says that if we can partition the affine context as  $\Delta_1, \Delta_2$  such that  $e_1$  has type  $U$  under contexts  $\Delta_1; \Gamma$  and  $e_2$  has type  $V$  under contexts  $\Delta_2; \Gamma$ , then the expression has type  $V$ . Notice that affine resources (e.g., read endpoints and the write token) must be split between



the child process and the parent process, thereby preventing their duplication.

$$\frac{\Delta_1; \Gamma \vdash e_1 : S \quad \Delta_2; \Gamma \vdash e_2 : \mathbf{Wr} S}{\Delta_1, \Delta_2, \textcircled{\mathbf{w}}; \Gamma \vdash \mathbf{wr}(e_1, e_2) : \mathbf{1}} \text{ wr}$$

The write expression  $\mathbf{wr}(e_1, e_2)$  sends the value that  $e_1$  evaluates to on the write endpoint that  $e_2$  evaluates to. One thing to mention is that only values of a sendable type (ranged over by  $S$ ) can be sent over channels (more on this later). Its typing rule says that if we own the write token and we can partition the affine context as  $\Delta_1, \Delta_2$  such that  $e_1$  has type  $S$  under contexts  $\Delta_1; \Gamma$  and  $e_2$  evaluates to a write endpoint (of type  $\mathbf{Wr} S$ ) under contexts  $\Delta_2; \Gamma$ , then the expression has type  $\mathbf{1}$  (unit). Notice that typing a write expression spends the write token, and so it cannot execute another write until it gets “reactivated” by reading from some other process.

$$\frac{\textcircled{\mathbf{w}} \notin \Delta_2 \quad \Delta_1; \Gamma \vdash e_1 : \mathbf{Rd} S \quad \Delta_2, \textcircled{\mathbf{w}}, x : !S \otimes \mathbf{Rd} S; \Gamma \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \mathbf{rd}(e_1, x.e_2) : U} \text{ rd}$$

The read expression  $\mathbf{rd}(e_1, x.e_2)$  reads a value on the read endpoint that  $e_1$  evaluates to and binds the value-endpoint pair as  $x$  in the affine context of  $e_2$ . Rebinding the read endpoint allows it to be reused. Its typing rule says that if we can partition the affine context as  $\Delta_1, \Delta_2$  such that  $e_1$  evaluates to a read endpoint (of type  $\mathbf{Rd} S$ ) under contexts  $\Delta_1; \Gamma$ , and  $e_2$  has type  $U$  under contexts  $\Delta_2, \textcircled{\mathbf{w}}, x : !S \otimes \mathbf{Rd} S; \Gamma$ , then the expression has type  $U$ .

There are a few things to unpack here. First, we explain the affine product type  $!S \otimes \mathbf{Rd} S$ . Since sendable values are unrestricted and read endpoints are affine, the value read on the channel is wrapped in a  $!$  operator (pronounced “bang”) so that it can be placed in an affine pair. Next, observe that  $\textcircled{\mathbf{w}}$  is available in the body  $e_2$  of the read expression (i.e., it is conserved), but only under the condition that it is not already in the affine context  $\Delta_2$  (otherwise, a process could arbitrarily mint write tokens, violating its affinity).

$$\frac{\Delta, x_1 : \mathbf{Rd} S; \Gamma, x_2 : \mathbf{Wr} S \vdash e : U}{\Delta; \Gamma \vdash \nu(x_1, x_2).e : U} \text{ nu}$$

The nu rule extends the affine context  $\Delta$  with a read endpoint  $x_1 : \mathbf{Rd} S$  and the unrestricted

context  $\Gamma$  with a corresponding write endpoint  $x_2 : \text{Wr } S$  before typing the body  $e$ .

$$\frac{\begin{array}{c} \textcircled{w} \notin \Delta_3 \quad \Delta_1; \Gamma \vdash e_1 : \text{Rd } S \quad \Delta_2; \Gamma \vdash e_2 : \text{Rd } T \\ \Delta_3, \textcircled{w}, x_1 : !S \otimes \text{Rd } S \otimes \text{Rd } T; \Gamma \vdash e_3 : U \\ \Delta_3, \textcircled{w}, x_2 : !T \otimes \text{Rd } S \otimes \text{Rd } T; \Gamma \vdash e_4 : U \end{array}}{\Delta_1, \Delta_2, \Delta_3; \Gamma \vdash \text{ch}(e_1, x_1.e_3, e_2, x_2.e_4) : U} \text{ choice}$$

The choice rule partitions the affine context as  $\Delta_1, \Delta_2, \Delta_3$ . The first two affine contexts are used to type  $e_1 : \text{Rd } S$  and  $e_2 : \text{Rd } T$ , respectively. The third affine context  $\Delta_3$  is extended with the affine write token and a variable  $x_1$  (or  $x_2$ ) binding an affine 3-tuple containing the read value and the two read endpoints before checking the continuation  $e_3$  (or  $e_4$ ). While somewhat cumbersome, the generality of this rule allows both read endpoints to be used in either continuation.

### 3.3 DYNAMIC SEMANTICS

Figure 3.6 defines the dynamic syntax of ILC. Figures 3.7 and 3.8 define the configuration and local reduction semantics of ILC, respectively. We define a *configuration*  $C$  as a tuple of dynamic channel and process names  $\Sigma$ , and a pool of running and terminated processes  $\pi$ .

We read the configuration reduction judgment  $C_1 \longrightarrow C_2$  as “configuration  $C_1$  steps to configuration  $C_2$ ,” and the local stepping judgment  $e_1 \longrightarrow e_2$  for a single process  $e$  as “expression  $e_1$  steps to expression  $e_2$ .” The rules of local stepping follow a standard call-by-value semantics, where we streamline the definition with an evaluation context  $E$ .

Configuration stepping consists of six rules. These include a congruence rule *congr* that permits some of the other rules to be simpler, by making the order of the pool unimportant. The relation  $\pi_1 \equiv_{\text{perm}} \pi_2$  holds when  $\pi_2$  is a permutation of  $\pi_1$ . The other five rules consist of local stepping (via *local*), creating new processes (via *fork*), creating new channels (via *nu*), read-write interactions (via *rw*), and choice-write interactions (via *cw*). To avoid allocating the same name twice, the name set  $\Sigma$  records names of allocated channels and processes. We define the relation  $c_1 \rightsquigarrow c_2$  to hold when  $c_1$  is the write endpoint of a corresponding read endpoint  $c_2$ .

### 3.4 METATHEORY

Intuitively, ILC’s type system design enforces that a configuration’s reduction consists of a unique (*deterministic*) sequence of reader-writer process pairings, and is *confluent* with

any other reduction choice that exchanges the order of *other* (non-interactive) reduction steps. As explained in Section 3, ILC’s type system does so by restricting the write effects (via an affine write token) and read effects (via affine read endpoints) of processes. The proofs of type soundness, whose statements we discuss next, establish the validity of these invariants. These language-level invariants support confluence theorems, also stated below. These theorems include *full confluence*: Any two full reductions of a configuration yield a pair of equivalent configurations (isomorphic, up to a renaming of nondeterministic name choices).

### 3.4.1 Type Soundness

We prove type soundness of ILC via mostly-standard notions of progress and preservation. To state these theorems, we follow the usual recipe, except that we give a special definition of program termination that permits deadlocks. (Recall that ILC is concerned with enforcing *confluence* as its central metatheoretic property, *not* deadlock freedom.) Informally,  $C \text{ term}$  holds when either:

1.  $C$  is fully normal: Every process in  $C$  is normalized (consists of a value), or
2.  $C$  is (at least partially) deadlocked: Some (possibly empty) portion of  $C$  is normal, and there exists one or more reading processes in  $C$ , or there exists one or more writing processes in  $C$ , however, no reader-writer process pair exists for a common channel.

We also extend the type system given in Section 3.2 with typing rules for configurations, including process pool typings  $\Phi$  from process names  $p$  to types  $U$ . These details, along with the proofs of progress and preservation, can be found in Appendix A.2.

**Theorem 3.1** (Progress). *If  $\Psi \vdash C : \Phi$ , then either  $C \text{ term}$  or there exists  $C'$  such that  $C \longrightarrow C'$ .*

**Theorem 3.2** (Preservation). *If  $\Psi \vdash C : \Phi$  and  $C \longrightarrow C'$ , then there exists  $\Psi' \supseteq \Psi$  and  $\Phi' \supseteq \Phi$  such that  $\Psi' \vdash C' : \Phi'$ .*

### 3.4.2 Confluence

Confluence implies, among other things, that the order of reduction steps is inconsequential, and that no process scheduling choices will affect the final outcome. ILC’s type system enforces confluence up to nondeterministic naming choices in rules `nu` and `fork` (Figure 3.7).

To account for different choices of dynamically-named channels and processes, respectively, we state and prove confluence with respect to a renaming function  $f$ , which consistently renames these choices in a related configuration:

**Theorem 3.3** (Single-step confluence). *For all well-typed configurations  $C$ , if  $C \longrightarrow C_1$  and  $C \longrightarrow C_2$ , then there exists a renaming function  $f$  such that either:*

1.  $C_1 = f(C_2)$ , or
2. there exists  $C_3$  such that  $C_1 \longrightarrow C_3$  and  $f(C_2) \longrightarrow C_3$ .

Intuitively, the sister configuration  $C_2$  is either different because of a name choice (case 1), or a different process scheduling choice (case 2). In either case, there exists a renaming of any choice made to reach  $C_2$ , captured by function  $f$ . By composing multiple uses of this theorem, and the renaming functions that they construct, we prove a multi-step notion of confluence that reduces a single configuration  $C$  to two equivalent terminal configurations,  $C_1$  and  $C_2$ :

**Theorem 3.4** (Full confluence). *For all well-typed configurations  $C$ , if  $C \longrightarrow^* C_1$  and  $C \longrightarrow^* C_2$  and  $C_1$  **term** and  $C_2$  **term**, then there exists renaming function  $f$  such that  $C_1 = f(C_2)$ .*

The proofs of these statements can be found in Appendix A.3.

$\boxed{\Delta; \Gamma \vdash e : U}$  Under affine context  $\Delta$  and unrestricted context  $\Gamma$ , expression  $e$  has type  $U$ .

$$\begin{array}{c}
\frac{\Gamma(x) = A}{\Delta; \Gamma \vdash x : A} \text{uvar} \qquad \frac{\Delta(x) = X}{\Delta; \Gamma \vdash x : X} \text{avar} \qquad \frac{}{\Delta; \Gamma \vdash () : \mathbb{1}} \text{unit} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e_1 : A_1 \quad \Delta_2; \Gamma \vdash e_2 : A_2}{\Delta_1, \Delta_2; \Gamma \vdash (e_1, e_2)_\infty : A_1 \times A_2} \text{upair} \qquad \frac{\Delta_1; \Gamma \vdash e_1 : S_1 \quad \Delta_2; \Gamma \vdash e_2 : S_2}{\Delta_1, \Delta_2; \Gamma \vdash (e_1, e_2)_w : S_1 \times S_2} \text{spair} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e_1 : X_1 \quad \Delta_2; \Gamma \vdash e_2 : X_2}{\Delta_1, \Delta_2; \Gamma \vdash (e_1, e_2)_1 : X_1 \otimes X_2} \text{apair} \qquad \frac{i \in \{1, 2\} \quad \Delta; \Gamma \vdash e : A_i}{\Delta; \Gamma \vdash \text{inj}_\infty^i(e) : A_1 + A_2} \text{uinj} \\[10pt]
\frac{i \in \{1, 2\} \quad \Delta; \Gamma \vdash e : S_i}{\Delta; \Gamma \vdash \text{inj}_w^i(e) : S_1 + S_2} \text{sinj} \qquad \frac{i \in \{1, 2\} \quad \Delta; \Gamma \vdash e : X_i}{\Delta; \Gamma \vdash \text{inj}_1^i(e) : X_1 \oplus X_2} \text{ainj} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e_1 : A_1 \times A_2 \quad \Delta_2; \Gamma, x_1 : A_1, x_2 : A_2 \vdash e : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{split}_\infty(e_1, x_1.x_2.e_2) : U} \text{usplit} \qquad \frac{\Delta_1; \Gamma \vdash e_1 : S_1 \times S_2 \quad \Delta_2; \Gamma, x_1 : S_1, x_2 : S_2 \vdash e : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{split}_w(e_1, x_1.x_2.e_2) : U} \text{ssplit} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e_1 : X_1 \otimes X_2 \quad \Delta_2, x_1 : X_1, x_2 : X_2; \Gamma \vdash e : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{split}_1(e_1, x_1.x_2.e_2) : U} \text{asplit} \qquad \frac{\Delta_1; \Gamma \vdash e : A_1 + A_2 \quad \Delta_2; \Gamma, x_1 : A_1 \vdash e_1 : U \quad \Delta_2; \Gamma, x_2 : A_2 \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{case}_\infty(e, x_1.e_1, x_2.e_2) : U} \text{ucase} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e : S_1 + S_2 \quad \Delta_2; \Gamma, x_1 : S_1 \vdash e_1 : U \quad \Delta_2; \Gamma, x_2 : S_2 \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{case}_w(e, x_1.e_1, x_2.e_2) : U} \text{scase} \qquad \frac{\Delta_1; \Gamma \vdash e : X_1 \oplus X_2 \quad \Delta_2, x_1 : X_1; \Gamma \vdash e_1 : U \quad \Delta_2, x_2 : X_2; \Gamma \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{case}_1(e, x_1.e_1, x_2.e_2) : U} \text{acase} \\[10pt]
\frac{\cdot; \Gamma, x : A \vdash e : U}{\Delta; \Gamma \vdash \lambda_\infty x. e : A \rightarrow_\infty U} \text{uabs} \qquad \frac{\textcircled{w}; \Gamma, x : A \vdash e : U}{\Delta; \Gamma \vdash \lambda_w x. e : A \rightarrow_w U} \text{wabs} \\[10pt]
\frac{\Delta, x : X; \Gamma \vdash e : U}{\Delta; \Gamma \vdash \lambda_1 x. e : X \rightarrow_1 U} \text{aabs} \qquad \frac{\Delta_1; \Gamma \vdash e_2 : A \quad \Delta_2; \Gamma \vdash e_1 : A \rightarrow_\infty U}{\Delta_1, \Delta_2; \Gamma \vdash (e_1 e_2)_\infty : U} \text{uapp} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e_2 : A \quad \Delta_2; \Gamma \vdash e_1 : A \rightarrow_w U}{\Delta_1, \Delta_2, \textcircled{w}; \Gamma \vdash (e_1 e_2)_w : U} \text{wapp} \qquad \frac{\Delta_1; \Gamma \vdash e_2 : X \quad \Delta_2; \Gamma \vdash e_1 : X \rightarrow_1 U}{\Delta_1, \Delta_2; \Gamma \vdash (e_1 e_2)_1 : U} \text{aapp}
\end{array}$$

Figure 3.4: Typing rules.

$$\begin{array}{c}
\frac{\cdot; \Gamma, x : A \rightarrow_{\infty} U \vdash e : A \rightarrow_{\infty} U}{\Delta; \Gamma \vdash \text{fix}_{\infty}(x.e) : A \rightarrow_{\infty} U} \text{ufix} \qquad \frac{\cdot; \Gamma, x : A \rightarrow_{\text{w}} U \vdash e : A \rightarrow_{\text{w}} U}{\Delta; \Gamma \vdash \text{fix}_{\text{w}}(x.e) : A \rightarrow_{\text{w}} U} \text{wfix} \\[10pt]
\frac{x : X \rightarrow_1 U; \Gamma \vdash e : X \rightarrow_1 U}{\Delta; \Gamma \vdash \text{fix}_1(x.e) : X \rightarrow_1 U} \text{afix} \qquad \frac{\Delta_1; \Gamma \vdash e_1 : A \quad \Delta_2; \Gamma, x : A \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{let}_{\infty}(e_1, x.e_2) : U} \text{ulet} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e_1 : X \quad \Delta_2, x : X; \Gamma \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{let}_1(e_1, x.e_2) : U} \text{alet} \qquad \frac{\Delta; \Gamma \vdash e : A}{\Delta; \Gamma \vdash !e : !A} \text{bang} \\[10pt]
\frac{\Delta; \Gamma \vdash e : !A}{\Delta; \Gamma \vdash \downarrow e : A} \text{gnab} \qquad \frac{\Delta, x_1 : \text{Rd } S; \Gamma, x_2 : \text{Wr } S \vdash e : U}{\Delta; \Gamma \vdash \nu(x_1, x_2).e : U} \text{nu} \\[10pt]
\frac{\Delta_1; \Gamma \vdash e_1 : S \quad \Delta_2; \Gamma \vdash e_2 : \text{Wr } S}{\Delta_1, \Delta_2, \textcircled{\text{w}}; \Gamma \vdash \text{wr}(e_1, e_2) : \mathbb{1}} \text{wr} \qquad \frac{\textcircled{\text{w}} \notin \Delta_2 \quad \Delta_1; \Gamma \vdash e_1 : \text{Rd } S \quad \Delta_2, \textcircled{\text{w}}, x : !S \otimes \text{Rd } S; \Gamma \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \text{rd}(e_1, x.e_2) : U} \text{rd} \\[10pt]
\frac{\textcircled{\text{w}} \notin \Delta_3 \quad \Delta_1; \Gamma \vdash e_1 : \text{Rd } S \quad \Delta_2; \Gamma \vdash e_2 : \text{Rd } T \quad \Delta_3, \textcircled{\text{w}}, x_1 : !S \otimes \text{Rd } S \otimes \text{Rd } T; \Gamma \vdash e_3 : U \quad \Delta_3, \textcircled{\text{w}}, x_2 : !T \otimes \text{Rd } S \otimes \text{Rd } T; \Gamma \vdash e_4 : U}{\Delta_1, \Delta_2, \Delta_3; \Gamma \vdash \text{ch}(e_1, x_1.e_3, e_2, x_2.e_4) : U} \text{choice} \qquad \frac{\Delta_1; \Gamma \vdash e_1 : U \quad \Delta_2; \Gamma \vdash e_2 : V}{\Delta_1, \Delta_2; \Gamma \vdash e_1 \mid \triangleright e_2 : V} \text{fork}
\end{array}$$

Figure 3.5: Typing rules continued.

$$\begin{array}{ll}
\text{Process names} & p, q ::= \dots \\
\text{Name sets} & \Sigma ::= \varepsilon \mid \Sigma, d \mid \Sigma, p \\
\text{Process pools} & \pi ::= \varepsilon \mid \pi, p : e \\
\text{Configurations} & C ::= \langle \Sigma; \pi \rangle \\[10pt]
\text{Evaluation contexts} & E ::= \bullet \mid (E, e)_{\ell} \mid (v, E)_{\ell} \mid \text{inj}_{\ell}^i(E) \\
& \mid \text{split}_{\ell}(E, x_1.x_2.e) \mid \text{case}_{\ell}(E, x_1.e_1, x_2.e_2) \\
& \mid (Ee)_{\ell} \mid (vE)_{\ell} \mid \text{let}_{\pi}(E, x.e) \mid !E \mid \downarrow E \\
& \mid \text{wr}(E, e) \mid \text{wr}(v, E) \mid \text{rd}(E, x.e) \\
& \mid \text{ch}(E, x_1.e_3, e_2, x_2.e_4) \mid \text{ch}(c, x_1.e_3, E, x_2.e_4)
\end{array}$$

Figure 3.6: ILC dynamic syntax.

$\boxed{C_1 \equiv C_2}$  Configurations  $C_1$  and  $C_2$  are equivalent.

$$\frac{\pi_1 \equiv_{\text{perm}} \pi_2}{\langle \Sigma; \pi_1 \rangle \equiv \langle \Sigma; \pi_2 \rangle} \text{permProcs}$$

$\boxed{c_1 \rightsquigarrow c_2}$  Write endpoint  $c_1$  connects to read endpoint  $c_2$ .

$$\frac{}{\text{Write}(d) \rightsquigarrow \text{Read}(d)} \text{bind}$$

$\boxed{C_1 \longrightarrow C_2}$  Configuration  $C_1$  reduces to  $C_2$ .

$$\frac{e_1 \longrightarrow e_2}{\langle \Sigma; \pi, p : E[e_1] \rangle \longrightarrow \langle \Sigma; \pi, p : E[e_2] \rangle} \text{local}$$

$$\frac{q \notin \Sigma}{\langle \Sigma; \pi, p : E[e_1 \mid \triangleright e_2] \rangle \longrightarrow \langle \Sigma, q; \pi, q : e_1, p : E[e_2] \rangle} \text{fork}$$

$$\frac{C_1 \equiv C'_1 \quad C'_1 \longrightarrow C'_2 \quad C'_2 \equiv C_2}{C_1 \longrightarrow C_2} \text{congr}$$

$$\frac{d \notin \Sigma}{\langle \Sigma; \pi, p : E[\nu(x_1, x_2).e] \rangle \longrightarrow \langle \Sigma, d; \pi, p : E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e] \rangle} \text{nu}$$

$$\frac{c_2 \rightsquigarrow c_1}{\langle \Sigma; \pi, p : E_1[\text{rd}(c_1, x.e)], q : E_2[\text{wr}(v, c_2)] \rangle \longrightarrow \langle \Sigma; \pi, p : E_1[(!v, c_1)_1/x]e, q : E_2[()] \rangle} \text{rw}$$

$$\frac{c \rightsquigarrow c_i \quad i \in \{1, 2\}}{\langle \Sigma; \pi, p : E_1[\text{ch}(c_1, x_1.e_1, c_2, x_2.e_2)], q : E_2[\text{wr}(v, c)] \rangle \longrightarrow \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_i]e_i, q : E_2[()] \rangle} \text{cw}$$

Figure 3.7: Configuration reduction rules.

$\boxed{e_1 \longrightarrow e_2}$  Expression  $e_1$  reduces to  $e_2$ .

$$\begin{array}{c}
\frac{}{\text{let}_\pi(v, x.e) \longrightarrow [v/x]e} \text{ let} \qquad \frac{}{((\lambda_\ell x. e) v)_\ell \longrightarrow [v/x]e} \text{ app} \\
\\
\frac{}{\text{split}_\ell((v_1, v_2)_\ell, x_1.x_2.e) \longrightarrow [v_1/x_1][v_2/x_2]e} \text{ split} \qquad \frac{}{\text{i}(!v) \longrightarrow v} \text{ gnab} \\
\\
\frac{}{\text{case}_\ell(\text{inj}_\ell^i(v), x_1.e_1, x_2.e_2) \longrightarrow [v/x_i]e_i} \text{ case} \qquad \frac{}{\text{fix}_\ell(x.e) \longrightarrow [\text{fix}_\ell(x.e)/x]e} \text{ fix}
\end{array}$$

Figure 3.8: Local reduction rules.



## CHAPTER 4: SAUCY

Using ILC, we build a concrete, executable implementation of a simplified UC framework, dubbed SaUCy. Then, we demonstrate the versatility of SaUCy in three ways:

1. We define a protocol composition operator and prove its associated composition theorem.
2. We walk through an instantiation of UC commitments.
3. We use ILC’s type system to reason about “reentrancy,” a subtle definitional issue in UC that has only recently been studied.

### 4.1 PROBABILISTIC POLYNOMIAL TIME IN ILC

The goal of cryptography reduction is to relate every bad event in a protocol to a *probabilistic polynomial time computation* that solves a hard problem. The ILC typing rules do not guarantee termination, let alone polynomial time normalization, so we must tackle this in metatheory. Also, since ILC is effectively deterministic (confluent), we will need to express random choices some other way. To meet these needs, we define a judgment about ILC terms that take a security parameter and a stream of random bits.

**Definition 4.1** (Polynomial time normalization). The judgment that  $e$  is polynomial time normalizable, written  $\text{PPT } e$ , is defined as follows:

$$\frac{\cdot; \cdot \vdash e : \text{Nat} \rightarrow [\text{Bit}] \rightarrow \text{Bit} \quad \forall k \in \text{Nat}. \forall r \in [\text{Bit}]^{\text{poly}(k)}. e \ k \ r \rightarrow^{\text{poly}(k)} v}{\text{PPT } e} \text{ ppt}$$

This says that if for all security parameters  $k$  and all bitstrings  $r$  (of length polynomial in  $k$ ) the term  $e \ k \ r$  normalizes to a value  $v$  in  $\text{poly}(k)$  steps, then  $\text{PPT } e$ .

Here, we have chosen a simple definition of polynomial time defined only for closed terms (i.e., an entire system of ITMs), and that requires polynomial time normalization for every choice of random bits, not just in expectation or with high probability.

We note that most UC variants use a more nuanced definition in which the individual ITM entities, such as the environment or protocol, can be judged polynomial time independently of their surrounding context [13, 14, 1]. Looking ahead to Section 4.3, this choice will constrain our definition of secure protocol emulation. Hofheinz et al. [15] give a detailed discussion of subtle issues arising with various polynomial time definitions and their consequences for

defining UC security. Regardless, the present notion suffices for our examples. We consider this issue complementary to the design of ILC itself, and adapting other notions of polynomial time to ILC as important future work. As an example, the polynomial time notion used in IITMs [14] relies on a distinction between “invited” and “uninvited” messages, which could be captured through refinement types à la the RCF calculus [16].

**Definition 4.2** (Value Distribution). Because processes are confluent, we know that if  $e \text{ } k \text{ } r \rightarrow^* v$ , then the value  $v$  is unique. We can therefore define the probability distribution ensemble  $D(e) = \{D_{e,k}\}_k$  based on a uniform distribution  $U_k$  over  $k$ -bit strings  $r$ , so the distribution  $D_{e,k}$  is given as

$$D_{e,k}(v) = \sum_{r \in R} U_k(r), \quad \text{for } R = \{r \mid e \text{ } k \text{ } r \rightarrow^* v\}.$$

**Definition 4.3** (Indistinguishability). What remains is to define a notion of indistinguishability for value distributions. However, we need to clarify when polynomial time normalization is an assumption or a proof obligation. To simplify things later, we define a partial order  $e_1 \leq e_2$ , which captures that  $e_2$  must be PPT if  $e_1$  is PPT, and if so, that their value distributions are statistically similar.

$$\frac{\text{PPT } e_1 \implies (\text{PPT } e_2 \text{ and } D(e_1) \sim D(e_2))}{e_1 \leq e_2} \text{ indist}$$

## 4.2 SAUCY EXECUTION MODEL

The implementation of SaUCy is centered around a definition of the UC execution model in ILC, presented in Figure 4.1. The function `execUC` takes as arguments an environment  $z$ , a pair of protocol processes  $(p, q)$ , a functionality  $f$ , an adversary  $a$ , a corruption model `crupt`, a security parameter  $k$ , and a random bitstring  $r$ . At a high level (ignoring corruption details for now), it runs each of the processes (allocating random bits to each of them) and connects channels as illustrated. The channels follow a uniform naming scheme. The read end of a channel is prefixed with `r-` and the write end of a channel is prefixed with `w-`. The channel `rZ2P` denotes the read end of communications from the environment  $z$  to the party  $p$ .

Next, we explain some of our main modeling choices and the consequences they have for the ILC implementation. To start with, we make several simplifications to standard UC, for example, focusing on the special case of two-party protocols (à la Simplified UC [17]).

```

execUC :: Az →w Ap × Aq → Af → Aa → Crupt → Nat → [Bit] → Bit
let execUC z (p,q) f a crupt k r =
  ν (rZ2P, wZ2P), (rP2Z, wP2Z)
  , (rZ2Q, wZ2Q), (rQ2Z, wQ2Z)
  , (rP2F, wP2F), (rF2P, wF2P)
  , (rQ2F, wQ2F), (rF2Q, wF2Q)
  , (rF2A, wF2A), (rA2F, wA2F)
  , (rA2Z, wA2Z), (rZ2A, wZ2A)
  , (rP2A, wP2A), (rA2P, wA2P)
  , (rQ2A, wQ2A), (rA2Q, wA2Q)
  , (rP2Q, wP2Q), (rQ2P, wQ2P)
  . let (rf,ra,rp,rq,rz) = splitBits r in
    f k rf crupt wF2P wF2Q wF2A rP2F rQ2F rA2F
    |▷ a k ra crupt wA2Z wA2F wA2P wA2Q rZ2A rF2A rP2A rQ2A
    |▷ corruptOrNot p k rp (crupt == CruptP) wP2Z wP2F wP2A wP2Q rZ2P rF2P rA2P rQ2P
    |▷ corruptOrNot q k rq (crupt == CruptQ) wQ2Z wQ2F wQ2A wQ2P rZ2Q rF2Q rA2Q rP2Q
    |▷ z k rz wZ2P wZ2Q wZ2A rP2Z rQ2Z rA2Z

```

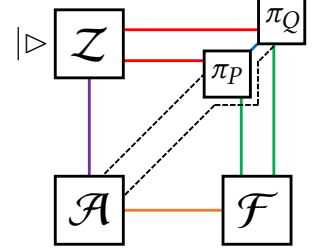


Figure 4.1: Full implementation of `execUC`.

We also only aim to show the case of *static* corruptions, in which the corrupt parties are determined at the onset. This is achieved by parameterizing the entire experiment by a value `crupt : Crupt` denoting which parties are corrupt (if any). The data type `Crupt` is defined as follows.

```
data Crupt = CruptP | CruptQ | CruptNone
```

For a more general model with adaptive corruptions, `execUC` would need to accept requests from the environment to add to the `crupt` list as the execution proceeds.

Our corruption model is Byzantine, meaning the adversary gets to exert complete control over the corrupted parties. For each party, depending on the value of `crupt`, either we run a copy of the honest party, or connect the channels to the adversary. This is implemented in the function `corruptOrNot`.

```

fwd :: ∀ a b . Wr a → Rd a → b
letrec fwd toR frS =
  let (!msg, frS) = rd frS in wr msg → toR ; fwd toR frS
corruptOrNot :: ∀ ... . Ap → Nat → [Bit] → Bool → ...
let corruptOrNot p k bits iscrupt toZ toF toA toQ
  frZ frF frA frQ =
  if iscrupt then

```

```

let _ = rd frZ in error "Z can't wr to corrupt"
|▷ fwd toA frF
|▷ fwd toA frQ
|▷ fwd toF frA
else
  p k bits toZ toF toQ frZ frF frQ

```

The `fwd` function simply forwards messages received on the read endpoint `frS` to the write endpoint `toR`. In `corruptOrNot`, if a party is corrupted, messages from the functionality and the other protocol party are forwarded to the adversary; messages from the adversary are forwarded to the functionality. Otherwise, the party is run as normal.

We also model a strong form of communication channels between the parties:  $P$  and  $Q$  are connected by a pair of raw ILC channels. Communication over these channels happens immediately, without activating the adversary or leaking even the existence of the message. In a more realistic model, the parties would only be able to communicate over a network channel modeled as a functionality,  $\mathcal{F}_{\text{SMT}}$  or  $\mathcal{F}_{\text{SYN}}$  [1]. Consequently our  $\mathcal{F}_{\text{COM}}$  functionality would need to be weakened by leaking some (model-specific) information about the message to the adversary.

A few important processes include the dummy party and the dummy adversary.

```

dummyP :: ∀ a b ... . Nat → [Bit] → Wr a → ... b
let dummyP k r toZ toF toQ frZ frF frQ = fwd toF frZ |▷ fwd toZ frF

```

The dummy party simply relays information between the environment and the functionality.

```

dummy :: ∀ a ... . Nat → [Bit] → Crupt → ... a
let dummyA k bits crupt toZ toF toP toQ toQasP frZ frF frP frQ toPasQ=
  let fwd2Z () c = loop (λ m . wr (X2Z m) → toZ) c in
    loop (λ x . match x with
      | A2F m ⇒ wr m → toF
      | A2P m ⇒ if crupt == CruptP
        then wr m → toQasP
        else wr m → toP) frZ
    |▷ fwd2Z () frF
    |▷ fwd2Z () frP
    |▷ fwd2Z () frQ

```

The dummy adversary forwards messages from the environment to either the functionality (if the message has constructor `A2F`) or the party `p` (if the message has constructor `A2P`).

Similarly, the dummy adversary forwards messages from the functionality or the protocol parties to the environment.

### 4.3 DEFINING UC SECURITY IN ILC

The central security definition in UC is protocol emulation. The guiding principle is that  $\pi$  emulates  $\phi$  if the environment cannot distinguish between the two protocols. Our first attempt is the following, where  $\mathcal{S}$  is the simulator that translates every attack in the real world into an attack expressed in the ideal world:

$$\frac{\forall \mathcal{Z}. \text{execUC } \mathcal{Z} \ \pi \ \mathcal{F}_1 \ \mathbb{1}_{\mathcal{A}} \leq \text{execUC } \mathcal{Z} \ \phi \ \mathcal{F}_2 \ \mathcal{S}}{\mathcal{S} \vdash (\pi, \mathcal{F}_1) \approx (\phi, \mathcal{F}_2)} \text{emulate}$$

To remark on a few notational choices: We make the functionality explicit, so emulation is a relationship between protocol-functionality pairs. Here,  $\mathbb{1}_{\mathcal{A}}$  is the dummy adversary, which just relays messages between the environment and the parties/functionality. We elide the standard dummy lemma that shows this is without loss of generality; the intuition is that whatever an adversary can do, the environment can achieve using  $\mathbb{1}_{\mathcal{A}}$ .

Unfortunately this simple definition turns out to be vacuous: a degenerate protocol  $\pi$  can emulate anything simply failing to be PPT, e.g., by diverging. To put it another way, the problem is the definition imposes a proof obligation on the simulator  $\mathcal{S}$  but not on  $\pi$ . What we want to say is that the real world protocol  $(\pi, \mathcal{F}_1)$  must be well behaved whenever the ideal world  $(\phi, \mathcal{F}_2)$  is. However, even a reasonable protocol can result in non-PPT executions if paired with a divergent environment. To solve this problem, we define protocol emulation by requiring a simulation in both directions, so every behavior in the ideal world must correspond to a behavior in the real world and vice versa.

**Definition 4.4** (Protocol Emulation). The judgment that one protocol-functionality pair  $(\pi, \mathcal{F}_1)$  securely emulates another  $(\phi, \mathcal{F}_2)$  (as proven by the simulators  $\mathcal{S}_{\mathcal{R}}, \mathcal{S}_{\mathcal{I}}$ ) is defined as

$$\frac{\begin{array}{c} \forall \mathcal{Z}. \text{execUC } \mathcal{Z} \ \phi \ \mathcal{F}_2 \ \mathbb{1}_{\mathcal{A}} \leq \text{execUC } \mathcal{Z} \ \pi \ \mathcal{F}_1 \ \mathcal{S}_{\mathcal{R}} \\ \text{execUC } \mathcal{Z} \ \pi \ \mathcal{F}_1 \ \mathbb{1}_{\mathcal{A}} \leq \text{execUC } \mathcal{Z} \ \phi \ \mathcal{F}_2 \ \mathcal{S}_{\mathcal{I}} \end{array}}{\mathcal{S}_{\mathcal{R}}, \mathcal{S}_{\mathcal{I}} \vdash (\pi, \mathcal{F}_1) \approx (\phi, \mathcal{F}_2)} \text{emulate}$$

We remark this definition goes against the UC convention of requiring simulation in one direction only. One direction is preferable intuitively because it should be fine if the protocol is even more secure than its specification. This does not pose any problem for our commitment

example; however, a protocol that leaks even less information than its ideal functionality requires would be impossible to prove secure under this definition. In any case, the benefit is this simplifies the polynomial time notion: vacuous protocols are clearly ruled out by the top condition, and both simulations are only required to be PPT when the environment  $\mathcal{Z}$  is well-behaved.

#### 4.4 A COMPOSITION THEOREM IN SAUCY

As a first demonstration of SaUCy, we work through the development of a composition operator, and give a theorem explaining its use.

**Definition 4.5** (UC realizes). To set out, we introduce the notation of “realizes,” which views a protocol as a way of instantiating a specification functionality  $\mathcal{F}_2$  from a setup assumption functionality  $\mathcal{F}_1$ ,

$$\frac{(\pi, \mathcal{F}_1) \approx (\text{id}_\pi, \mathcal{F}_2)}{\mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2} \text{ realizes}$$

where  $\text{id}_\pi$  is the *dummy protocol*, which simply relays messages between the environment and the functionality. This notation is convenient because it suggests a categorical approach to composition.

**Theorem 4.1** (Composition Theorem).

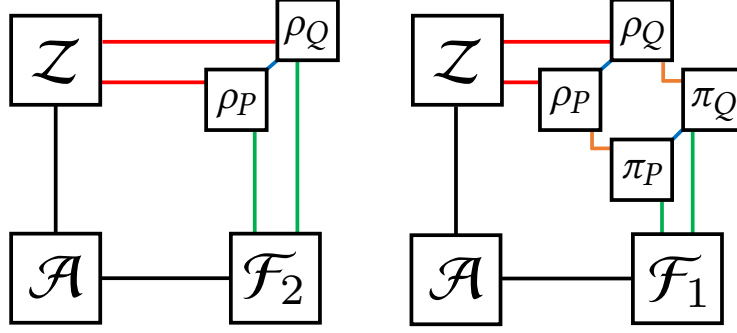
$$\frac{\mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2 \quad \mathcal{F}_2 \xrightarrow{\rho} \mathcal{F}_3}{\mathcal{F}_1 \xrightarrow{\rho \circ \pi} \mathcal{F}_3}$$

The idea is that the  $\rho \circ \pi$  can be defined in a natural way, where the ideal functionality channel of  $\rho$  is connected to the environment channel of  $\pi$ , as illustrated and defined in Figure 4.2.

*Proof.* To prove the theorem we construct the simulators  $S_{\mathcal{R},\rho} \circ S_{\mathcal{R},\pi}$  (respectively  $S_{\mathcal{I},\rho} \circ S_{\mathcal{I},\pi}$ ) in the natural way as well (see the appendix). Our proof obligation is to introduce an arbitrary environment  $\mathcal{Z}$  and conclude

$$\text{execUC } \mathcal{Z} (\rho \circ \pi) \mathcal{F}_1 \mathbb{1}_{\mathcal{A}} \leq \text{execUC } \mathcal{Z} \mathbb{1}_\pi \mathcal{F}_3 (S_{\mathcal{I},\rho} \circ S_{\mathcal{I},\pi}).$$

The main idea is to notice that that we can bring  $\rho$  from the composed protocol into the environment as  $(\mathcal{Z} \circ \rho)$ , reflecting the fact that the environment is meant to represent arbitrary



**let**  $(\circ) (\rho_P, \rho_Q) (\pi_P, \pi_Q)$   
 $\text{w}\rho_P 2Z \text{w}\rho_Q 2Z \text{w}\rho_P 2F \text{w}\rho_Q 2F$   
 $\text{w}\rho_P 2\rho_Q \text{w}\rho_Q 2\rho_P \text{r}Z 2\rho_P \text{r}Z 2\rho_Q$   
 $\text{r}F 2\rho_P \text{r}F 2\rho_Q \text{r}\rho_Q 2\rho_P \text{r}\rho_P 2\rho_Q =$   
 $\nu \dots \pi_P \text{w}\pi_P 2\rho_P \text{w}\rho_P 2F \text{w}\pi_P 2\pi_Q \text{r}\rho_P 2\pi_P \text{r}F 2\rho_P \text{r}\pi_Q 2\pi_P$   
 $\mid \triangleright \pi_Q \text{w}\pi_Q 2\rho_Q \text{w}\rho_Q 2F \text{w}\pi_Q 2\pi_P \text{r}\rho_Q 2\pi_Q \text{r}F 2\rho_Q \text{r}\pi_P 2\pi_Q$   
 $\mid \triangleright \rho_P \text{w}\rho_P 2Z \text{w}\rho_P 2\pi_P \text{w}\rho_P 2\rho_Q \text{r}Z 2\rho_P \text{r}\pi_P 2\rho_P \text{r}\rho_Q 2\rho_P$   
 $\mid \triangleright \rho_Q \text{w}\rho_Q 2Z \text{w}\rho_Q 2\pi_Q \text{w}\rho_Q 2\rho_P \text{r}Z 2\rho_Q \text{r}\pi_Q 2\rho_Q \text{r}\rho_P 2\rho_Q$

Figure 4.2: Protocol composition operator.

outer protocols. This transformation results in an equivalent term, given that ILC configurations are invariant to channel renaming and reordering of processes in a configuration (as in Section 3.4). The following derivation completes the proof:

$$\text{execUC } \mathcal{Z} (\rho \circ \pi) \mathcal{F}_1 \mathbb{1}_A \quad (4.1)$$

$$\equiv \text{execUC } (\mathcal{Z} \circ \rho) \pi \mathcal{F}_1 \mathbb{1}_A \quad (\text{By equivalence}) \quad (4.2)$$

$$\leq \text{execUC } (\mathcal{Z} \circ \rho) \text{id}_\pi \mathcal{F}_2 \mathcal{S}_{\mathcal{I},\pi} \quad (\text{From } \mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2) \quad (4.3)$$

$$\equiv \text{execUC } (\mathcal{S}_{\mathcal{I},\pi} \circ \mathcal{Z}) \rho \mathcal{F}_2 \mathbb{1}_A \quad (\text{By equivalence}) \quad (4.4)$$

$$\leq \text{execUC } (\mathcal{S}_{\mathcal{I},\pi} \circ \mathcal{Z}) \text{id}_\pi \mathcal{F}_3 \mathcal{S}_{\mathcal{I},\rho} \quad (\text{From } \mathcal{F}_2 \xrightarrow{\rho} \mathcal{F}_3) \quad (4.5)$$

$$\equiv \text{execUC } \mathcal{Z} \text{id}_\pi \mathcal{F}_3 (\mathcal{S}_{\mathcal{I},\pi} \circ \mathcal{S}_{\mathcal{I},\rho}) \quad (\text{By equivalence}) \quad (4.6)$$

The remaining case for  $\mathcal{S}_{\mathcal{R},\rho} \circ \mathcal{S}_{\mathcal{R},\pi}$  is symmetric. QED.

**Other notions of composition.** Our composition operator above is just a starting point. The “universal composition” [1] operator essentially multiplexes sessions identified by unique tags (*session ids*), while a joint state composition theorem collapses multiple subroutines into one [18]. Despite its name, development in UC often involves defining additional composition operators. For example, interesting composition often happens “in the functionality”

through higher order “wrapper” functionalities [19, 20] which we would express through abstraction. Some security properties require a generalized notion of ideal functionality that the environment can interact with directly. All the above motivate the development of the ILC core calculus as a flexible foundation; developing them in ILC is important future work.

#### 4.5 INSTANTIATING UC COMMITMENTS

Instantiation proofs in SaUCy follow a standard rhythm. We start with a security definition as an ideal functionality, give the protocol, construct a simulator, and finally complete the relational analysis on paper. To make this more concrete, we walk through an instantiation of UC commitments (à la Canetti and Fischlin [21]).

Commitment is an essential building block in many cryptographic protocols [22]. The idea behind commitment is simple: A *committer* provides a *receiver* with the digital equivalent of a “sealed envelope” containing some value that can later be revealed. The commitment scheme must be *hiding* in the sense that the commitment itself reveals no information about the committed value, and *binding* in the sense that the committer can only open the commitment to a single value. For security under composition, an additional *non-malleability* property is required, which roughly prevents an attacker from using one commitment to derive another related one.

While commitments are one of the simplest UC primitives, as a case study, this serves two main purposes. First, the proof demonstrates several representative UC techniques [8], in particular the simulator makes use of a “trusted setup” and extracts inputs from a corrupt sender. Second, the protocol makes use of computational primitives and thus requires a reduction step in the proof, which can go through because of ILC’s confluent design.

**Extending ILC with cryptographic primitives.** UC Commitments are realized from cryptographic primitives, such as trapdoor permutations, which require extensions to ILC. The new syntactic forms are **kgen**, **tdp**, **inv**, and **hc** with the static and dynamic semantics shown in Figure 4.3. The semantics are written in terms of the cryptographic objects themselves.

The key generation function **keygen** takes as input a random bitstring and outputs a random public key  $v_{pk}$  and a trapdoor  $v_{td}$ . The trapdoor permutation function **tdp** takes as inputs a key  $v_{pk}$  and a bitstring  $v_{in}$  and outputs a bitstring  $v_{out}$ . The **inv** function takes as inputs a key-trapdoor pair  $(v_{pk}, v_{td})$  and a bitstring  $v_{in}$  and outputs a bitstring  $v_{out}$ . The hardcore predicate function **hc** takes as input a key  $v_{pk}$  and outputs a single bit.



Expressions  $e ::= \text{kgen}(e) \mid \text{tdp}(e_1, e_2) \mid \text{inv}(e_1, e_2) \mid \text{hc}(e)$

$\boxed{\Delta; \Gamma \vdash e : U}$  Under affine context  $\Delta$  and unrestricted context  $\Gamma$ , expression  $e$  has type  $U$ .

$$\frac{\Delta; \Gamma \vdash e : [\text{Bit}]}{\Delta; \Gamma \vdash \text{kgen}(e) : [\text{Bit}] \times [\text{Bit}]} \text{kgen} \quad \frac{\Delta_1; \Gamma \vdash e_1 : [\text{Bit}] \quad \Delta_2; \Gamma \vdash e_2 : [\text{Bit}]}{\Delta_1, \Delta_2; \Gamma \vdash \text{tdp}(e_1, e_2) : [\text{Bit}]} \text{tdp}$$

$$\frac{\Delta_1; \Gamma \vdash e_1 : [\text{Bit}] \times [\text{Bit}] \quad \Delta_2; \Gamma \vdash e_2 : [\text{Bit}]}{\Delta_1, \Delta_2; \Gamma \vdash \text{inv}(e_1, e_2) : [\text{Bit}]} \text{inv} \quad \frac{\Delta; \Gamma \vdash e : [\text{Bit}] \rightarrow \text{Bit}}{\Delta; \Gamma \vdash \text{hc}(e) : \text{Bit}} \text{hc}$$

$\boxed{e_1 \longrightarrow e_2}$  Expression  $e_1$  reduces to  $e_2$ .

$$\frac{\mathbf{Gen}(v_r) = (v_{pk}, v_{td})_\infty \quad v_{pk}, v_{td} \in \{0, 1\}^k}{\text{kgen}(v_r) \longrightarrow (v_{pk}, v_{td})_\infty} \text{kgen}$$

$$\frac{\mathbf{f}(v_{pk}, v_{in}) = v_{out} \quad \mathbf{f}: \{0, 1\}^k \rightarrow \{0, 1\}^k \rightarrow \{0, 1\}^k}{\text{tdp}(v_{pk}, v_{in}) \longrightarrow v_{out}} \text{tdp}$$

$$\frac{\mathbf{Inv}((v_{pk}, v_{td})_\infty, v_{in}) = v_{out} \quad \mathbf{Inv}: \{0, 1\}^k \times \{0, 1\}^k \rightarrow \{0, 1\}^k \rightarrow \{0, 1\}^k}{\text{inv}((v_{pk}, v_{td})_\infty, v_{in}) \longrightarrow v_{out}} \text{inv}$$

$$\frac{\mathbf{B}(v_{pk}) = v \quad \mathbf{B}: \{0, 1\}^k \rightarrow \{0, 1\}}{\text{hc}(v_{pk}) \longrightarrow v} \text{hc}$$

Figure 4.3: Extending ILC with trapdoor permutations. The semantics are parameterized by a security parameter  $k$ .

The UC commitment protocol makes use of a cryptographic primitive, namely a trapdoor pseudorandom generator. This is provided by extending ILC with new syntactic forms, along with their static and dynamic semantics.

We can use these to implement a special pseudorandom number generator  $G_{pk}: \{0, 1\}^k \rightarrow \{0, 1\}^{4k}$  that has a trapdoor property, i.e., it is easy to compute, but difficult to invert except with special information called the “trapdoor.”

$$G_{pk}(r) = (\mathbf{f}_{pk}^{(3n)}(r), \mathbf{B}(\mathbf{f}_{pk}^{(3n-1)}(r)), \dots, \mathbf{B}(\mathbf{f}_{pk}(r)), \mathbf{B}(r)) \quad (4.7)$$

Here,  $\mathbf{f}_{pk}$  is a trapdoor permutation over  $\{0, 1\}^k$ , with  $\mathbf{f}_{pk}^{(i)}(r)$  denoting the  $i^{\text{th}}$ -fold application of  $\mathbf{f}_{pk}$ , and  $\mathbf{B}$  is a hardcore predicate for  $\mathbf{f}_{pk}$ . In ILC, this can be implemented as:

$\text{iterate} :: \forall a . \text{Int} \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a$

```

prg :: [Bit] → [Bit] → Nat → [Bit]
let prg pk r k =
  letrec aux j =
    if j ≤ 0 then [hc r]
    else hc (iterate j (tdp pk) r) : aux pk r (j - 1) in
  iterate (3 * k) (tdp pk) r ++ aux pk r (3 * k - 1)

```

While in a symbolic setting we would instantiate these with algebraic data, in ILC we give the stepping rule in terms of an arbitrary pseudorandom function family, i.e., the actual computational definition. This can be instantiated concretely for execution (e.g., with an RSA-based function) or treated abstractly in the metatheory when we get to the reduction step of the proof.

The commitment protocol also relies on a “trusted setup,” or common reference string (CRS), which is essentially public parameters generated ahead of time. The common reference string is modeled as an ideal functionality  $\mathcal{F}_{\text{CRS}}$ .

**Commitment Protocol.** We now give the full elaboration of our UC commitment instantiation. The specification functionality is as follows:

```

fCom :: Nat → [Bit] → Crupt → ... →R 1
let fCom k bits crupt toP toQ toA frP frQ frA =
  let (! (Commit b), frP) = rd frP in
    wr Receipt → toQ ;
  let (! Open, frP) = rd frP in
    wr (Opened b) → toQ

```

The functionality simply waits for the committer  $P$  to commit to some bit  $b$ , notifies the receiver  $Q$  that it has taken place, and reveals  $b$  to  $Q$  upon request by  $P$ . Notice that  $Q$  never actually sees a commitment to  $b$  (only the (Receipt) message), so the three properties (hiding, binding, and non-malleability) hold trivially.

Our development follows closely from the psuedocode in the UC literature [21], which we show here in Protocol 4.1. In ILC, the committer and receiver are defined like so:

```

committer :: ∀ ... . Nat → [Bit] → ... → 1
let committer k bits crupt toZ toF toQ frZ frF frQ =
  let (! (Commit b), frZ) = rd frZ in
    wr GetCRS → toF ;
  let (! (PublicStrings  $\sigma$  pk0pk1), frF) = rd frF in
    let r = take k bits in

```

---

**Protocol 4.1:** Universally Composable Commitment

---

- 1 Public strings:
  - 2  $\sigma$ : Random string in  $\{0, 1\}^{4n}$
  - 3  $pk_0, pk_1$ : Keys for generator  $G_k: \{0, 1\}^n \rightarrow \{0, 1\}^{4n}$
  - 4 **Commit**( $b$ ):
  - 5  $r \leftarrow \{0, 1\}^n$
  - 6  $y := G_{pk_b}(r)$
  - 7 if  $b = 1$  then  $y := y \oplus \sigma$
  - 8 Send (**Commit**,  $y$ ) to receiver.
  - 9 Upon receiving (**Commit**,  $y$ ) from  $A$ ,  $B$  outputs (**Receipt**).
  - 10 **Decommit**( $x$ ):
  - 11 Send ( $b, r$ ) to receiver.
  - 12 Receiver checks  $y = G_{pk_b}(r)$  for  $b = 0$ , or  $y = G_{pk_b}(r) \oplus \sigma$  for  $b = 1$ . If verification succeeds, then  $B$  outputs (**Open**,  $b$ ).
- 

```
let x = if b == 0 then prg pk0 r
      else xors (prg pk1 r)  $\sigma$  in
wr Commit' x  $\rightarrow$  toQ ;
let (!Open, frZ) = rd frZ in
wr (Open' b r)  $\rightarrow$  toQ
```

```
receiver ::  $\forall \dots . \text{Nat} \rightarrow [\text{Bit}] \rightarrow \dots \multimap \mathbb{1}$ 
let receiver k bits crupt toZ toF toP frZ frF frP =
  let (!(Commit' x), frP) = rd frP in
    wr GetCRS  $\rightarrow$  toF ;
    let (!(PublicStrings  $\sigma$  pk0pk1), frF) = rd frF in
      wr Receipt  $\rightarrow$  toZ ;
      let (!(Open' b r), frP) = rd frP in
        if (b == 0 && x == prg pk0 r) ||
          (b == 1 && x == xors (prg pk1 r)  $\sigma$ )
        then wr (Opened b)  $\rightarrow$  toZ
        else error "Cannot occur in honest case."
```

The protocol also relies on the CRS functionality:

```
fCrs ::  $\forall a \dots . \text{Nat} \rightarrow [\text{Bit}] \rightarrow \text{Crupt} \rightarrow \dots a$ 
let fCrs k bits crupt toP toQ toA frP frQ frA =
  let ( $\sigma$ , bits) = sample (4*k) bits in
  let (r0, bits) = sample k bits in
```

```

let (r1, bits) = sample k bits in
let pk0 = kgen k r0 in
let pk1 = kgen k r1 in
let pub = PublicStrings σ pk0 pk1 in
let replyCrs to fr = loop (λ _ . wr pub → to) fr in
  replyCrs toP frP
  |▷ replyCrs toQ frQ
  |▷ replyCrs toA frA

```

To briefly summarize what is going on: the setup CRS samples a random string  $\sigma$  and two trapdoor pseudorandom generators (prgs  $\mathbf{pk}_0, \mathbf{pk}_1$ ). To commit to the bit  $b$ , the committer produces a string  $y$  that is the result of applying one or the other of the prgs, and if  $b = 1$  additionally applying xor with  $\sigma$ . The intuitive explanation why this is hiding is that without the trapdoor, it is difficult to tell whether a random  $4k$ -bit string is in the range of either prg. To open the commitment, the committer simply reveals the preimage and the receiver checks which of the two cases applies. The intuitive explanation why this is binding is that it is difficult to find a pair  $y, y \oplus \sigma$  that are respectively in the range of both prgs.

**Defining the simulator.** The UC proof consists of two simulators, one for the ideal world and one for the real world. The ideal world simulator is ported directly from the UC literature [21]:

```

siml :: Nat → [Bit] → Crupt → ... → 1
let siml k bits crupt toZ toF toP toQ frZ frF frP frQ =
  let (pk0, td0) = kgen k in
  let (pk1, td1) = kgen k in
  let (r0, bits) = sample k bits in
  let (r1, bits) = sample k bits in
  let σ = xors (prg pk0 r0) (prg pk1 r1) in
  match crupt with
  | CruptP ⇒
    let (!GetCRS, frZ) = rd frZ in
    wr (X2Z (PublicStrings σ pk0 pk1)) → toZ ;
    let (!A2P (Commit' y), frZ) = rd frZ in
    if check td0 pk0 y then
      wr (Commit 0) → toP
    else

```

```

    if check td1 pk1 (xors y σ) then
      wr (Commit 1) → toP
    else error "Fail" ;
  let (! (A2P (Open' b r)), frZ) = rd frZ in
    if b == 0 && y == prg pk0 r ||
      b == 1 && y == xors (prg pk1 r) σ
    then wr Open → toP
    else error "Fail"
| CruptQ ⇒
  let (!GetCRS, frZ) = rd frZ in
    wr (X2Z (PublicStrings σ pk0 pk1)) → toZ ;
  let (!Receipt, frQ) = rd frQ in
    let y = prg pk0 r0 in
      wr (X2Z (Commit' y)) → toZ ;
    let (! (Opened b'), frQ) = rd frQ in
      if (b' == 0) then
        wr (X2Z (Opened' r0)) → toZ
      else
        wr (X2Z (Opened' r1)) → toZ
| CruptNone ⇒ error "Fail"

```

On the other hand, the non-standard real world simulator is required because our protocol emulation definition requires simulation in both directions.

$\text{simR} :: \text{Nat} \rightarrow [\text{Bit}] \rightarrow \text{Crupt} \rightarrow \dots \multimap \mathbb{1}$

**let**  $\text{simR } k \text{ bits } \text{crupt } \text{toZ } \text{toF } \text{toP } \text{toQ } \text{frZ } \text{frF } \text{frP } \text{frQ} =$

**match**  $\text{crupt}$  **with**

|  $\text{CruptP} \Rightarrow$

```

  let (! (Commit b), frZ) = rd frZ in
    wr GetCRS → toF ;
  let (! (PublicStrings σ pk0 pk1), frF) = rd frF in
    let r = take k bits in
      let y = if b == 0 then prg pk0 r else xors (prg pk0 r) σ in
        wr (Commit' y) → toQ ;
    let (! (Open), frZ) = rd frZ in
      wr (Open' b r) → toQ

```

|  $\text{CruptQ} \Rightarrow$

```

let (!(Commit'  $y$ ),  $\text{frQ}$ ) = rd  $\text{frQ}$  in
  wr Receipt  $\rightarrow \text{toZ}$  ;
  let (!(Open'  $b$   $r$ ),  $\text{frQ}$ ) = rd  $\text{frQ}$  in
    wr (Opened  $b$ )  $\rightarrow \text{toZ}$ 
| CruptNone  $\Rightarrow$  error "Fail"

```

The key to the ideal world simulator is to allow the simulator to generate its own “fake” CRS, for which it stores the trapdoors. The string  $\sigma$  is not truly random, but instead is the result of combining two evaluations of the prgs. The ideal world simulator consists of two cases, depending on which of the parties is corrupt.

In the case that the committer  $P$  is corrupt, the simulator needs to be able to *extract* the committed value. The simulator is activated when  $\mathcal{Z}$  sends a message (**Commit**'  $y$ ); in the real world, this is relayed by the dummy adversary to  $Q$ , who outputs **Committed** back to the environment. Hence to achieve the same effect in the ideal world, the simulator must send (**Commit**  $b$ ) to  $\mathcal{F}_{\text{COM}}$ . To extract  $b$  from  $y$ , the simulator makes use of the prg trapdoor check which one has  $y$  in its range. It is necessary to argue by cryptographic reduction that this simulation is sound. To show this, we would define an alternative execution where the prg is substituted for a truly random function (i.e., a random oracle). If an environment  $\mathcal{Z}$  could distinguish between these two worlds, then we could adapt the execution to distinguish the prg from random, violating the prg assumption.

In the case that the receiver  $Q$  is corrupt, the simulator needs to *equivocate*. The simulator is activated when  $\mathcal{Z}$  inputs (**Commit**  $b$ ) to  $P$ , after which  $\mathcal{F}_{\text{COM}}$  sends **Committed** to the simulator. In the real world, the environment receives a commitment message (**Commit**'  $y$ ) from corrupted  $Q$  for some seemingly-random  $y$ . To achieve the same effect, the simulator must choose  $y$ . However, the simulator is next activated when the  $\mathcal{Z}$  inputs (**Open**  $b$ ) to  $P$ , after which the simulator learns  $b$  from  $\mathcal{F}_{\text{COM}}$ . However, in the real world the environment receives a valid opening (**Opened**'  $b$   $r$ ) that is consistent with  $y$  and with the value chosen by the environment. Thus the simulator must initially choose  $y$  so that it can later be opened to either value  $b$  may take. The simulator achieves this by choosing  $\sigma$  and  $y$  ahead of time while generating the fake CRS. The reduction step is the same, and involves replacing prg with a true random function.

Recall that the motivation for the real world simulator is to rule out degenerate protocols that diverge in some way. For every well behaved environment such that the ideal world is PPT, we need to demonstrate an adversary in the real world that is also PPT. Fortunately, the real world simulator is much simpler than ideal world simulator. Essentially the simulator runs a copy of the honest protocol for each of the corrupted parties. The simulation that

results in this case is identical.

**Relational argument.** The goal of the relational analysis is to show that an environment's output in the real world is indistinguishable from its output in the ideal world. The proof follows the one in Canetti and Fischlin [21].

*Proof Sketch.* Consider the following ensembles:

$$\begin{aligned} D_{\mathcal{R}} &= D(\text{execUC } \mathcal{Z} \text{ (committer, receiver) fCrs dummyA}) \\ D'_{\mathcal{R}} &= D(\text{execUC } \mathcal{Z} \text{ (committer, receiver) bCrs dummyA}) \\ D_{\mathcal{I}} &= D(\text{execUC } \mathcal{Z} \text{ (dummyP, dummyQ) fCom siml}) \end{aligned}$$

The ensemble  $D_{\mathcal{R}}$  is over the output of  $\mathcal{Z}$  in a real world execution. The ensemble  $D'_{\mathcal{R}}$  is similar, except  $\mathcal{Z}$  runs with a bad functionality **bCrs** that computes fake public strings in the same way that the simulator does.

```

bCrs :: ∀ a ... . Nat → [Bit] → Crupt → ... a
let bCrs k bits crupt toP toQ toA frP frQ frA =
  let (r0, bits) = sample k bits in
  let (r1, bits) = sample k bits in
  let pk0 = kgen k r0 in
  let pk1 = kgen k r1 in
  let σ = xors (prg pk0 r0) (prg pk1 r1)
  let pub = PublicStrings σ pk0 pk1 in
  let replyCrs to fr = loop (λ _ . wr pub → to) fr in
    replyCrs toP frP
    |▷ replyCrs toQ frQ
    |▷ replyCrs toA frA

```

The ensemble  $D_{\mathcal{I}}$  is over the output of  $\mathcal{Z}$  in an ideal world execution. The goal is to show that  $D_{\mathcal{R}} \sim D_{\mathcal{I}}$ . The proof proceeds by first showing that breaking the pseudorandomness of the PRG reduces to distinguishing between  $D_{\mathcal{R}}$  and  $D'_{\mathcal{R}}$  (hence,  $D_{\mathcal{R}} \sim D'_{\mathcal{R}}$ ), and then by showing that breaking the pseudorandomness of the PRG also reduces to distinguishing between  $D'_{\mathcal{R}}$  and  $D_{\mathcal{I}}$  (hence,  $D'_{\mathcal{R}} \sim D_{\mathcal{I}}$ ). By the transitivity of indistinguishability, we have that  $D_{\mathcal{R}} \sim D_{\mathcal{I}}$ . QED.

Here, ILC's confluence property plays a critical role: It is necessary for defining the probability ensembles  $D_{\mathcal{R}}$ ,  $D'_{\mathcal{R}}$ , and  $D_{\mathcal{I}}$ , without which we would not be able to obtain a reduction

from some computationally hard problem to distinguishing the real world and ideal world ensembles.

#### 4.6 REENTRANCY IN SAUCY

Camenisch et al. [23] recently identified subtleties in defining UC ideal functionalities (related to reentrancy and the scheduling of concurrent code) such that several functionalities in the literature are ambiguous as ITMs. Although concerning, these issues have no cryptographic flavor, and so they are better addressed from a PL standpoint. To illustrate, consider the following (untypeable) ILC process **reentrantF**, which allows an adversary  $\mathcal{A}$  to control the delivery schedule of messages from  $P$  to  $Q$  (i.e., an asynchronous channel):

```

loop ::  $\forall a\ b. (a \rightarrow b) \rightarrow \text{Rd } a \multimap b$ 
letrec loop f frS = let (!v, frS) = rd frS in f v; loop f frS
let reentrantF ... frP frA =
  loop ( $\lambda \text{msg}. (\text{let } (!\text{Ok}, \text{frA}) = \text{rd } \text{frA} \text{ in } \text{wr } \text{msg} \rightarrow \text{toQ})$ 
     $\triangleright \text{wr } \text{msg} \rightarrow \text{toA}) \text{frP}$ 

```

After receiving input from party  $P$ , it notifies the adversary, then forks a background thread to wait for **Ok** before delivering the message. This introduces a race condition: Suppose input message  $m_1$  is sent by  $P$ , but then  $\mathcal{A}$ , before sending **Ok**, instead returns control to  $\mathcal{Z}$ , which passes  $P$  a second input  $m_2$ . Now there are two queued messages. Which one gets delivered when the adversary sends **Ok**?

To resolve this issue, notice that **reentrantF** is untypeable in ILC. The race condition occurs because the read endpoint **frA** is duplicated (appears free in an unrestricted function). Camenisch et al. [23] identified several strategies for resolving this problem in UC, which in turn are expressible ILC. One approach is to make the process explicitly sequential, such that the arrival of a second message before the first is delivered causes execution to get stuck:

```

letrec sequentialF ... frP frA =
  let (!msg, frP) = rd frP in
    wr msg  $\rightarrow \text{toA}$  ;
    let (!Ok, frA) = rd frA in
      wr msg  $\rightarrow \text{toQ}$  ;
      sequentialF ... frP frA

```

Alternatively, we may discard such messages arriving out of order, returning them to sender; we express this in ILC using the external choice operator:



```

letrec discardingF ... frP frA =
  let (!msg, frP) = rd frP in
    wr msg → toA ;
  letrec iloop () frP frA =
    choice
    | (⊥, frP, frA) @ (rd frP) ⇒ wr Discard → toP ;
                                iloop () frP frA
    | (⊥, frP, frA) @ (rd frA) ⇒ wr msg → toQ ;
                                discardingF ... frP frA
  in iloop () frP frA

```

Ultimately, Camenisch et al. propose a different strategy, which is to restrict how the environment/adversary respond to certain “urgent” messages that are used to exchange meta-information (modeling related messages). That is, upon receiving an urgent message from process  $P$ , the environment (or adversary) must return control back to  $P$  immediately. Modeling this solution is left as future work, but ILC provides an ideal starting point—restrictions on the environment/adversary could be expressed by behavior refinements: upon receiving an urgent message from  $P$ , the environment (or adversary) must not send a message on its other channels before sending a message to  $P$ .

## CHAPTER 5: RELATED WORK

### 5.1 PROCESS CALCULI

Process calculi have a long and rich history. ILC occupies a point in this space that is particularly suited to faithfully capturing interactive Turing machines (and hence, computational cryptography), but plenty of existing calculi are also cryptographically-flavored and/or enjoy similar properties to ILC. We survey some of them here.

**With symbolic semantics.** Two early adaptations of process calculi for reasoning about cryptographic protocols were the spi calculus [6] and the applied  $\pi$ -calculus [9], both of which extend the  $\pi$ -calculus with cryptographic operations [7]. Symbolic UC [4] is a simulation-based security framework in this setting. However, protocols proven secure in the symbolic setting may not be realizable with any cryptographic primitives based on hardness assumptions.

**With computational semantics.** Naturally, ensuing work has turned to bridging the gap between this PL-style of formalization and the computational model of cryptography by outfitting these calculi with a computational semantics. Lincoln et al. [10] give a computational semantics to a variant of the  $\pi$ -calculus, which allows one to define communicating probabilistic polynomial-time processes; Mateus et al. [24] adapts their calculus to explore (sequential) compositionality properties in protocols. A drawback of these protocols is that they embed probabilistic choices directly into the definition—essentially when faced with nondeterminism, each path has equal probability. Laud [25] gives a computational semantics to the spi calculus, which additionally includes a type system for ensuring well-typed protocols preserve the secrecy of messages given to it by users.

**With confluence.** There are a number of other process calculi that enjoy confluence. Berger et al. [26] describe a type system for capturing deterministic (sequential) computation in the  $\pi$ -calculus. The type system uses affineness and stateless replication to achieve deterministic computation. Fowler et al. [12] present a core linear lambda calculus with (binary) session-typed channels and exception handling that enjoys confluence and termination. The calculus only considers two-party protocols, so for our multiparty setting, ILC requires a sophisticated type system to achieve confluence.

## 5.2 TOOLS FOR CRYPTOGRAPHIC ANALYSIS

Computer-aided tools for cryptographic analysis operate in either the symbolic model or the computational model. The survey by Blanchet [27] highlights some of their differences.

The symbolic tools include the NRL protocol analyzer [28], Maude-NPA [29], Proverif [30], and Tamarin [31]. In the symbolic setting, cryptographic operations are abstracted as term algebras (a variant of the applied  $\pi$ -calculus in the case of Proverif), and adversary capabilities are nondeterministic applications of deduction rules over these terms. Here, nondeterminism allows the adversary to find attack traces (if there are any), whereas the presence of nondeterminism in the computational setting would frustrate cryptographic reduction proofs.

The computational tools include CertiCrypt [32], EasyCrypt [5], CryptoVerif [33], and CryptHOL [34]. Although these tools focus on game-based security, which, in contrast to simulation-based definitions (such as UC), only guarantee security in a standalone setting (no composition guarantees), recent efforts have used them to mechanize UC-based proofs. EasyUC [35] uses EasyCrypt to mechanize UC proofs for key exchange and secure communication. Lochbihler et al. [36] use CryptHOL to formalize the constructive cryptography framework, an alternative formulation of universally composable security [37].

## 5.3 VARIATIONS OF UNIVERSAL COMPOSABILITY

A number of models for universal composability have been proposed in the literature [1, 38, 18, 14, 39, 40, 37, 17, 4, 13, 23, 41, 42]. We highlight a few that have similar goals to ours.

In contrast with UC, which uses ITMs as its computational model, the reactive simulatability framework (RSIM) [14] uses probabilistic IO automata, which are amenable to automated reasoning. In contrast with RSIM, ILC is intended to be the basis for a convenient and flexible programming language to which we can easily port existing UC pseudocode.

Models based on inexhaustible interactive Turing machines (IITMs) [43, 41] aim to address drawbacks of UC models for which polynomial time ITMs can be “exhausted” (by having other machines send useless messages, forcing them to halt). In turn, models with exhaustible ITMs are less expressive. Because IITMs maintain the “single-threaded” execution semantics of ITMs, ILC can be used to build a concrete programming model for IITM-based frameworks as well.

The abstract cryptography framework [40] advocates a top-down approach: developing theory at an abstract level (ignoring low level details such as computational models and

complexity notions) to simplify definitions. While we stick to a bottom-up approach, we aim to simplify UC via PL formalisms.

Simplified universal composability (SUC) [17] gives a simpler and restricted variant of the UC framework. The main difference from vanilla UC [1] is that the set of parties is fixed, which greatly simplifies polynomial time reasoning and protocol composition while maintaining the same strong properties. We follow this in our `execUC` implementation.

## CHAPTER 6: CONCLUSION

The universal composability (UC) framework is widely used in cryptography for proofs. SaUCy takes a step towards mechanizing UC as a programming framework for constructing and analyzing large systems. We envision using SaUCy to tackle, for example, applications involving blockchains and smart contracts [44, 45, 46], which comprise an array of cryptography and distributed computing components and suffer from increasingly unwieldy formalisms.

We can view ILC typechecking of simulators in SaUCy as a partial mechanization of UC proofs, though the indistinguishability analysis is still on paper. Even partial mechanization is useful for catching bugs; we imagine using SaUCy to systematically implement functionalities and protocols from the literature and fuzz test them. Future work would be to embed ILC within a mechanized proof system, such as  $F^*$  or EasyCrypt.

## APPENDIX A: INTERACTIVE LAMBDA CALCULUS

### A.1 ALGORITHMIC TYPING RULES

$\Delta_{in}; \Gamma \vdash e : U \dashv \Delta_{out}$  Under input contexts  $\Delta_{in}$  and  $\Gamma$ , expression  $e$  has type  $U$  and output context  $\Delta_{out}$ .

$$\begin{array}{c}
\overline{\Delta; \Gamma, x : A \vdash x : A \dashv \Delta} \text{ uvar} \qquad \overline{\Delta, x : X; \Gamma \vdash x : X \dashv \Delta} \text{ avar} \qquad \overline{\Delta; \Gamma \vdash () : \mathbb{1} \dashv \Delta} \text{ unit} \\
\\
\frac{\Delta_1; \Gamma \vdash e_1 : A_1 \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_2 : A_2 \dashv \Delta_3}{\Delta_1; \Gamma \vdash (e_1, e_2)_\infty : A_1 \times A_2 \dashv \Delta_3} \text{ upair} \qquad \frac{\Delta_1; \Gamma \vdash e_1 : S_1 \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_2 : S_2 \dashv \Delta_3}{\Delta_1; \Gamma \vdash (e_1, e_2)_w : S_1 \times S_2 \dashv \Delta_3} \text{ spair} \\
\\
\frac{\Delta_1; \Gamma \vdash e_1 : X_1 \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_2 : X_2 \dashv \Delta_3}{\Delta_1; \Gamma \vdash (e_1, e_2)_1 : X_1 \otimes X_2 \dashv \Delta_3} \text{ apair} \qquad \frac{i \in \{1, 2\} \quad \Delta_1; \Gamma \vdash e : A_i \dashv \Delta_2}{\Delta_1; \Gamma \vdash \text{inj}_\infty^i(e) : A_1 + A_2 \dashv \Delta_2} \text{ uinj} \\
\\
\frac{i \in \{1, 2\} \quad \Delta_1; \Gamma \vdash e : S_i \dashv \Delta_2}{\Delta_1; \Gamma \vdash \text{inj}_w^i(e) : S_1 + S_2 \dashv \Delta_2} \text{ sinj} \qquad \frac{i \in \{1, 2\} \quad \Delta_1; \Gamma \vdash e : X_i \dashv \Delta_2}{\Delta_1; \Gamma \vdash \text{inj}_1^i(e) : X_1 \oplus X_2 \dashv \Delta_2} \text{ ainj} \\
\\
\frac{\Delta_1; \Gamma \vdash e_1 : A_1 \times A_2 \dashv \Delta_2 \quad \Delta_2; \Gamma, x_1 : A_1, x_2 : A_2 \vdash e : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \text{split}_\infty(e_1, x_1.x_2.e_2) : U \dashv \Delta_3} \text{ usplit} \qquad \frac{\Delta_1; \Gamma \vdash e_1 : S_1 \times S_2 \dashv \Delta_2 \quad \Delta_2; \Gamma, x_1 : S_1, x_2 : S_2 \vdash e : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \text{split}_w(e_1, x_1.x_2.e_2) : U \dashv \Delta_3} \text{ ssplit} \\
\\
\frac{\Delta_1; \Gamma \vdash e_1 : X_1 \otimes X_2 \dashv \Delta_2 \quad \Delta_2, x_1 : X_1, x_2 : X_2; \Gamma \vdash e : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \text{split}_1(e_1, x_1.x_2.e_2) : U \dashv \Delta_3 \div (x_1 : X_1, x_2 : X_2)} \text{ asplit} \\
\\
\frac{\Delta_1; \Gamma \vdash e : A_1 + A_2 \dashv \Delta_2 \quad \Delta_2; \Gamma, x_1 : A_1 \vdash e_1 : U \dashv \Delta_3 \quad \Delta_2; \Gamma, x_2 : A_2 \vdash e_2 : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \text{case}_\infty(e, x_1.e_1, x_2.e_2) : U \dashv \Delta_3} \text{ ucase}
\end{array}$$

Figure A.1: Algorithmic typing rules.

$$\begin{array}{c}
\frac{\Delta_1; \Gamma \vdash e : S_1 + S_2 \dashv \Delta_2 \quad \Delta_2; \Gamma, x_1 : S_1 \vdash e_1 : U \dashv \Delta_3 \quad \Delta_2; \Gamma, x_2 : S_2 \vdash e_2 : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \mathbf{case}_w(e, x_1.e_1, x_2.e_2) : U \dashv \Delta_3} \text{scase} \\
\\
\frac{\Delta_1; \Gamma \vdash e : X_1 \oplus X_2 \dashv \Delta_2 \quad \Delta_2, x_1 : X_1; \Gamma \vdash e_1 : U \dashv \Delta_3 \quad \Delta_2, x_2 : X_2; \Gamma \vdash e_2 : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \mathbf{case}_1(e, x_1.e_1, x_2.e_2) : U \dashv \Delta_3 \div (x_1 : X_1, x_2 : X_2)} \text{acase} \\
\\
\frac{.; \Gamma, x : A \vdash e : U \dashv \cdot}{\Delta; \Gamma \vdash \lambda_\infty x. e : A \rightarrow_\infty U \dashv \Delta} \text{uabs} \quad \frac{\textcircled{w}; \Gamma, x : A \vdash e : U \dashv \cdot}{\Delta; \Gamma \vdash \lambda_w x. e : A \rightarrow_w U \dashv \Delta} \text{wabs} \\
\\
\frac{\Delta_1, x : X; \Gamma \vdash e : U \dashv \Delta_2}{\Delta_1; \Gamma \vdash \lambda_1 x. e : X \rightarrow_1 U \dashv \Delta_2 \div (x : X)} \text{aabs} \quad \frac{\Delta_1; \Gamma \vdash e_2 : A \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_1 : A \rightarrow_\infty U \dashv \Delta_3}{\Delta_1; \Gamma \vdash (e_1 e_2)_\infty : U \dashv \Delta_3} \text{uapp} \\
\\
\frac{\Delta_1; \Gamma \vdash e_2 : A \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_1 : A \rightarrow_w U \dashv \Delta_3}{\Delta_1, \textcircled{w}; \Gamma \vdash (e_1 e_2)_w : U \dashv \Delta_3} \text{wapp} \quad \frac{\Delta_1; \Gamma \vdash e_2 : X \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_1 : X \rightarrow_1 U \dashv \Delta_3}{\Delta_1; \Gamma \vdash (e_1 e_2)_1 : U \dashv \Delta_3} \text{aapp} \\
\\
\frac{.; \Gamma, x : A \rightarrow_\infty U \vdash e : A \rightarrow_\infty U \dashv \cdot}{\Delta; \Gamma \vdash \mathbf{fix}_\infty(x.e) : A \rightarrow_\infty U \dashv \Delta} \text{ufix} \quad \frac{.; \Gamma, x : A \rightarrow_w U \vdash e : A \rightarrow_w U \dashv \cdot}{\Delta; \Gamma \vdash \mathbf{fix}_w(x.e) : A \rightarrow_w U \dashv \Delta} \text{wfix} \\
\\
\frac{x : X \rightarrow_1 U; \Gamma \vdash e : X \rightarrow_1 U \dashv \cdot}{\Delta; \Gamma \vdash \mathbf{fix}_1(x.e) : X \rightarrow_1 U \dashv \Delta} \text{afix} \quad \frac{\Delta_1; \Gamma \vdash e_1 : A \dashv \Delta_2 \quad \Delta_2; \Gamma, x : A \vdash e_2 : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \mathbf{let}_\infty(e_1, x.e_2) : U \dashv \Delta_3} \text{ulet} \\
\\
\frac{\Delta_1; \Gamma \vdash e_1 : X \dashv \Delta_2 \quad \Delta_2, x : X; \Gamma \vdash e_2 : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \mathbf{let}_1(e_1, x.e_2) : U \dashv \Delta_3 \div (x : X)} \text{alet} \quad \frac{\Delta_1; \Gamma \vdash e : A \dashv \Delta_2}{\Delta_1; \Gamma \vdash !e : !A \dashv \Delta_2} \text{bang} \\
\\
\frac{\Delta_1; \Gamma \vdash e : !A \dashv \Delta_2}{\Delta_1; \Gamma \vdash !e : A \dashv \Delta_2} \text{gnab} \quad \frac{\Delta_1, x_1 : \mathbf{Rd } S; \Gamma, x_2 : \mathbf{Wr } S \vdash e : U \dashv \Delta_2}{\Delta_1; \Gamma \vdash \nu(x_1, x_2). e : U \dashv \Delta_2 \div (x_1 : \mathbf{Rd } S)} \text{nu} \\
\\
\frac{\Delta_1; \Gamma \vdash e_1 : S \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_2 : \mathbf{Wr } S \dashv \Delta_3}{\Delta_1, \textcircled{w}; \Gamma \vdash \mathbf{wr}(e_1, e_2) : \mathbf{1} \dashv \Delta_3} \text{wr} \quad \frac{\textcircled{w} \notin \Delta_2 \quad \Delta_1; \Gamma \vdash e_1 : \mathbf{Rd } S \dashv \Delta_2 \quad \Delta_2, \textcircled{w}, x : !S \otimes \mathbf{Rd } S; \Gamma \vdash e_2 : U \dashv \Delta_3}{\Delta_1; \Gamma \vdash \mathbf{rd}(e_1, x.e_2) : U \dashv \Delta_3 \div (\textcircled{w}, x : !S \otimes \mathbf{Rd } S)} \text{rd}
\end{array}$$

Figure A.2: Algorithmic typing rules continued.

$$\begin{array}{c}
\textcircled{w} \notin \Delta_3 \quad \Delta_1; \Gamma \vdash e_1 : \text{Rd } S \dashv \Delta_2 \quad \Delta_2; \Gamma \vdash e_2 : \text{Rd } T \dashv \Delta_3 \\
\Delta_3, \textcircled{w}, x_1 : !S \otimes \text{Rd } S \otimes \text{Rd } T; \Gamma \vdash e_3 : U \dashv \Delta_4 \\
\Delta_3, \textcircled{w}, x_2 : !T \otimes \text{Rd } S \otimes \text{Rd } T; \Gamma \vdash e_4 : U \dashv \Delta_4 \\
\hline
\Delta_1; \Gamma \vdash \text{ch}(e_1, x_1.e_3, e_2, x_2.e_4) : U \dashv \Delta_4 \div \\
(\textcircled{w}, x_1 : !S \otimes \text{Rd } S \otimes \text{Rd } T, x_2 : !T \otimes \text{Rd } T \otimes \text{Rd } S) \quad \text{choice}
\end{array}$$

$$\begin{array}{c}
\Delta_1; \Gamma \vdash e_1 : U \dashv \Delta_2 \\
\Delta_2; \Gamma \vdash e_2 : V \dashv \Delta_3 \\
\hline
\Delta_1; \Gamma \vdash e_1 \mid\triangleright e_2 : V \dashv \Delta_3 \quad \text{fork}
\end{array}$$

Figure A.3: Algorithmic typing rules continued again.

## A.2 TYPE SOUNDNESS

We first define syntax for process and channel typings, which each map a kind of identifier (process name or channel name) to its associated type:

$$\begin{array}{ll}
\text{Process pool typings} & \Phi ::= \cdot \mid \Phi, p : U \\
\text{Channel typings} & \Psi ::= \cdot \mid \Psi, d : S
\end{array}$$

Using the syntax above, we define configuration typing as a straightforward extension of single-process typing, given in Section 3.2:

$\boxed{\Psi \vdash C : \Phi}$  Configuration  $C$  is well-typed.

$$\begin{array}{c}
\frac{}{\Psi \vdash \langle \Sigma; \varepsilon \rangle : \cdot} \text{empty} \qquad \frac{\Psi \vdash e : U \quad \Psi \vdash \langle \Sigma; \pi \rangle : \Phi}{\Psi \vdash \langle \Sigma; \pi, p : e \rangle : \Phi, (p : U)} \text{cons}
\end{array}$$

### A.2.1 Progress

Progress for the functional fragment of ILC (local progress) is fairly standard. We follow the usual recipe, except that we give a special definition of local process termination:

$\boxed{e \text{ lterm}}$  Expression  $e$  is locally terminated.



$$\begin{array}{c}
\frac{}{v \textbf{lterm}} \text{ val} \qquad \frac{}{E[\textbf{rd}(c, x.e)] \textbf{lterm}} \text{ rdterm} \qquad \frac{}{E[\textbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)] \textbf{lterm}} \text{ chterm} \\
\\
\frac{}{E[\textbf{wr}(v, c)] \textbf{lterm}} \text{ wrterm}
\end{array}$$

In other words,  $e \textbf{lterm}$  holds when  $e$  is a value, is reading (either as a standalone read or an external choice), or is writing.

**Lemma A.1** (Local Progress). *If  $\Psi \vdash e : U$ , then either  $e \textbf{lterm}$  or there exists  $e'$  such that  $e \rightarrow e'$ .*

*Proof.* By structural induction on the derivation of  $\Psi \vdash e : U$ . QED.

To state progress on configurations, we give a special definition of “program termination” that permits deadlocks:

$C \textbf{term}$  Configuration  $C$  is terminated.

$$\frac{\begin{array}{l} \forall (p : e) \in \pi. e \textbf{lterm} \\ \text{RdChans}(\pi) = \Sigma_1 \quad \text{WrChans}(\pi) = \Sigma_2 \\ \{(c_1, c_2) \mid c_1 \in \Sigma_1, c_2 \in \Sigma_2, c_2 \rightsquigarrow c_1\} = \emptyset \end{array}}{\langle \Sigma; \pi \rangle \textbf{term}} \text{ Cterm}$$

$$\text{RdChans}(\varepsilon) = \cdot \tag{A.1}$$

$$\text{WrChans}(\varepsilon) = \cdot \tag{A.2}$$

$$\text{RdChans}(\pi, p : E[\textbf{rd}(c, x.e)]) = \text{RdChans}(\pi), c \tag{A.3}$$

$$\text{WrChans}(\pi, p : E[\textbf{rd}(c, x.e)]) = \text{WrChans}(\pi) \tag{A.4}$$

$$\text{RdChans}(\pi, p : E[\textbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)]) = \text{RdChans}(\pi), c_1, c_2 \tag{A.5}$$

$$\text{WrChans}(\pi, p : E[\textbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)]) = \text{WrChans}(\pi) \tag{A.6}$$

$$\text{RdChans}(\pi, p : E[\textbf{wr}(v, c)]) = \text{RdChans}(\pi) \tag{A.7}$$

$$\text{WrChans}(\pi, p : E[\textbf{wr}(v, c)]) = \text{WrChans}(\pi), c \tag{A.8}$$

$$\text{RdChans}(\pi, p : v) = \text{RdChans}(\pi) \tag{A.9}$$

$$\text{WrChans}(\pi, p : v) = \text{WrChans}(\pi) \tag{A.10}$$

In other words,  $C$  **term** holds when either:

1.  $C$  is fully normal: Every process in  $C$  is normalized (consists of a value), or
2.  $C$  is (at least partially) deadlocked: Some (possibly empty) portion of  $C$  is normal, and there exists one or more reading processes in  $C$ , or there exists one or more writing processes in  $C$ , however, no reader-writer process pair exists for a common channel.

**Theorem A.1** (Progress). *If  $\Psi \vdash C : \Phi$ , then either  $C$  **term** or there exists  $C'$  such that  $C \longrightarrow C'$ .*

*Proof.* By structural induction on the derivation of  $\Psi \vdash C : \Phi$ .

**Case A.1.1.**

$$\frac{}{\Psi \vdash \langle \Sigma; \varepsilon \rangle : \cdot} \text{empty}$$

$$\begin{array}{ll} \forall (p : e) \in \varepsilon. e \text{ **lterm**} & (\text{Vacuous}) \\ \Sigma_1 = \text{RdChans}(\varepsilon) = \cdot & (\text{By definition of RdChans}) \\ \Sigma_2 = \text{WrChans}(\varepsilon) = \cdot & (\text{By definition of WrChans}) \\ \{(c_1, c_2) \mid c_1 \in \Sigma_1, c_2 \in \Sigma_2, c_2 \rightsquigarrow c_1\} = \emptyset & \\ \langle \Sigma; \varepsilon \rangle \text{ **term**} & (\text{By rule Cterm}) \end{array}$$

**Case A.1.2.**

$$\frac{\Psi \vdash e : U \quad \Psi \vdash \langle \Sigma; \pi \rangle : \Phi}{\Psi \vdash \langle \Sigma; \pi, p : e \rangle : \Phi, (p : U)} \text{cons}$$

$$e \text{ **lterm**} \text{ or } \exists e' \text{ s.t. } e \rightarrow e'$$

(By i.h.)

$$\langle \Sigma; \pi \rangle \text{ **term**} \text{ or } \exists \langle \Sigma'; \pi' \rangle \text{ s.t. } \langle \Sigma; \pi \rangle \rightarrow \langle \Sigma'; \pi' \rangle$$

(By i.h.)

$$\text{Subcase } \exists e' \text{ s.t. } e \rightarrow e'$$

Subsubcase local

$$e = E[e_1] \text{ and } e' = E[e_2]$$

(Suppose)

$$\langle \Sigma; \pi, p : E[e_1] \rangle \rightarrow \langle \Sigma; \pi, p : E[e_2] \rangle$$

(By rule local)

**Subsubcase fork**

$$e = E[e_1 \mid \triangleright e_2], \ e' = E[e_2], \text{ and } q \notin \Sigma$$

(Suppose)

$$\langle \Sigma; \pi, p : E[e_1 \mid \triangleright e_2] \rangle \rightarrow \langle \Sigma, q; \pi, q : e_1, p : E[e_2] \rangle$$

(By rule fork)

**Subsubcase nu**

$$e = E[\nu(x_1, x_2).e_1], \ e' = E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e_1], \ d \notin \Sigma$$

(Suppose)

$$\langle \Sigma; \pi, p : [\nu(x_1, x_2).e_1] \rangle \rightarrow \langle \Sigma, d; \pi, p : E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e_1] \rangle$$

(By rule nu)

**Subsubcase rw**

$$e = E[\text{rd}(c_1, x.e_1)], \ e' = E[(\text{!}v, c_1)_1/x]e_1, \text{ and } c_2 \rightsquigarrow c_1, \text{ or}$$

$$e = E[\text{wr}(v, c_2)], \ e' = E[()], \text{ and } c_2 \rightsquigarrow c_1$$

**Subsubsubcase**  $e = E[\text{rd}(c_1, x.e_1)], \ e' = E[(\text{!}v, c_1)_1/x]e_1, \text{ and } c_2 \rightsquigarrow c_1$

$$\exists (q : E[\text{wr}(v, c_2)]) \in \pi$$

(By  $c_2 \rightsquigarrow c_1$ )

$$\langle \Sigma; \pi, p : E[\text{rd}(c_1, x.e_1)] \rangle \rightarrow \langle \Sigma; \pi, p : E[(\text{!}v, c_1)_1/x]e_1 \rangle$$

(By rule rw)

**Subsubsubcase**  $e = E[\text{wr}(v, c_2)], \ e' = E[()], \text{ and } c_2 \rightsquigarrow c_1$

$$\exists (q : E[\text{rd}(c_1, x.e_1)]) \in \pi$$

(By  $c_2 \rightsquigarrow c_1$ )

$$\langle \Sigma; \pi, p : E[\text{wr}(v, c_2)] \rangle \rightarrow \langle \Sigma; \pi, p : E[()] \rangle$$

(By rule rw)

**Subsubcase cw**

$$e = E[\text{ch}(c_1, x_1.e_1, c_2, x_2.e_2)], \ e' = E[(\text{!}v, c_1, c_2)_1/x_i]e_i, \ c \rightsquigarrow c_i, \ i \in \{1, 2\}, \text{ or}$$

$$e = E[\text{wr}(v, c)], \ e' = E[()], \ c \rightsquigarrow c_i, \ i \in \{1, 2\}$$

**Subsubsubcase**  $e = E[\text{ch}(c_1, x_1.e_1, c_2, x_2.e_2)], \ e' = E[(\text{!}v, c_1, c_2)_1/x_i]e_i,$

$$c \rightsquigarrow c_i, \ i \in \{1, 2\}$$

$$\exists (q : E[\mathbf{wr}(v, c)]) \in \pi$$

(By  $c \rightsquigarrow c_i$ )

$$\langle \Sigma; \pi, p : E[\mathbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)] \rightarrow \langle \Sigma; \pi, p : E[[(!v, c_1, c_2)_1/x_i]e_i] \rangle \rangle$$

(By rule cw)

**Subsubsubcase**  $e = E[\mathbf{wr}(v, c)], e' = E[()], c \rightsquigarrow c_i, i \in \{1, 2\}$

$$\exists (q : E[\mathbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)]) \in \pi$$

(By  $c \rightsquigarrow c_i$ )

$$\langle \Sigma; \pi, p : E[\mathbf{wr}(v, c)] \rightarrow \langle \Sigma; \pi, p : E[()] \rangle \rangle$$

(By rule cw)

**Subcase**  $\exists \langle \Sigma'; \pi' \rangle$  s.t.  $\langle \Sigma; \pi \rangle \rightarrow \langle \Sigma'; \pi' \rangle$

$$\langle \Sigma; \pi, p : e \rangle \rightarrow \langle \Sigma'; \pi', p : e \rangle$$

(By rules local and congr)

**Subcase**  $\langle \Sigma; p : e \rangle$  **term** and  $\langle \Sigma; \pi \rangle$  **term**

$$\Sigma_1 = \text{RdChans}(\pi, p : e) \text{ and } \Sigma_2 = \text{WrChans}(\pi, p : e)$$

(Suppose)

$$\{(c_1, c_2) \mid c_1 \in \Sigma_1, c_2 \in \Sigma_2, c_2 \rightsquigarrow c_1\} = \emptyset \text{ or}$$

$$\{(c_1, c_2) \mid c_1 \in \Sigma_1, c_2 \in \Sigma_2, c_2 \rightsquigarrow c_1\} \neq \emptyset$$

**Subsubcase**  $\{(c_1, c_2) \mid c_1 \in \Sigma_1, c_2 \in \Sigma_2, c_2 \rightsquigarrow c_1\} = \emptyset$

$$\langle \Sigma; \pi, p : e \rangle \text{ **term**}$$

(By rule Cterm)

**Subsubcase**  $\{(c_1, c_2) \mid c_1 \in \Sigma_1, c_2 \in \Sigma_2, c_2 \rightsquigarrow c_1\} \neq \emptyset$

$$\exists c_2 \rightsquigarrow c_1 \text{ s.t. } c_1 \in \Sigma_1, c_2 \in \Sigma_2$$

(Above)

$$p : v \text{ or } p : E[\mathbf{rd}(c_1, x.e)] \text{ or } p : E[\mathbf{ch}(c_1, x_1.e_1, c_3, x_2.e_2)] \text{ or}$$

$$p : E[\mathbf{ch}(c_3, x_1.e_1, c_1, x_2.e_2)] \text{ or } p : E[\mathbf{wr}(v, c_2)]$$

(By definition of **lterm**)

**Subsubsubcase**  $p : v$

(Impossible)

**Subsubsubcase**  $p : E[\mathbf{rd}(c_1, x.e)]$

$$\exists q : E[\mathbf{wr}(v, c_2)] \in \pi$$

(By  $c_2 \rightsquigarrow c_1$ )

$$\langle \Sigma; \pi, p : E[\mathbf{rd}(c_1, x.e)] \rangle \longrightarrow \langle \Sigma; \pi, p : E[[(!v, c_1)_1/x]e] \rangle$$

(By rule rw)

**Subsubsubcase**  $p : E[\mathbf{ch}(c_1, x_1.e_1, c_3, x_2.e_2)]$

$\exists q : E[\mathbf{wr}(v, c_2)] \in \pi$

(By  $c_2 \rightsquigarrow c_1$ )

$\langle \Sigma; \pi, p : E[\mathbf{ch}(c_1, x_1.e_1, c_3, x_2.e_2)] \rangle \longrightarrow \langle \Sigma; \pi, p : E[(\mathbf{!}v, c_1, c_3)_1/x_1]e_1 \rangle$

(By rule cw)

**Subsubsubcase**  $p : E[\mathbf{ch}(c_3, x_1.e_1, c_1, x_2.e_2)]$

$\exists q : E[\mathbf{wr}(v, c_2)] \in \pi$

(By  $c_2 \rightsquigarrow c_1$ )

$\langle \Sigma; \pi, p : E[\mathbf{ch}(c_3, x_1.e_1, c_1, x_2.e_2)] \rangle \longrightarrow \langle \Sigma; \pi, p : E[(\mathbf{!}v, c_1, c_3)_1/x_2]e_2 \rangle$

(By rule cw)

**Subsubsubcase**  $p : E[\mathbf{wr}(v, c_2)]$

$\exists q : E[\mathbf{rd}(c_1, x.e)] \in \pi$  or  $\exists q : E[\mathbf{ch}(c_1, x_1.e_1, c_3, x_2.e_2)] \in \pi$  or

$\exists q : E[\mathbf{ch}(c_3, x_1.e_1, c_1, x_2.e_2)] \in \pi$

(By  $c_2 \rightsquigarrow c_1$ )

$\langle \Sigma; \pi, p : E[\mathbf{wr}(v, c_2)] \rangle \longrightarrow \langle \Sigma; \pi, p : E[()] \rangle$

(By rule rw)

QED.

## A.2.2 Preservation

Preservation for the functional fragment of ILC (local preservation) is standard.

**Lemma A.2** (Local Preservation). *If  $\Psi \vdash e : U$  and  $e \rightarrow e'$ , then there exists  $\Psi' \supseteq \Psi$  such that  $\Psi \vdash e' : U$ .*

*Proof.* By structural induction on the derivation of  $e \rightarrow e'$ . QED.

To state preservation on configurations, we first state several auxiliary results, which follow the formulation of Gay and Vasconcelos [47]. Lemma A.3 shows that typing of configurations is preserved under configuration equivalence.

**Lemma A.3** (Preservation Modulo Equivalence). *If  $\Psi \vdash C : \Phi$  and  $C \equiv C'$ , then  $\Psi \vdash C' : \Phi$ .*

*Proof.* By structural induction on  $\Psi \vdash C : \Phi$ . QED.

Lemma A.4 shows that a subterm of a well-typed evaluation context is typeable with a subset of the type contexts.

**Lemma A.4** (Typeability of Subterms). *If  $\mathcal{D}$  is a derivation of  $\Psi; \Delta; \Gamma \vdash E[e] : U$  (written  $\mathcal{D} :: \Psi; \Delta; \Gamma \vdash E[e] : U$ ), then*

1. *there exists  $\Psi_1, \Psi_2; \Delta_1, \Delta_2; \Gamma_1, \Gamma_2$  and  $V$  such that  $\Psi = \Psi_1, \Psi_2, \Delta = \Delta_1, \Delta_2, \Gamma = \Gamma_1, \Gamma_2$ ,*
2.  *$\mathcal{D}$  has a subderivation  $\mathcal{D}'$  (written  $\mathcal{D}' \sqsubseteq \mathcal{D}$ ) concluding  $\Psi_1; \Delta_1; \Gamma_1 \vdash e : V$ ,*
3. *the position of  $\mathcal{D}'$  in  $\mathcal{D}$  corresponds to the position of the hole in  $E$  (written  $E[\mathcal{D}' \sqsubseteq \mathcal{D}]$ ).*

*Proof.* By structural induction on the structure of  $E$ . QED.

Lemma A.5 shows that the subterm of a well-typed evaluation context can be replaced.

**Lemma A.5** (Replacement (Evaluation Contexts)). *If*

1.  $\mathcal{D} :: \Psi_1, \Psi_2; \Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash E[e] : U$ ,
2.  $\mathcal{D}' \sqsubseteq \mathcal{D}$  *such that*  $\mathcal{D}' :: \Psi_2; \Delta_2; \Gamma_2 \vdash e : V$ ,
3.  $E[\mathcal{D}' \sqsubseteq \mathcal{D}]$ ,
4.  $\Psi_3; \Delta_3; \Gamma_3 \vdash e' : V$ ,
5.  $\Psi_1, \Psi_3; \Delta_1, \Delta_3; \Gamma_1, \Gamma_3$  *is defined*,

*then*  $\Psi_1, \Psi_3; \Delta_1, \Delta_3; \Gamma_1, \Gamma_3 \vdash E[e'] : U$ .

*Proof.* By structural induction on the structure of  $E$ . QED.

Finally, Lemmas A.6, A.7, A.8, A.9 show that typing of terms is preserved by substitution.

**Lemma A.6** (Substitution (Unrestricted)). *If*

1.  $\Psi_1; \Delta_1; \Gamma_1, x : A \vdash e : U$ ,
2.  $\Psi_2; \Delta_2; \Gamma_2 \vdash e' : A$ ,
3.  $\Psi_1, \Psi_2; \Delta_1, \Delta_2; \Gamma_1, \Gamma_2$  *is defined*,

*then*  $\Psi_1, \Psi_2; \Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash [e'/x]e : U$ .

*Proof.* By structural induction on the derivation of  $\Psi_1; \Delta_1; \Gamma_1, x : A \vdash e : U$ . QED.

**Lemma A.7** (Substitution (Affine)). *If*

1.  $\Psi_1; \Delta_1, x : X; \Gamma_1 \vdash e : U,$
2.  $\Psi_2; \Delta_2; \Gamma_2 \vdash e' : X,$
3.  $\Psi_1, \Psi_2; \Delta_1, \Delta_2; \Gamma_1, \Gamma_2$  is defined,

then  $\Psi_1, \Psi_2; \Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash [e'/x]e : U.$

*Proof.* By structural induction on the derivation of  $\Psi_1; \Delta_1, x : X; \Gamma_1 \vdash e : U.$  QED.

**Lemma A.8** (Substitution (Read Endpoint)). *If*

1.  $\Psi; \Delta, x : \text{Rd } S; \Gamma \vdash e : U,$
2.  $\Psi, d : S; \Delta; \Gamma$  is defined,

then  $\Psi, d : S; \Delta; \Gamma \vdash [\text{Read}(d)/x]e : U.$

*Proof.* By structural induction on the derivation of  $\Psi; \Delta, x : \text{Rd } S; \Gamma \vdash e : U.$  QED.

**Lemma A.9** (Substitution (Write Endpoint)). *If*

1.  $\Psi; \Delta; \Gamma, x : \text{Wr } S \vdash e : U,$
2.  $\Psi, d : S, \Psi; \Delta; \Gamma$  is defined,

then  $\Psi, d : S; \Delta; \Gamma \vdash [\text{Write}(d)/x]e : U.$

*Proof.* By structural induction on the derivation of  $\Psi; \Delta; \Gamma, x : \text{Wr } S \vdash e : U.$  QED.

**Theorem A.2** (Preservation). *If  $\Psi \vdash C : \Phi$  and  $C \longrightarrow C'$ , then there exists  $\Psi' \supseteq \Psi$  and  $\Phi' \supseteq \Phi$  such that  $\Psi' \vdash C' : \Phi'.$*

*Proof.* By structural induction on the derivation of  $C \longrightarrow C'.$

**Case A.2.1.**

$$\frac{e_1 \longrightarrow e_2}{\langle \Sigma; \pi, p : E[e_1] \rangle \longrightarrow \langle \Sigma; \pi, p : E[e_2] \rangle} \text{ local}$$

$$\Psi \vdash \langle \Sigma; \pi, p : E[e_1] \rangle : \Phi \text{ s.t. } \Phi = \Phi_{\pi, p : U},$$

$$\Psi = \Psi_1, \Psi_2, \text{ and } \mathcal{D} :: \Psi_1, \Psi_2 \vdash E[e_1] : U \quad (\text{Assumption})$$

$$\exists \mathcal{D}' \sqsubseteq \mathcal{D} \text{ s.t. } \mathcal{D}' :: \Psi_2 \vdash e_1 : V \text{ and } E[\mathcal{D}' \sqsubseteq \mathcal{D}] \quad (\text{By Lemma A.4})$$

$\Psi_2 \vdash e_2 : V$	(By i.h. and Lemma A.2)
$\Psi_1, \Psi_2 \vdash E[e_2] : U$	(By Lemma A.5)
$\Psi \vdash E[e_2] : U$	(By above equalities)
$\Psi \vdash \langle \Sigma; \pi \rangle : \Phi_\pi$	(Above)
$\Psi \vdash \langle \Sigma; \pi, p : E[e_2] \rangle : (\Phi_\pi, p : U)$	(By rule cons)
$\Psi \vdash \langle \Sigma; \pi, p : E[e_2] \rangle : \Phi$	(By above equalities)
$\Psi' = \Psi$ and $\Phi' = \Phi$	(Suppose)
$\Psi' \vdash \langle \Sigma; \pi, p : E[e_2] \rangle : \Phi'$	(By above equalities)

**Case A.2.2.**

$$\frac{q \notin \Sigma}{\langle \Sigma; \pi, p : E[e_1 \mid \triangleright e_2] \rangle \longrightarrow \langle \Sigma, q; \pi, q : e_1, p : E[e_2] \rangle} \text{ fork}$$

$\Psi \vdash \langle \Sigma; \pi, p : E[e_1 \mid \triangleright e_2] \rangle : \Phi$ s.t. $\Phi = \Phi_\pi, p : U$ ,	
$\Psi = \Psi_1, \Psi_2$ , and $\mathcal{D} :: \Psi_1, \Psi_2 \vdash E[e_1 \mid \triangleright e_2] : U$	(Assumption)
$\exists \mathcal{D}' \sqsubseteq \mathcal{D}$ s.t. $\mathcal{D}' :: \Psi_2 \vdash e_1 \mid \triangleright e_2 : V_2$ and $E[\mathcal{D}' \sqsubseteq \mathcal{D}]$	(By Lemma A.4)
$\Psi_2 \vdash e_1 : V_1$	(By inversion on fork)
$\Psi_2 \vdash e_2 : V_2$	(By inversion on fork)
$\Psi_1, \Psi_2 \vdash E[e_2] : U$	(By Lemma A.5)
$\Psi \vdash E[e_2] : U$	(By above equalities)
$\Psi \vdash \langle \Sigma; \pi \rangle : \Phi_\pi$	(Above)
$\Psi \vdash \langle \Sigma, q; \pi \rangle : \Phi_\pi$	(By $q \notin \Sigma$ )
$\Psi \vdash \langle \Sigma, q; \pi, q : e_1 \rangle : (\Phi_\pi, q : V_1)$	(By rule cons)
$\Psi \vdash \langle \Sigma, q; \pi, q : e_1, p : E[e_2] \rangle : (\Phi_\pi, q : V_1, p : U)$	(By rule cons)
$\Psi \vdash \langle \Sigma, q; \pi, q : e_1, p : E[e_2] \rangle : \Phi, q : V_1$	(By above equalities)
$\Psi' = \Psi$ and $\Phi' = \Phi, q : V_1$	(Suppose)
$\Psi' \vdash \langle \Sigma, q; \pi, q : e_1, p : E[e_2] \rangle : \Phi'$	(By above equalities)



**Case A.2.3.**

$$\frac{C_1 \equiv C'_1 \quad C'_1 \longrightarrow C'_2 \quad C'_2 \equiv C_2}{C_1 \longrightarrow C_2} \text{congr}$$

$$\begin{array}{ll} \Psi \vdash C_1 : \Phi & (\text{Assumption}) \\ C_1 \equiv C'_1 & (\text{Given}) \\ \Psi \vdash C'_1 : \Phi & (\text{By Lemma A.3}) \\ \Psi' \supseteq \Psi \text{ and } \Phi' \supseteq \Phi & (\text{Suppose}) \\ \Psi' \vdash C'_2 : \Phi' & (\text{By i.h.}) \\ \Psi' \vdash C_2 : \Phi' & (\text{By Lemma A.3}) \end{array}$$

**Case A.2.4.**

$$\frac{d \notin \Sigma}{\langle \Sigma; \pi, p : E[\nu(x_1, x_2). e] \rangle \longrightarrow \langle \Sigma, d; \pi, p : E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e] \rangle} \text{nu}$$

$$\begin{array}{l} \Psi \vdash \langle \Sigma; \pi, p : E[\nu(x_1, x_2). e] \rangle : \Phi \text{ s.t. } \Phi = \Phi_\pi, p : U, \\ \Psi = \Psi_1, \Psi_2 \text{ and } \mathcal{D} :: \Psi_1, \Psi_2 \vdash E[\nu(x_1, x_2). e] : U \end{array} \quad (\text{Assumption})$$

$$\begin{array}{l} \exists \mathcal{D}' \sqsubseteq \mathcal{D} \text{ s.t. } \mathcal{D}' :: \Psi_2 \vdash \nu(x_1, x_2). e : V \text{ and } E[\mathcal{D}' \sqsubseteq \mathcal{D}] \\ \text{(By Lemma A.4)} \end{array}$$

$$\Psi_2; \Gamma; \Delta \vdash e : U \text{ where } \Gamma; \Delta = x_1 : \text{Rd } S; x_2 : \text{Wr } S$$

(By inversion on nu)

$$d : S \vdash \text{Read}(d) : \text{Rd } S$$

(By rule rdend)

$$d : S \vdash \text{Write}(d) : \text{Rd } S$$

(By rule wrend)

$$\Psi_3 \vdash [\text{Read}(d)/x_1][\text{Write}(d)/x_2]e : U \text{ where } \Psi_3 = \Psi_2, d : S$$

(By Lemmas A.8 and A.9)

$$\Psi_1, \Psi_3 \vdash E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e] : U_p$$

$$\begin{aligned}
& \text{(By Lemma A.5)} \\
\Psi, \Psi_4 \vdash E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e] : U_p \text{ where } \Psi_4 = c_1 : \text{Rd } S, c_2 : \text{Wr } S \\
& \text{(By above equalities)} \\
\Psi, \Psi_4 \vdash \langle \Sigma; \pi \rangle : \Phi_\pi \\
& \text{(Above)} \\
\Psi, \Psi_4 \vdash \langle \Sigma; \pi, p : E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e] \rangle : (\Phi_\pi, p : U) \\
& \text{(By rule cons)} \\
\Psi, \Psi_4 \vdash \langle \Sigma; \pi, p : E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e] \rangle : \Phi \\
& \text{(Above)} \\
\Psi' = \Psi, \Psi_4 \text{ and } \Phi' = \Phi \\
& \text{(Suppose)} \\
\Psi' \vdash \langle \Sigma; \pi, p : E[[\text{Read}(d)/x_1][\text{Write}(d)/x_2]e] \rangle : \Phi' \\
& \text{(By above equalities)}
\end{aligned}$$

**Case A.2.5.**

$$\frac{c_2 \rightsquigarrow c_1}{\langle \Sigma; \pi, p : E_1[\text{rd}(c_1, x.e)], q : E_2[\text{wr}(v, c_2)] \rangle \longrightarrow \langle \Sigma; \pi, p : E_1[(!v, c_1)_1/x]e, q : E_2[()] \rangle} \text{ rw}$$

$$\begin{aligned}
\Psi \vdash \langle \Sigma; \pi, p : E_1[\text{rd}(c_1, x.e)], q : E_2[\text{wr}(v, c_2)] \rangle : \Phi \text{ s.t. } \Phi = \Phi_\pi, p : U, q : V, \\
\Psi = \Psi_1, \Psi_2, \mathcal{D}_p :: \Psi_1, \Psi_2 \vdash E_1[\text{rd}(c_1, x.e)] : U, \\
\Psi = \Psi_3, \Psi_4, \text{ and } \mathcal{D}_q :: \Psi_3, \Psi_4 \vdash E_2[\text{wr}(v, c_2)] : V \\
& \text{(Assumption)} \\
\exists \mathcal{D}'_p \sqsubseteq \mathcal{D}_p \text{ s.t. } \mathcal{D}'_p :: \Psi_2 \vdash \text{rd}(c_1, x.e) : U' \text{ and } E_1[\mathcal{D}'_p \sqsubseteq \mathcal{D}_p] \\
& \text{(By Lemma A.4)} \\
\exists \mathcal{D}'_q \sqsubseteq \mathcal{D}_q \text{ s.t. } \mathcal{D}'_q :: \Psi_4 \vdash \text{wr}(v, c_2) : \mathbb{1} \text{ and } E_2[\mathcal{D}'_q \sqsubseteq \mathcal{D}_q] \\
& \text{(By Lemma A.4)} \\
c_2 \rightsquigarrow c_1 \text{ s.t. } \Psi(c_2) = \text{Wr } S \text{ and } \Psi(c_1) = \text{Rd } S \\
& \text{(Given)}
\end{aligned}$$

$$\Psi_2; \Delta; \cdot \vdash e : U' \text{ where } \Delta = \textcircled{\mathbf{w}}, x : !S \otimes \text{Rd } S$$

(By inversion on rd)

$$\vdash v : S$$

$$\begin{array}{ll}
& (\text{By inversion on wr}) \\
\vdash !v : !S & \\
& (\text{By rule bang}) \\
\vdash (!v, c_1)_1 : !S \otimes \mathbf{Rd} S & \\
& (\text{By rule apair}) \\
\Psi_2; \textcircled{\mathbf{w}}; \cdot \vdash [(!v, c_1)_1/x]e : U' & \\
& (\text{By Lemma A.7}) \\
\Psi_1, \Psi_2 \vdash E_1[(!v, c_1)_1/x]e : U & \\
& (\text{By Lemma A.5}) \\
\Psi \vdash E_1[(!v, c_1)_1/x]e : U & \\
& (\text{By above equalities}) \\
\Psi \vdash \langle \Sigma; \pi \rangle : \Phi_\pi & \\
& (\text{Above}) \\
\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1)_1/x]e \rangle : (\Phi_\pi, p : U) & \\
& (\text{By rule cons}) \\
\Psi_4 \vdash () : \mathbf{1} & \\
& (\text{By rule unit}) \\
\Psi_3, \Psi_4 \vdash E_2[()] : V & \\
& (\text{By Lemma A.5}) \\
\Psi \vdash E_2[()] : V & \\
& (\text{By above equalities}) \\
\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1)_1/x]e, q : E_2[()] \rangle : (\Phi_\pi, p : U, q : V) & \\
& (\text{By rule cons}) \\
\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1)_1/x]e, q : E_2[()] \rangle : \Phi & \\
& (\text{By above equalities}) \\
\Psi' = \Psi \text{ and } \Phi' = \Phi & \\
& (\text{Suppose}) \\
\Psi' \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1)_1/x]e, q : E_2[()] \rangle : \Phi' & \\
& (\text{By above equalities})
\end{array}$$

**Case A.2.6.**

$$\begin{array}{c}
\frac{c \rightsquigarrow c_i \quad i \in \{1, 2\}}{\langle \Sigma; \pi, p : E_1[\mathbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)], q : E_2[\mathbf{wr}(v, c)] \rangle \longrightarrow \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_i]e_i, q : E_2[()] \rangle} \text{cw} \\
\\
\Psi \vdash \langle \Sigma; \pi, p : E_1[\mathbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)], q : E_2[\mathbf{wr}(v, c)] \rangle : \Phi \\
\text{s.t. } \Phi = \Phi_{\pi, p : U, q : V}, \\
\Psi = \Psi_1, \Psi_2, \mathcal{D}_p :: \Psi_1, \Psi_2 \vdash E_1[\mathbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)] : U, \\
\Psi = \Psi_3, \Psi_4, \text{ and } \mathcal{D}_q :: \Psi_3, \Psi_4 \vdash E_2[\mathbf{wr}(v, c)] : V \\
\text{(Assumption)} \\
\exists \mathcal{D}'_p \sqsubseteq \mathcal{D}_p \text{ s.t. } \mathcal{D}'_p :: \Psi_2 \vdash \mathbf{ch}(c_1, x_1.e_1, c_2, x_2.e_2) : U' \text{ and } E_1[\mathcal{D}'_p \sqsubseteq \mathcal{D}_p] \\
\text{(By Lemma A.4)} \\
\exists \mathcal{D}'_q \sqsubseteq \mathcal{D}_q \text{ s.t. } \mathcal{D}'_q :: \Psi_4 \vdash \mathbf{wr}(v, c_2) : \mathbb{1} \text{ and } E_2[\mathcal{D}'_q \sqsubseteq \mathcal{D}_q] \\
\text{(By Lemma A.4)} \\
c \rightsquigarrow c_1 \text{ s.t. } \Psi(c) = \mathbf{Wr} S, \Psi(c_1) = \mathbf{Rd} S, \Psi(c_2) = \mathbf{Rd} T \text{ or} \\
c \rightsquigarrow c_2 \text{ s.t. } \Psi(c) = \mathbf{Wr} T, \Psi(c_1) = \mathbf{Rd} S, \Psi(c_2) = \mathbf{Rd} T \\
\text{(Given)} \\
\\
\textbf{Subcase } c \rightsquigarrow c_1 \\
\Psi_2; \Delta; \cdot \vdash e : U' \text{ where } \Delta = \textcircled{\mathbf{w}}, x_1 : !S \otimes \mathbf{Rd} S \otimes \mathbf{Rd} T \\
\text{(By inversion on choice)} \\
\\
\vdash v : S \\
\text{(By inversion on wr)} \\
\\
\vdash !v : !S \\
\text{(By rule bang)} \\
\\
\vdash (!v, c_1, c_2)_1 : !S \otimes \mathbf{Rd} S \otimes \mathbf{Rd} T \\
\text{(By rule apair)} \\
\\
\Psi_2; \textcircled{\mathbf{w}}; \cdot \vdash (!v, c_1, c_2)_1/x_1]e_1 : U' \\
\text{(By Lemma A.7)} \\
\\
\Psi_1, \Psi_2 \vdash E_1[(!v, c_1, c_2)_1/x_1]e_1 : U \\
\text{(By Lemma A.5)} \\
\\
\Psi \vdash E_1[(!v, c_1, c_2)_1/x_1]e_1 : U
\end{array}$$

(By above equalities)

$$\Psi \vdash \langle \Sigma; \pi \rangle : \Phi_\pi$$

(Above)

$$\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_1]e_1 \rangle : (\Phi_\pi, p : U)$$

(By rule cons)

$$\Psi_4 \vdash () : \mathbb{1}$$

(By rule unit)

$$\Psi_3, \Psi_4 \vdash E_2[()] : V$$

(By Lemma A.5)

$$\Psi \vdash E_2[()] : V$$

(By above equalities)

$$\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_1]e_1, q : E_2[()] \rangle : (\Phi_\pi, p : U, q : V)$$

(By rule cons)

$$\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_1]e_1, q : E_2[()] \rangle : \Phi$$

(By above equalities)

$$\Psi' = \Psi \text{ and } \Phi' = \Phi$$

(Suppose)

$$\Psi' \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_1]e_1, q : E_2[()] \rangle : \Phi'$$

(By above equalities)

**Subcase**  $c \rightsquigarrow c_2$

$$\Psi_2; \Delta; \cdot \vdash e : U' \text{ where } \Delta = \textcircled{\mathbf{w}}, x_2 : !T \otimes \text{Rd } S \otimes \text{Rd } T$$

(By inversion on choice)

$$\vdash v : T$$

(By inversion on wr)

$$\vdash !v : !T$$

(By rule bang)

$$\vdash (!v, c_1, c_2)_1 : !T \otimes \text{Rd } S \otimes \text{Rd } T$$

(By rule apair)

$$\Psi_2, \textcircled{\mathbf{w}}; \cdot \vdash (!v, c_1, c_2)_1/x_2]e_2 : U'$$

(By Lemma A.7)

$$\Psi_1, \Psi_2 \vdash E_1[(!v, c_1, c_2)_1/x_2]e_2 : U$$

(By Lemma A.5)

$$\Psi \vdash E_1[(!v, c_1, c_2)_1/x_2]e_2 : U$$

(By above equalities)

$$\Psi \vdash \langle \Sigma; \pi \rangle : \Phi_\pi$$

(Above)

$$\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_2]e_2 \rangle : (\Phi_\pi, p : U)$$

(By rule cons)

$$\Psi_4 \vdash () : \mathbb{1}$$

(By rule unit)

$$\Psi_3, \Psi_4 \vdash E_2[()] : V$$

(By Lemma A.5)

$$\Psi \vdash E_2[()] : V$$

(By above equalities)

$$\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_2]e_2, q : E_2[()] \rangle : (\Phi_\pi, p : U, q : V)$$

(By rule cons)

$$\Psi \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_2]e_2, q : E_2[()] \rangle : \Phi$$

(By above equalities)

$$\Psi' = \Psi \text{ and } \Phi' = \Phi$$

(Suppose)

$$\Psi' \vdash \langle \Sigma; \pi, p : E_1[(!v, c_1, c_2)_1/x_2]e_2, q : E_2[()] \rangle : \Phi'$$

(By above equalities)

QED.

### A.3 CONFLUENCE

The following lemmas state structural invariants over write effects and read endpoints of a well-typed configuration: at most one process owns the write token  $\textcircled{w}$ , and every read endpoint is a non-duplicable (affine) resource.

**Lemma A.10** (Unique writer process). *If  $C$  is a well-typed configuration with process pool  $\pi$ , then there exists at most one process in  $\pi$  that owns the write token  $\textcircled{w}$  (i.e., has  $\textcircled{w}$  in its affine context).*

*Proof.* By structural induction over the typing derivation for  $C$ . QED.

**Lemma A.11** (Unique reader process). *If  $C$  is a well-typed configuration with process pool  $\pi$ , and  $c$  is a read endpoint in this configuration, then there exists at most one process in  $\pi$  where  $c$  appears.*

*Proof.* By structural induction over the typing derivation for  $C$ . QED.

**Theorem A.3** (Single-step confluence). *For all well-typed configurations  $C$ , if  $C \longrightarrow C_1$  and  $C \longrightarrow C_2$  then there exists renaming a function  $f$  such that either:*

1.  $C_1 = f(C_2)$ , or
2. there exists  $C_3$  such that  $C_1 \longrightarrow C_3$  and  $f(C_2) \longrightarrow C_3$ .

*Proof.* By induction on the pair of steps  $\langle C \longrightarrow C_1, C \longrightarrow C_2 \rangle$ .

We consider the following cases:

**Case A.3.1** (Congruence).

If either step uses congr, we apply the inductive hypothesis.

**Case A.3.2** (Independent processes).

If both steps advance distinct processes, using any of the rules local, fork and nu, we produce  $C_3$  by combining those two (independent) steps.

**Case A.3.3** (One process).

If both steps advance the same process, we show that this is deterministic (up to naming) by constructing the naming function  $f$  such that  $C_2 = f(C_1)$ . Most cases are straightforward since they perform no nondeterministic choices. The only source of nondeterminism is the name choices, in rules nu and fork. In each case, we map the name choice from the second step to that of the first step.

**Case A.3.4** (Interaction).

If either step uses rw or cw, we rely on Lemmas A.10 and A.11 to show that both steps use either rw or cw, and that the reader-writer process pair is unique.

QED.

By composing multiple uses of this theorem we prove multi-step confluence. However, to carry forth this composition, we need a more general notion of single-step confluence, which is parameteric in a renaming function for the initial configurations.

**Theorem A.4** (Single-step confluence, generalized). *For all well-typed configurations  $C$  and renaming functions  $f$ , if  $C \longrightarrow C_1$  and  $f(C) \longrightarrow C_2$  then there exists renaming function  $g$  such that either:*

1.  $C_1 = g(C_2)$ , or
2. there exists  $C_3$  such that  $C_1 \longrightarrow C_3$  and  $g(C_2) \longrightarrow C_3$ .

*Proof.* Analogous to the proof of Theorem A.3 (single-step confluence). QED.

We prove a full confluence theorem that is generalized similarly, by accepting a renaming function  $f$  to produce a new function  $g$ :

**Theorem A.5** (Full confluence). *For all well-typed configurations  $C$ , and renaming functions  $f$ , if  $C \longrightarrow^* C_1$  and  $f(C) \longrightarrow^* C_2$  and  $C_1$  **term** and  $C_2$  **term** then there exists a renaming function  $g$  such that  $C_1 = g(C_2)$ .*

*Proof.* By induction on the reduction sequence pair  $\langle C \longrightarrow^* C_1, f(C) \longrightarrow^* C_2 \rangle$ . Because of single-step confluence, we know that if either reduction sequence is empty, then the other must be empty, and that if either takes a step, the other must take a step.

**Case A.5.1** (Empty).

When empty, we have the resulting renaming function  $g$  via single-step confluence.

**Case A.5.2** (Step).

We consider the case where each reduction consists of at least one step:  $C \longrightarrow C'_1$  and  $C'_1 \longrightarrow^* C_1$  and  $f(C) \longrightarrow C'_2$  and  $C'_2 \longrightarrow^* C_2$ . By single-step confluence, we have that there exists  $g_0$  such that  $g_0(C'_2) = C'_1$ . By the inductive hypothesis, we have that there exists  $g$  such that  $C_1 = g(C_2)$ .



## REFERENCES

- [1] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2001.
- [2] Y. Lindell and J. Katz, *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.
- [3] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game,” in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 218–229.
- [4] F. Böhl and D. Unruh, “Symbolic universal composability,” *Journal of Computer Security*, vol. 24, no. 1, pp. 1–38, 2016.
- [5] G. Barthe, B. Grégoire, S. Heraud, and S. Béguelin, “Computer-aided security proofs for the working cryptographer,” in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2011.
- [6] M. Abadi and A. D. Gordon, “A calculus for cryptographic protocols: The spi calculus,” *Information and computation*, vol. 148, no. 1, pp. 1–70, 1999.
- [7] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.
- [8] Y. Lindell, “How to simulate it—a tutorial on the simulation proof technique,” in *Tutorials on the Foundations of Cryptography*. Springer, 2017, pp. 277–346.
- [9] M. Abadi and C. Fournet, “Mobile values, new names, and secure communication,” in *ACM Sigplan Notices*, vol. 36, no. 3. ACM, 2001, pp. 104–115.
- [10] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov, “A probabilistic poly-time framework for protocol analysis,” in *Proceedings of the 5th ACM conference on Computer and communications security*. ACM, 1998, pp. 112–121.
- [11] N. Kobayashi, B. C. Pierce, and D. N. Turner, “Linearity and the pi-calculus,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 914–947, 1999.
- [12] S. Fowler, S. Lindley, J. G. Morris, and S. Decova, “Session types without tiers,” 2018.
- [13] D. Hofheinz and V. Shoup, “Gnuc: A new universal composability framework,” *Journal of Cryptology*, vol. 28, no. 3, pp. 423–508, 2015.
- [14] M. Backes, B. Pfitzmann, and M. Waidner, “The reactive simulatability (rsim) framework for asynchronous systems,” *Information and Computation*, vol. 205, no. 12, pp. 1685–1720, 2007.

- [15] D. Hofheinz, D. Unruh, and J. Müller-Quade, “Polynomial runtime and composability,” *Journal of Cryptology*, vol. 26, no. 3, pp. 375–441, 2013.
- [16] M. Bugliesi, S. Calzavara, F. Eigner, and M. Maffei, “Affine refinement types for secure distributed programming,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 4, p. 11, 2015.
- [17] R. Canetti, A. Cohen, and Y. Lindell, “A simpler variant of universally composable security for standard multiparty computation,” in *Annual Cryptology Conference*. Springer, 2015, pp. 3–22.
- [18] R. Canetti and T. Rabin, “Universal composition with joint state,” in *Annual International Cryptology Conference*. Springer, 2003, pp. 265–281.
- [19] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts,” in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.
- [20] J. Katz, “Universally composable multi-party computation using tamper-proof hardware,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2007, pp. 115–128.
- [21] R. Canetti and M. Fischlin, “Universally composable commitments,” in *Annual International Cryptology Conference*. Springer, 2001, pp. 19–40.
- [22] G. Brassard, D. Chaum, and C. Crépeau, “Minimum disclosure proofs of knowledge,” *Journal of Computer and System Sciences*, vol. 37, no. 2, pp. 156–189, 1988.
- [23] J. Camenisch, R. R. Enderlein, S. Krenn, R. Küsters, and D. Rausch, “Universal composition with responsive environments,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 807–840.
- [24] P. Mateus, J. Mitchell, and A. Scedrov, “Composition of cryptographic protocols in a probabilistic polynomial-time process calculus,” in *International Conference on Concurrency Theory*. Springer, 2003, pp. 327–349.
- [25] P. Laud, “Secrecy types for a simulatable cryptographic library,” in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 26–35.
- [26] M. Berger, K. Honda, and N. Yoshida, “Sequentiality and the  $\pi$ -calculus,” in *International Conference on Typed Lambda Calculi and Applications*. Springer, 2001, pp. 29–45.
- [27] B. Blanchet, “Security protocol verification: Symbolic and computational models,” in *Proceedings of the First international conference on Principles of Security and Trust*. Springer-Verlag, 2012, pp. 3–29.

- [28] C. Meadows, “The nrl protocol analyzer: An overview,” *The Journal of Logic Programming*, vol. 26, no. 2, pp. 113–131, 1996.
- [29] S. Escobar, C. Meadows, and J. Meseguer, “Maude-npa: Cryptographic protocol analysis modulo equational properties,” in *Foundations of Security Analysis and Design V*. Springer, 2009, pp. 1–50.
- [30] B. Blanchet, V. Cheval, X. Allamigeon, and B. Smyth, “Proverif: Cryptographic protocol verifier in the formal model,” URL <http://prosecco.gforge.inria.fr/personal/b-blanche/proverif>, 2010.
- [31] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The tamarin prover for the symbolic analysis of security protocols,” in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 696–701.
- [32] G. Barthe, B. Grégoire, and S. Zanella Béguelin, “Formal certification of code-based cryptographic proofs,” *ACM SIGPLAN Notices*, vol. 44, no. 1, pp. 90–101, 2009.
- [33] B. Blanchet, “Cryptoverif: Computationally sound mechanized prover for cryptographic protocols,” in *Dagstuhl seminar “Formal Protocol Verification Applied*, 2007, p. 117.
- [34] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, “Cryphtol: Game-based proofs in higher-order logic.” *IACR Cryptology ePrint Archive*, vol. 2017, p. 753, 2017.
- [35] R. Canetti, A. Stoughton, and M. Varia, “Easyuc: Using easycrypt to mechanize proofs of universally composable security,” in *32nd IEEE Computer Security Foundations Symposium (CSF 2019)*, 2019.
- [36] A. Lochbihler, S. R. Sefidgar, D. Basin, and U. Maurer, “Formalizing constructive cryptography using crypthol,” in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 152–15214.
- [37] U. Maurer, “Constructive cryptography—a new paradigm for security definitions and proofs,” in *Theory of Security and Applications*. Springer, 2011, pp. 33–56.
- [38] B. Pfitzmann and M. Waidner, “A model for asynchronous reactive systems and its application to secure message transmission,” in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 184–200.
- [39] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala, “Analyzing security protocols using time-bounded task-pioas,” *Discrete Event Dynamic Systems*, vol. 18, no. 1, pp. 111–159, 2008.
- [40] U. Maurer and R. Renner, “Abstract cryptography,” in *In Innovations in Computer Science*. Citeseer, 2011.
- [41] J. Camenisch, S. Krenn, R. Küsters, and D. Rausch, “iuc: Flexible universal composability made simple (full version).”

- [42] J. Camenisch, M. Drijvers, and B. Tackmann, “Multi-protocol uc and its use for building modular and efficient protocols.”
- [43] R. Kusters, “Simulation-based security with inexhaustible interactive turing machines,” in *Computer Security Foundations Workshop, 2006. 19th IEEE*. IEEE, 2006, pp. 12–pp.
- [44] S. Dziembowski, S. Faust, and K. Hostáková, “General state channel networks,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 949–966.
- [45] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, “Sprites: Payment channels that go faster than lightning,” *CoRR abs/1702.05812*, 2017.
- [46] S. Dziembowski, L. Ekey, S. Faust, and D. Malinowski, “Perun: Virtual payment channels over cryptographic currencies,” Tech. Rep.
- [47] S. J. Gay and V. T. Vasconcelos, “Linear type theory for asynchronous session types,” *Journal of Functional Programming*, vol. 20, no. 1, pp. 19–50, 2010.