Total Correctness Type Refinements for Communicating Processes

SIVA KAMESH SOMAYYAJULA

CMU-CS-24-108

May 2024

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee
Frank Pfenning, Chair
Robert Harper
Karl Crary
Brigitte Pientka, McGill University

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

This research was sponsored by: HRL Laboratories, LLC under award number 17090181689USPOLINE15; the Defense Advanced Research Projects Agency under award number HR00112020006; the National Science Foundation under award number CNS1446725; and the Algorand Foundation under award number 1031489. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: process calculus, type refinements, type-based termination, dependent types, correctness, futures, proof theory

Abstract

Process calculi are language-based formalisms for investigating software systems with concurrent and/or parallel behaviors. In particular, reasoning about the correctness of such systems can be carried out by proving theorems in a program logic tailored to said behaviors. In sequential functional programming, the celebrated Curry-Howard correspondence interprets logical propositions as types—sets of correct program states—so that a proof of a proposition can be identified as a well-typed and, therefore, correct program. With enough expressive power, type systems in this tradition can collapse the distinction between programming language and program logic.

Due to this singular advantage, much effort has been devoted to realizing this correspondence for process calculi. What remains elusive is a system with dependent types, analogous to logical quantification over process components, that accommodates rich schemes of terminating recursion. In fact, the tension between recursion and logical consistency has traditionally led to design shortcomings and complications in the sequential functional setting. Yet, typability in such a system would imply total correctness: a high-water mark of formal verification where interacting processes are guaranteed to terminate in a desirable state. Thus, many have advocated for establishing total correctness

by a decomposition into partial correctness—correctness oblivious to termination—and termination on its own. Type refinement systems, which can encode such properties on top of an existing type system, satisfy the desideratum for orthogonality implied by this decomposition.

The primary contributions of this thesis are two type refinement systems such that typability of the same process in both establishes its total correctness: sized type refinements determine the termination of mixed inductive-coinductive processes, whereas dependent type refinements verify partial correctness—even in the presence of general recursion. The secondary contribution of this thesis is a design regime based on proof theory. Starting with a simply-typed asynchronous process calculus based on the intuitionistic semi-axiomatic sequent calculus, the design and metatheory of these type refinement systems begin with a novel variation of bidirectional typing to manage the structural presence of refinements, reaching for infinitary proof theory to integrate recursion. We demonstrate the utility of both type refinement systems by building up to an idealized model of verified asynchronous reactive programming.

Acknowledgements

It takes a village to raise a grad student. I thank my advisor and other committee members for not only developing me as a scientist but also instilling in me a research worldview based on proof-theoretic maximalism that *finally* scratched that itch. I also owe an unpayable debt of gratitude to my friends and family, whose encouragement and support over the years inspired me to start my doctorate, stick with it, and get me across the finish line when the time finally came. I am resisting the temptation to reference names and places — those who know, know.

Contents

List of Figures

Chapter 1

Introduction

> *The expressive power of a programming language arises from its strictures and not from its affordances.* — Robert Harper

Language-based models of concurrency and parallelism inform the development of safe and compositional programming language constructs internalizing aspects of both. They involve, at minimum, a *process calculus* paired with a *semantics* for process execution. Reasoning about system correctness, then, is a matter of proving certain theorems in a *program logic* that can discuss process dynamics.

Representing processes and their execution themselves as logical phenomena is an area of active research. It is not only an interesting theoretical exercise—logical properties also tend to translate to desirable language-theoretic results. For example, intuitionistic linear logic can be read as a *session type* discipline that guarantees deadlock freedom for message-passing processes [CP10, PP21]. To that end, [WCPW04] identify two paradigms for developing core languages with respect to a logical theory T, a formula ϕ in T, and a

proof P of ϕ:

	formulas-as-processes	*proofs-as-processes*
process	ϕ	P
operational semantics	construction of P	reduction of P
program logic	metalogic for T	T

While formulas-as-processes typically demands a separate metalogic to reason about correctness, proofs-as-processes implies formulas are types according to the Curry-Howard correspondence. One can view ϕ as compositionally specifying the set of process states that are correct for the system in question. As a result, a proof of ϕ is a process that can only inhabit these states, obviating the need for *ad hoc* metalogical reasoning.

In the analogous paradigm for functional programming, *dependent types* are the gold standard, allowing types to directly index on program states encountered at runtime by analogy to logical quantification. For example, the dependent function type $(x : A) \to B(x)$, corresponding to universal quantification over the domain A, classifies functions whose codomain $B(x)$ varies based on the exact value of the input x. As a result, *dependent session types* [TCP11] were introduced to allow session types for message-passing processes to index on objects communicated. In a retrospective article [TCP21], the authors note that no contemporary dependent session type system supports both of the following critical features at once:

- *Type dependency*: with the exception of [TY18], systems are restricted to *index dependency* [XP99], where domains of quantification (A from above) are *not* types, but

index domains from a separate logical theory. This precludes type dependency on processes or, equivalently, on references to the primitives with which they communicate. That being said, [TY18] does not formally consider recursion.

- *Expressive terminating recursion*: only [SP22] considers nested finite and infinite behaviors classified by *mixed inductive-coinductive types*, but does not go beyond index dependency.

The primary contribution of this dissertation is a process calculus with *both* features. In their presence, typing implies *total correctness*—a high-water mark for formal verification. In other words, a system is guaranteed to terminate in a state satisfying its specification by virtue of typechecking. That being said, traditional dependent type systems for functional programming tend to entangle termination checking with correctness reasoning to ensure type soundness [CSW14], which is prohibitive in the presence of expressive recursion for which termination is non-trivial to demonstrate.

On the other hand, Hoare logic for sequential programs is not only oblivious to the scheme of recursion (induction, coinduction, etc.), but also to termination itself, establishing *partial correctness* while maintaining logical consistency [Hoa69]. Interestingly, treating termination orthogonally in the static analysis of processes has already been advocated by Kobayashi and Sangiorgi [KS08]. As a result, we aim for a solution that fits the following decomposition:

$$\text{total correctness} \;=\; \text{termination} \;+\; \text{partial correctness}$$

In other words, we must give two *separate* type systems for the *same* process calculus such that typability in both establishes termination and partial correctness. Thus, we must take an *extrinistic* view of typing [Rey00] where neither system leaves a syntactic footprint at the level of processes, allowing them to operate orthogonally. We begin with *type refinements* [FP91, XP99], which differ from "traditional" dependent type systems by being extrinsic and intentionally not committing to the theory of $\beta\eta$-equivalence of programs to effect typechecking. Rather, membership in a type can be made contigent on the satisfaction of an *assertion* from an external logical theory. This affords some flexibility: the theory in question can be restricted to hide intensional properties of programs (perhaps something weaker than $\beta\eta$-equivalence) or extended as needed for domain-specific verification. We exploit both aspects to solve the equation above as follows:

$$\text{total correctness type refinements} = \text{sized type refinements} + \text{dependent type refinements}$$

Sized type refinements [SP22] utilize index refinements over the theory of arithmetic to subsume the capabilities of *sized type systems* for termination checking in the functional setting [HPS96]. On the other hand, *dependent type refinements* appeal to a certain first-order theory that is expressive enough to encode *defunctionalized* [Rey72] higher-order process dynamics.

The secondary contribution of this dissertation is our design methodology: a strict adherence to proof theory. First, we base our development on SAX, a process calculus

derived from the Curry-Howard interpretation of the intuitionistic *semi-axiomatic sequent calculus* [DPP20]. Aside: choosing intuitionistic SAX postpones dealing with the interaction between session type linearity and dependency to the future work. Rather than to effect algorithmic typechecking, we then exploit the existing convention of presenting type refinement systems *bidirectionally* [PT00, JV21] to determine the shape of types and their typing rules in the presence of assertions augmenting the judgmental structure. In particular, we view SAX as an intermediate point between (bidirectional) natural deduction and the (unidirectional) sequent calculus. Then, recursion both at the level of types and processes is treated uniformly using *infinitary proof theory* [Sch77]. The contents of this dissertation are outlined below.

- Chapter 2 introduces SAX and its bidirectional type system following [SP23]. In anticipation of the following two chapters, we prove syntactic type soundness as well as termination of an operational semantics of process *configurations* communicating via *futures* [Hal85]. The latter is a consequence of the fundamental theorem of a certain *Kripke logical relation* inspired by models of name-passing calculi [Plo73, BHN14].

- Chapter 3 introduces Index Refined SAX (IRSAX), which extends SAX with *property types* (index refinements) [Dun07] in the style of [DK19].

  - As desired, sized type refinements are conceived as a special case of property types over an arithmetic index domain and uniformly reduce induction, coinduction, and mixed induction and coinduction to lexicographic recursion on sizes/indices.

  - Mixed induction and coinduction To handle recursive types and processes, we initially take a mixed inductive-coinductive view of (sub)typing [DA09, DA10], calling the resulting system IRSAX ∞. Thus, (sub)typing derivations are infinitely deep proofs that represent the unfolding of recursion.
  - We are then able to give an elegant account of termination: we translate well-formed (sub)typing derivations to IRSAX ω, which has infinitely wide but finitely deep (purely inductive) derivations via ω-rules inspired by the infinitary proof theory of arithmetic. Our results then follow from a straightforward extension of the arguments made in Chapter 2. As a sanity check, we show that (co)inductive types are generated by fixed points of *semi-continuous* [CC79, Abe06, LM14] type constructors.

- Chapter 4 introduces Dependent Refined SAX (DRSAX), which extends SAX with dependent refinements following [SP23] in service of partial correctness reasoning.
  - By augmenting the judgmental structure of SAX with Hoare logic-style *pre-* and *post-conditions*, the interaction between bidirectional and infinitary typing pays in dividends. First, we reproduce standard path-sensitive and assume-guarantee reasoning for data types and recursion, respectively. In particular, the interaction between typing derivation circularity and subsumption uniformly treats invariants ((co)inductive hypotheses) for partial recursion. Curiously, bidirectionality also enables us to derive a lightweight mechanism for the encapsulation of codata, where refinements may hide information about internal processes.

  - Syntactic type soundness implies *observable partial correctness*, based on a proof-theoretic characterization of *observability* in SAX, where non-encapsulated data satisfy their postconditions directly. Moreover, because our operational model is based on futures and not speculations [Har16, Chapter 38], we are able to elide a metatheoretic discussion of termination entirely, unlike prior work on type refinements for lazily-evaulating programs [VSJ+14].

- Chapter 5 is where the rubber meets the road: we use IRSAX and DRSAX to verify *causality*, a restricted form of termination, and *recursive refinements* [VRJ13] for programs modeled in the paradigm of *asynchronous reactive programming*, demonstrating both the practical utility of total correctness type refinements for SAX and also limitations to be addressed in the future work.

- Chapter 6 closes this dissertation and discusses avenues for said future work.

In short, we evidence the following thesis:

A proof-theoretic investigation of type refinements within the semi-axiomatic sequent calculus enables the verification of total correctness, decomposed into partial correctness and termination, for communicating processes.

Chapter 2

The Semi-Axiomatic Sequent Calculus Redux

> *So, what is logic actually? Well, if you Google it, you find out it's a rapper.* — Frank Pfenning

Recall that the primary contribution of this thesis is to produce two type refinement systems for the SAX process calculus in service of total correctness checking via typability. In preparation for this goal, this chapter presents the common framework for both systems: simply-typed SAX and its bidirectional type system due to the author and Pfenning [SP23]. As alluded to in the introduction, like Jhala and Vazou [JV21] in the context of functional programming, we consider this bidirectional type system to be the *canonical* presentation of simple typing for SAX. This convention finally pays off in Chapter 4 when it significantly simplifies the design space for dependent type refinements. The structure of this chapter and subsequent ones is as follows.

- In Section §2.1, we introduce the bidirectional semi-axiomatic sequent, adding refinements in subsequent chapters. Notably, our approach incorporates *backwards bidirectional typing* [Zei15] in addition to the "conventional" forward flow of type information.

- In Section §2.2, we give formal syntaxes for types and processes and then analyze their typing rules, also adding refinements in subsequent chapters. At a high level, axioms are inspired by their counterparts in (bidirectional) natural deduction and invertible rules are taken directly from the (unidirectional) sequent calculus.

- In Section §2.3, we review the operational semantics of *configurations* of processes and the futures with which they communicate. Then, we prove syntactic type soundness, which we extend to imply observable partial correctness in Chapter 4. Lastly, we define a certain Kripke logical relation for which its fundamental theorem establishes termination of configuration reduction, which is a significant simplification of the corresponding proofs found in [DPP20, SP22]. In the next chapter, it is extended to accommodate terminating recursion.

2.1 Judgmental Structure

In this section, we develop the bidirectional semi-axiomatic sequent by recalling the proof theory of intuitionistic natural deduction and the corresponding sequent calculus. First, recall that in the sequent calculus, inference rules are either *invertible*—and can be applied at any point in the proof search process, like the right rule for implication—or *noninvert-*

ible, which can only be applied when the sequent "contains enough information," like the right rules for disjunction. Connectives that have noninvertible right rules are *positive* and those that have noninvertible left rules are *negative* [AND92]. The key innovation of the semi-axiomatic sequent calculus is to make the noninvertible rules *axiomatic*. Consider the following right rule for implication as well as the original left rule in the middle that is replaced with its axiomatic counterpart on the right.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to\text{R} \qquad \frac{\Gamma, A \to B \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \to\text{L} \qquad \frac{}{\Gamma, A \to B, A \vdash B} \to\text{X}$$

Via the Curry-Howard correspondence, we understand the right rule for implication to perform function abstraction, whereas the left rule and axiom mean some sort of function call. To understand the difference in computational behavior between both forms of application, we need to consider two different readings of the left rule above, leaving the axiom alone for the moment. To do so, we label the antecedents and succedents in the rules above with communication endpoints $x, y, z, \ldots$ and consider derivations to be processes communicating across these endpoints. Caires and Pfenning [CP10] provide one such labelling, which forces a linear-logical reading of the left rule:

$$\frac{\Gamma, y : A \vdash x : B}{\Gamma \vdash x : A \to B} \to\text{R} \qquad \frac{\Gamma \vdash y : A \quad \Delta, x : B \vdash w : C}{\Gamma, x : A \to B, \Delta \vdash w : C} \to\text{L}$$

Thus, a function is implemented by receiving an argument y along x and later communicating some result along x. As a result, function calls must be *synchronous*, i.e., must wait for x to successfully receive y before any further communication (at the risk of out-of-

order communication along x [DCPT12]). On the other hand, DeYoung et al. (the authors of *op. cit.*) provide an alternate labelling:

$$\frac{\Gamma, y : A \vdash z : B}{\Gamma \vdash x : A \to B} \to\text{R} \quad \frac{\Gamma, x : A \to B \vdash y : A \quad \Gamma, z : B \vdash w : C}{\Gamma, x : A \to B \vdash w : C} \to\text{L}$$

In this reading, a new endpoint z is used to communicate the result of the function call, conferring two benefits:

- *Asynchrony*: receipt of the argument y along x need not complete before other communication
- *Non-linearity*: we can now read $\to$L as a rule in ordinary intuitionistic logic because x and z are distinguished

The final piece in the puzzle is to take asynchrony seriously and, in the function call, decouple the communication of the argument from the use of the result entirely. From the point of view of proof theory, this leaves us with the left axiom below.

$$\frac{}{\Gamma, x : A \to B, y : A \vdash z : B} \to\text{X}$$

Thus, SAX is positioned to be a core calculus for asynchronous communication. In the futures-based interpretation of SAX, endpoints are addresses of write-once read-many references called *futures*. The sequent *qua* typing judgment takes on the following form:

$$s : A, \ldots, t : B \vdash P(s, \ldots, t, u) \div (u : C)$$

This reads "the process P performs blocking reads from *source addresses* $s, \ldots, t$ and one non-blocking write to the *destination address* u," the latter corresponding to the asynchronous initialization of a future at said address [Hal85], according to types $A, B, \ldots, C$. To make the jump to bidirectional typing, we first refine the judgmental structure from above by distinguishing the antecedents from the succedent using arrows $\Rightarrow$ and $\Leftarrow$, respectively, as is typical for sequent calculi:

$$s \Rightarrow A, \ldots, t \Rightarrow B \vdash P(s, \ldots, t, u) \div (u \Leftarrow C)$$

Reading this sequent as the specification of an algorithm for typechecking, $A, \ldots, B, C$ would ordinarily all be *inputs*, corresponding to the *checking mode* of bidirectional natural deduction, with the only *output* being success or failure. This is sufficient for positive left and negative right rules, which are inherited from the sequent calculus as-is. On the other hand, we claim that the natural bidirectional reading of axioms comes from natural deduction, which also has a(n) output/synthesis mode. For example, consider our conversion of positive conjunction introduction and implication elimination to axioms below, where $\Leftarrow$ and $\Rightarrow$ may also appear on the left- and right-hand sides of the sequent, respectively, as *outputs*.

$$\frac{\Gamma \vdash \ldots \Leftarrow A \quad \Gamma \vdash \ldots \Leftarrow B}{\Gamma \vdash \ldots \Leftarrow A \otimes B} \otimes\mathrm{I} \quad \rightsquigarrow \quad \frac{}{\Gamma, s \Leftarrow A, t \Leftarrow B \vdash \ldots \div (u \Leftarrow A \otimes B)} \otimes\mathrm{X}$$

$$\frac{\Gamma \vdash \ldots \Rightarrow A \to B \quad \Gamma \vdash \ldots \Leftarrow A}{\Gamma \vdash \ldots \Rightarrow B} \to\mathrm{E} \quad \rightsquigarrow \quad \frac{}{\Gamma, s \Rightarrow A \to B, t \Leftarrow A \vdash \ldots \div (u \Rightarrow B)} \to\mathrm{X}$$

In $\otimes$I, $A \otimes B$ in the conclusion flows to A and B in the premises by having all judgments in input/checking mode. Likewise, in $\otimes$X, $A \otimes B$ as the succedent in input mode flows to A and B into the context in output mode. As a result, the directionality of the arrows is preserved at the expense of some modulation in input/output behavior. Dually, the top-down flow of $A \to B$ to A and B in $\to$E translates to a left-to-right flow in $\to$X. While seemingly an unexpected novelty, synthesis of types into the context recalls Zeilberger's *backwards bidirectional typing* [Zei15] for linear natural deduction. To summarize the current judgmental situation:

	$\Rightarrow$	$\Leftarrow$
antecedent	input/checking	output/synthesis
succedent	output/synthesis	input/checking

Finally, to aim for a minimal set of typing rules, the generic succedent is an input, which is the most general [DK21]. The resulting judgmental structure is in Figure 2.1.

$$\begin{array}{c} \text{judgments } J := s \Rightarrow A \mid t \Leftarrow A \\ \text{contexts } \Gamma := \cdot \mid \Gamma, J \\ \text{generic succedent } \gamma := t \Leftarrow C \\ \text{process typing } \Gamma \vdash P \div J \\ \text{subtyping } A \leq B \end{array}$$

Figure 2.1: SAX Judgments

2.2 Syntax and Bidirectional Typing

The syntax for addresses, types, and processes are given in Figure 2.2. Before we de-

scribe a representative sample of process constructors and their associated typing rules, we briefly comment on syntax and variable binding conventions below.

- In addition to eager pair (positive conjunction) and function (implication) types, SAX includes labelled sums (disjunction) and lazy records (negative conjunction).

- As is typical for name-passing calculi [BHN14], addresses are separated into *address variables* and *runtime addresses*, where the latter may be substituted for the former. Thus, $P(x)$ indicates that x is free in P whereas $P(t)$ means $[t/x]P$ when x is unambiguous in context. Process typing contexts freely mixing address variables and runtime variables departs from other presentations of process calculi where the latter are sequestered to a separate zone [Har16]. Below, we give the substitution principle for address variables to clarify this novel syntactic convention.

- Each process constructor corresponds to a judgmental or logical rule in the semi-axiomatic sequent calculus except for definition calls $f\,(\overline{s})$, which draw from a typed signature of definitions of the form $f\,(\overline{x : A}) = P\,(\overline{x})$, where $\overline{\cdot}$ indicates an object list. Their computational interpretation is justified by principal cut reduction (Figure 2.6). In this chapter, we impose the restriction that definitions be non-recursive by inductively generating the typing judgment from its rules. This is relaxed in the following two chapters when we move to a mixed inductive-coinductive view of (sub)typing.

Lemma 2.1 (Address Substitution Principle)**.**

- *If* $\Gamma, x : A \vdash P\,(x) \div \gamma$, *then* $\Gamma, s : A \vdash P\,(s) \div \gamma$

- *If* $\Gamma \vdash P\,(x) \div (x : A)$, *then* $\Gamma \vdash P\,(s) \div (s : A)$

Proof. By a routine induction. □

type $A :=$	$A^+ \mid A^-$		
pos. type $A^+ :=$	$\mathbf{1}$		positive unit
	$\mid$	$A \otimes B$	eager pair
	$\mid$	$\oplus \{\ell : A_\ell\}_{\ell \in S}$	labelled sum
neg. type $A^- :=$		$A \to B$	function
	$\mid$	$\& \{\ell : A_\ell\}_{\ell \in S}$	lazy record
addresses $s, t :=$		$x, y, z, \ldots$	address variables
	$\mid$	$a, b, c, \ldots$	runtime addresses
value $V :=$		$\langle\rangle$	$(\mathbf{1})$
	$\mid$	$\langle s, t \rangle$	$(\otimes\mathrm{R}, \to\mathrm{L})$
	$\mid$	$\ell \cdot t$	$(\oplus\mathrm{R}, \&\mathrm{L})$
continuation $K :=$		$\langle\rangle \Rightarrow P$	$(\mathbf{1}\mathrm{L})$
	$\mid$	$\langle x, y \rangle \Rightarrow P(x, y)$	$(\to\mathrm{R}, \otimes\mathrm{L})$
	$\mid$	$\{\ell \cdot x \Rightarrow P_\ell\,(x)\}_{\ell \in S}$	$(\&\mathrm{R}, \oplus\mathrm{L})$
storable $S :=$		$V \mid K$	

process $P :=$	$\texttt{copy}\,t\,s$	copy contents of s to t	(identity)
$\mid$	$x \leftarrow P\,(x)\,; Q\,(x)$	spawn P writing to x, proceed concurrently as Q	(cut/snip, x bound)
$\mid$	$\texttt{read}\,t\,S$	pass V in t to $S = K$ or pass $S = V$ to K in t	(left rule)
$\mid$	$\texttt{write}\,t\,S$	write S to t	(right rule)
$\mid$	$f\,(\bar{s})$	definition call	$f\,(\overline{x : A}) = P\,(\bar{x})$

Figure 2.2: SAX Syntax

2.2.1 Logical Rules

Let us pick up where we left off with eager pairs and functions. The right axiom for positive conjunction given below types $\texttt{write}\,u\,\langle s, t \rangle$, which writes the pair of addresses $\langle s, t \rangle$ to u. As we noted before, the bottom-up flow of type information in natural deduction translates to a right-to-left flow.

$$\frac{\Gamma \vdash \ldots \Leftarrow A \quad \Gamma \vdash \ldots \Leftarrow B}{\Gamma \vdash \ldots \Leftarrow A \otimes B}\ \otimes\mathrm{I} \quad \rightsquigarrow \quad \frac{}{\Gamma, s \Leftarrow A, t \Leftarrow B \vdash \mathtt{write}\, u\, \langle s, t\rangle \div (u \Leftarrow A \otimes B)}\ \otimes\mathrm{X}$$

The corresponding left rule, $\mathtt{read}\, u\ (\langle x, y\rangle \Rightarrow P\,(x, y))$, is inherited directly from the sequent calculus (where all type components are uniformly in input mode) and pattern matches on the value at u.

$$\frac{\Gamma, u \Rightarrow A \otimes B, x \Rightarrow A, y \Rightarrow B \vdash P\,(x, y) \div \gamma}{\Gamma, u \Rightarrow A \otimes B \vdash \mathtt{read}\, u\ (\langle x, y\rangle \Rightarrow P\,(x, y)) \div \gamma}\ \otimes\mathrm{L}$$

The unit for eager pairs, $\mathbf{1}$, involves the unit value $\langle\rangle$ and trivial continuation $\langle\rangle \Rightarrow P$ in the same roles as above.

$$\frac{}{\Gamma \vdash \mathtt{write}\, t\, \langle\rangle \div (t \Leftarrow \mathbf{1})}\ \mathbf{1}\mathrm{X} \qquad \frac{\Gamma, t \Rightarrow \mathbf{1} \vdash P \div \gamma}{\Gamma, t \Rightarrow \mathbf{1} \vdash \mathtt{read}\, t\ (\langle\rangle \Rightarrow P) \div \gamma}\ \mathbf{1}\mathrm{L}$$

Dually, the left axiom for functions, $\mathtt{read}\, u\, \langle s, t\rangle$, stores the result of applying the function at u to the argument at s in destination t. Once again, the top-to-bottom flow of type information for implication elimination translates to a left-to-right flow.

$$\frac{\Gamma \vdash \ldots \Rightarrow A \to B \quad \Gamma \vdash \ldots \Leftarrow A}{\Gamma \vdash \ldots \Rightarrow B}\ \to\mathrm{E} \quad \rightsquigarrow \quad \frac{}{\Gamma, u \Rightarrow A \to B, s \Leftarrow A \vdash \mathtt{read}\, u\, \langle s, t\rangle \div (t \Rightarrow B)}\ \to\mathrm{X}$$

As a result, the right rule for implication, $\mathtt{write}\, t\ (\langle x, y\rangle \Rightarrow P\,(x, y))$, writes a process abstracting over an argument x and destination y to t (once again, inherited from the sequent calculus, requiring all scrutinees to be in input mode).

$$\frac{\Gamma, x \Rightarrow A \vdash P\,(x,y) \div (y \Leftarrow B)}{\Gamma \vdash \texttt{write}\,t\ (\langle x,y\rangle \Rightarrow P\,(x,y)) \div (t \Leftarrow A \to B)}\ \to\mathrm{R}$$

The following examples develop intuition about their use.

Example 2.1 (Eager Swap)**.** The following definition reads a pair in z and stores the swapped version in w.

$$\texttt{swap}(z : A \otimes B, w : B \otimes A) =$$
$$\texttt{read}\,z\ (\langle x,y\rangle \Rightarrow \texttt{write}\,w\,\langle y,x\rangle)$$

Example 2.2 (Function Composition)**.** The following definition writes the composition of functions f then g into h.

$$\texttt{comp}(f : A \to B, g : B \to C, h : A \to C) =$$
$$\texttt{write}\,h\ (\langle x,z\rangle \Rightarrow y \leftarrow \texttt{read}\,f\,\langle x,y\rangle; \texttt{read}\,g\,\langle y,z\rangle)$$

Labelled sums and lazy records follow the same script: both involve tagged values $\ell \cdot t$ and case-analyzing continuations $\{\ell \cdot x \Rightarrow P_\ell\,(x)\}_{\ell \in S}$ in dual roles, with right and left axioms respectively flowing type information from right to left and left to right. Then, the invertible rules are taken directly from the sequent calculus. In particular, lazy records are also destination-passing by writing the result of projection to t and x.

$$\frac{}{\Gamma, s \Leftarrow A_k \vdash \mathtt{write}\, t\ (k \cdot s) \div \left(t \Leftarrow \oplus\left\{\ell : A_\ell\right\}_{\ell \in S}\right)}\ \oplus\mathrm{X}, k \in S$$

$$\frac{\left\{\Gamma, t \Rightarrow \oplus\left\{\ell : A_\ell\right\}_{\ell \in S}, x : A_k \vdash P_k\left(x\right) \div \gamma\right\}_{k \in S}}{\Gamma, t \Rightarrow \oplus\left\{\ell : A_\ell\right\}_{\ell \in S} \vdash \mathtt{read}\, t\ \left\{\ell \cdot x \Rightarrow P_\ell\left(x\right)\right\}_{\ell \in S} \div \gamma}\ \oplus\mathrm{L}$$

$$\frac{\left\{\Gamma \vdash P_k\left(x\right) \div \left(x \Leftarrow A_\ell\right)\right\}_{\ell \in S}}{\Gamma \vdash \mathtt{write}\, t\ \left\{\ell \cdot x \Rightarrow P_\ell\left(x\right)\right\}_{\ell \in S} \div \left(t \Leftarrow \&\left\{\ell : A_\ell\right\}_{\ell \in S}\right)}\ \&\mathrm{R}$$

$$\frac{}{\Gamma, t \Rightarrow \&\left\{\ell : A_\ell\right\}_{\ell \in S} \vdash \mathtt{read}\, t\ \left(k \cdot s\right) \div \left(s \Rightarrow A_k\right)}\ \&\mathrm{X}, k \in S$$

Once again, the following examples develop intuition about their use.

Example 2.3 (Negation I)**.** Let $\mathtt{bool} \triangleq \oplus\left\{\mathtt{true} : \mathbf{1}, \mathtt{false} : \mathbf{1}\right\}$. Then, the following definition writes the negation of x into y.

$$\mathtt{negate}(x : \mathtt{bool}, y : \mathtt{bool}) =$$

$$\mathtt{read}\, x\ \left\{\mathtt{true} \cdot x' \Rightarrow \mathtt{write}\, y\ \left(\mathtt{false} \cdot x'\right), \mathtt{false} \cdot x' \Rightarrow \mathtt{write}\, y\ \left(\mathtt{true} \cdot x'\right)\right\}$$

Example 2.4 (Lazy Swap)**.** Let $A \,\&\, B \triangleq \&\left\{\mathtt{fst} : A, \mathtt{snd} : B\right\}$. Then, the following definition writes a lazy record with the components of z swapped into w.

$$\mathtt{swap}(z : A \,\&\, B, w : B \,\&\, A) =$$

$$\mathtt{write}\, w\ \left\{\mathtt{fst} \cdot x \Rightarrow \mathtt{read}\, z\ \left(\mathtt{snd} \cdot x\right), \mathtt{snd} \cdot y \Rightarrow \mathtt{read}\, z\ \left(\mathtt{fst} \cdot y\right)\right\}$$

2.2.2 Phase Change: Subsumption and Definition Calls

Analogous to bidirectional natural deduction, the change of phase from input to output mode, reading rules from the bottom up, is mediated by *subsumption* rules $\leq \{\mathrm{R}, \mathrm{L}\}$. Note that the judgmental distinction between the left- and right-hand sides of the sequent requires two rules [LDM06]. The subtyping judgment, $A \leq B$, is defined in Figure 2.5. All rules are standard: positive subtyping is covariant in both components (label sets and type constituents) whereas negative subtyping is contravariant in the first component [LDD+22]. Thus, labelled sums and lazy records also enjoy *width* and *depth* subtyping [LDD+22].

$$\frac{A \leq B \quad \Gamma \vdash P \div (t \Rightarrow A)}{\Gamma \vdash P \div (t \Leftarrow B)} \leq \mathrm{R} \qquad \frac{A \leq B \quad \Gamma, s \Leftarrow B \vdash P \div \gamma}{\Gamma, s \Rightarrow A \vdash P \div \gamma} \leq \mathrm{L}$$

On the other hand, typed process definitions take the place of type annotations, which would admit changes of phase from output to input mode (once again, reading rules from the bottom up). Following [LDD+22], a definition call synthesizes the type signature of its associated definition as long as the definition body checks against it. Because our type system is inductively generated, typing precludes recursive definitions. This restriction is relaxed in the following two chapters by way of *infinite proofs*.

$$\frac{f\left(\overline{x : A}, y : B\right) = P\left(\overline{x}, y\right) \quad \overline{s \Rightarrow A} \vdash P\left(\overline{s}, t\right) \div (t \Leftarrow B)}{\overline{s \Leftarrow A} \vdash f\left(\overline{s}, t\right) \div (t \Rightarrow B)} \text{ call}$$

2.2.3 Cut, Snips, and Identity

Recalling that cut reduction provides a computational interpretation for proof systems, the process behind the cut rule forms the core of computation with futures in SAX: $x \leftarrow P\,(x)\,;Q\,(x)$ spawns P to perform a non-blocking write to a newly allocated future addressed by x while concurrently proceeding as Q, which may perform a blocking read from x. We give two forms of the cut rule depending on which premise outputs the cut formula A.

$$\frac{\Gamma \vdash P(x) \div (x \Leftarrow A) \quad \Gamma, x \Leftarrow A \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma}\ \text{snipR}$$

$$\frac{\Gamma \vdash P(x) \div (x \Rightarrow A) \quad \Gamma, x \Rightarrow A \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma}\ \text{snipL}$$

In the absence of definition calls, these rules actually correspond to *snips*: *analytic cuts* [Smu69] where A is a subformula of an axiom's principal formula found in either the right- or left-hand side of the cut. While SAX does not admit cut directly, it still does by reduction to snips, thus establishing the subformula property that implies logical consistency. Insofar as both forms of snip above are not derivable from each other via subsumption, they are also not inter-derivable in the original type system for SAX. Thus, the bidirectional type system presented in this chapter is, in a technical sense, minimal. The reader may have noticed an additional departure from the sequent calculus: having type annotations by way of definition calls separately from an annotated cut rule (where A is an input in both premises) has an extralogical character. To confirm that we have not lost any expressive power, we can derive such a rule below.

Example 2.5 (Cut in the Sequent Calculus)**.** The cut rule from the sequent calculus can be

derived in the following two ways assuming $f\left(\overline{x : A}, y : B, z : C\right) = Q\left(\overline{x}, y, z\right)$.

$$\dfrac{\overline{s \Rightarrow A} \vdash P\left(x\right) \div \left(x \Leftarrow B\right) \quad \dfrac{\dfrac{\overline{s \Rightarrow A}, x \Rightarrow B \vdash Q\left(\overline{s}, x, t\right) \div \left(t \Leftarrow C\right)}{\overline{s \Leftarrow A}, x \Leftarrow B \vdash f\left(\overline{s}, x, t\right) \div \left(t \Rightarrow C\right)} \text{ call}}{\overline{s \Rightarrow A}, x \Leftarrow B \vdash f\left(\overline{s}, x, t\right) \div \left(t \Leftarrow C\right)} \leq \{\text{R}, \text{L}\}}{\overline{s \Rightarrow A} \vdash x \leftarrow P\left(x\right); f\left(\overline{s}, x, t\right) \div \left(t \Leftarrow C\right)} \text{ snip}^{+}$$

or, assuming $f\left(\overline{x : A}, y : B\right) = Q\left(\overline{x}, y\right)$:

$$\dfrac{\dfrac{\dfrac{\overline{s \Rightarrow A} \vdash Q\left(\overline{s}, x\right) \div \left(x \Leftarrow B\right)}{\overline{s \Leftarrow A} \vdash f\left(\overline{s}, x\right) \div \left(x \Rightarrow B\right)} \text{ call}}{\overline{s \Rightarrow A} \vdash f\left(\overline{s}, x\right) \div \left(x \Rightarrow B\right)} \leq \{\text{L}\} \quad \overline{s \Rightarrow A}, x \Rightarrow B \vdash P\left(x\right) \div \left(t \Leftarrow C\right)}{\overline{s \Rightarrow A} \vdash x \leftarrow f\left(\overline{s}, x\right); P\left(x\right) \div \left(t \Leftarrow C\right)} \text{ snip}^{-}$$

Now, the process corresponding to the identity rule, $\mathtt{copy}\, t\, s$, copies the contents of s to t. Likewise, it comes in two forms depending on the side of the sequent where the principal formula A is output.

$$\dfrac{}{\Gamma, s \Leftarrow A \vdash \mathtt{copy}\, t\, s \div \left(t \Leftarrow A\right)} \text{ idR} \qquad \dfrac{}{\Gamma, s \Rightarrow A \vdash \mathtt{copy}\, t\, s \div \left(t \Rightarrow A\right)} \text{ idL}$$

As before, neither rule is derivable from the other. Moreover, the reader may have once again noticed our departure from the sequent calculus: usually, subsumption is "baked into" the identity rule, allowing distinct formulas to be inputs on both sides of the sequent [DHP18, Theorem 6.1]. While the symmetry with snips is pleasing, we must comment on whether this design choice removes the path to algorithmic typing. Luckily, the bidirectional system can be transformed into a (finite) syntax-directed system where all judgments are in input mode ($\Rightarrow$ on the left and $\Leftarrow$ on the right, like the sequent calculus) such that all apparent occurrences of output mode are met with a subtyping check

instead, like in the examples above and below. However, we do not dwell on algorithmic typing as it is complicated by the incorporation of infinite proofs and implicit quantification in subsequent chapters, which we discuss *en passant*. In fact, as above, we can derive the general identity rule from their combination.

Example 2.6 (Identity in the Sequent Calculus)**.** The general identity rule from the sequent calculus is given below.

$$\frac{A \leq B}{\Gamma, s \Rightarrow A \vdash \mathtt{copy}\, t\, s \div (t \Leftarrow B)}\ \mathrm{id}$$

It can be derived in the following two ways:

$$\frac{A \leq B \quad \dfrac{}{\Gamma, s \Leftarrow B \vdash \mathtt{copy}\, t\, s \div (t \Leftarrow B)}\ \mathrm{idR}}{\Gamma, s \Rightarrow A \vdash \mathtt{copy}\, t\, s \div (t \Leftarrow B)}\ {\leq}\mathrm{L}$$

or

$$\frac{A \leq B \quad \dfrac{}{\Gamma, s \Rightarrow A \vdash \mathtt{copy}\, t\, s \div (t \Rightarrow A)}\ \mathrm{idL}}{\Gamma, s \Rightarrow A \vdash \mathtt{copy}\, t\, s \div (t \Leftarrow B)}\ {\leq}\mathrm{R}$$

2.2.4 Typing Summary and Metatheory

Before we complete this chapter by discussing the futures-based operational metatheory for SAX, we need to check whether the bidirectional typing presented so far is "the right one." Technically, it needs to be sound and complete with respect to the corresponding non-bidirectional type system, which we establish below.

Definition 2.1 (Non-Bidirectional Typing)**.** Let $\mathbf{\Gamma}$ be a context of judgments $\boldsymbol{J} := t : A$. Then, let $\mathbf{\Gamma} \vdash P \div \boldsymbol{J}$ be generated by rules identical to those in Figure 2.4, but with $(\Rightarrow)$

and $(\Leftarrow)$ replaced by $(:)$. Note that subtyping is still included and that the snip rules maintain their distinction as opposed to the identity rules, which collapse into one.

To determine completeness of bidirectional typing, we need the following definition to bridge the gap between definition calls and type annotation as found in bidirectional natural deduction.

Definition 2.2 (Unfolding)**.** P' *unfolds to* P iff, after a finite number of definition call unfoldings in P', P' is syntactically identical to P.

Theorem 2.1 (SAX Soundness and Completeness of Bidirectional Typing)**.** *Let* $|J|$ *turn* $(\Rightarrow)$ *and* $(\Leftarrow)$ *to* $(:)$*. Extending* $|\cdot|$ *to* Γ *in the obvious way:*

- Soundness*: if* $\Gamma \vdash P \div J$*, then* $|\Gamma| \vdash P \div |J|$ *(note that the converse does not hold, e.g.,* $|s \Leftarrow A, t \Leftarrow B| \vdash \mathtt{write}\, u \langle s, t \rangle \div |u \Rightarrow A \otimes B|$ *is derivable but the underlying bidirectional sequent is not).*

- Completeness*: if* $\mathbf{\Gamma} \vdash P \div \boldsymbol{J}$*, then* $\Gamma \vdash P' \div J$ *where* $|\Gamma| = \mathbf{\Gamma}$*,* $|J| = \boldsymbol{J}$*, and there exists an extension of the process definition signature and* P' *such that* P' *unfolds to* P*.*

Proof. Both are routine inductions on the process typing derivation: soundness "forgets" bidirectionality whereas completeness requires some more explanation: in natural deduction, to orient the arrows of the judgments in preparation for the corresponding bidirectional rule, either subsumption or type annotation is used. In our case, we simulate type annotation of a process subterm by extending the process definition signature with an auxiliary definition pointing to the subterm. Then, the subterm is replaced by a call to said definition. Because completeness proceeds by induction, this only happens finitely many times, as desired by the definition of "unfolding." □

Aesthetically, we have already shown that our system has a minimal number of (non-inter-derivable) typing rules. However, we can make an even stronger claim: we have the "right" system according to Pfenning's recipe for bidirectional typing [DK21], where we can draw a correspondence between certain bidirectional typing derivations and proofs in normal form. This is presented in Figure 2.3.

bidirectional typing	natural deduction	SAX / sequent calculus
annotation/call-free	β-normal	cut-free
identity- and subsumption-free	η-long	identity-free

Figure 2.3: Correspondence between Bidirectional Typing and Proof Normal Forms

To close this section, we tabulate the bidirectional typing presented into Figure 2.4 and Figure 2.5.

$$\frac{A \leq B \quad \Gamma \vdash P \div (t \Rightarrow A)}{\Gamma \vdash P \div (t \Leftarrow B)} \leq\mathrm{R} \qquad \frac{A \leq B \quad \Gamma, s \Leftarrow B \vdash P \div \gamma}{\Gamma, s \Rightarrow A \vdash P \div \gamma} \leq\mathrm{L}$$

$$\frac{f\left(\overline{x : A}, y : B\right) = P\left(\overline{x}, y\right) \quad \overline{s \Rightarrow A} \vdash P\left(\overline{s}, t\right) \div (t \Leftarrow B)}{\overline{s \Leftarrow A} \vdash f\left(\overline{s}, t\right) \div (t \Rightarrow B)}\ \text{call}$$

$$\frac{\Gamma \vdash P(x) \div (x \Leftarrow A) \quad \Gamma, x \Leftarrow A \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma}\ \text{snipR}$$

$$\frac{\Gamma \vdash P(x) \div (x \Rightarrow A) \quad \Gamma, x \Rightarrow A \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma}\ \text{snipL}$$

$$\frac{}{\Gamma, s \Leftarrow A \vdash \mathtt{copy}\, t\, s \div (t \Leftarrow A)}\ \text{idR} \qquad \frac{}{\Gamma, s \Rightarrow A \vdash \mathtt{copy}\, t\, s \div (t \Rightarrow A)}\ \text{idL}$$

$$\frac{}{\Gamma \vdash \mathtt{write}\, t\, \langle\rangle \div (t \Leftarrow \mathbf{1})}\ \mathbf{1}\mathrm{X} \qquad \frac{\Gamma, t \Rightarrow \mathbf{1} \vdash P \div \gamma}{\Gamma, t \Rightarrow \mathbf{1} \vdash \mathtt{read}\, t\ (\langle\rangle \Rightarrow P) \div \gamma}\ \mathbf{1}\mathrm{L}$$

$$\frac{}{\Gamma, s \Leftarrow A, t \Leftarrow B \vdash \mathtt{write}\, u\, \langle s, t\rangle \div (u \Leftarrow A \otimes B)}\ \otimes\mathrm{X}$$

$$\frac{\Gamma, u \Rightarrow A \otimes B, x \Rightarrow A, y \Rightarrow B \vdash P\left(x, y\right) \div \gamma}{\Gamma, u \Rightarrow A \otimes B \vdash \mathtt{read}\, u\ (\langle x, y\rangle \Rightarrow P\left(x, y\right)) \div \gamma}\ \otimes\mathrm{L}$$

$$\frac{\Gamma, x \Rightarrow A \vdash P\left(x, y\right) \div (y \Leftarrow B)}{\Gamma \vdash \mathtt{write}\, t\ (\langle x, y\rangle \Rightarrow P\left(x, y\right)) \div (t \Leftarrow A \to B)}\ \to\mathrm{R}$$

$$\frac{}{\Gamma, u \Rightarrow A \to B, s \Leftarrow A \vdash \mathtt{read}\, u\, \langle s, t\rangle \div (t \Rightarrow B)}\ \to\mathrm{X}$$

$$\frac{}{\Gamma, s \Leftarrow A_k \vdash \mathtt{write}\, t\ (k \cdot s) \div \left(t \Leftarrow \oplus \left\{\ell : A_\ell\right\}_{\ell \in S}\right)}\ \oplus\mathrm{X}, k \in S$$

$$\frac{\left\{\Gamma, t \Rightarrow \oplus \left\{\ell : A_\ell\right\}_{\ell \in S}, x : A_k \vdash P_k\left(x\right) \div \gamma\right\}_{k \in S}}{\Gamma, t \Rightarrow \oplus \left\{\ell : A_\ell\right\}_{\ell \in S} \vdash \mathtt{read}\, t\ \left\{\ell \cdot x \Rightarrow P_\ell\left(x\right)\right\}_{\ell \in S} \div \gamma}\ \oplus\mathrm{L}$$

$$\frac{\left\{\Gamma \vdash P_k\left(x\right) \div (x \Leftarrow A_\ell)\right\}_{\ell \in S}}{\Gamma \vdash \mathtt{write}\, t\ \left\{\ell \cdot x \Rightarrow P_\ell\left(x\right)\right\}_{\ell \in S} \div \left(t \Leftarrow \& \left\{\ell : A_\ell\right\}_{\ell \in S}\right)}\ \&\mathrm{R}$$

$$\frac{}{\Gamma, t \Rightarrow \& \left\{\ell : A_\ell\right\}_{\ell \in S} \vdash \mathtt{read}\, t\ (k \cdot s) \div (s \Rightarrow A_k)}\ \&\mathrm{X}, k \in S$$

Figure 2.4: SAX Typing

$$\frac{A \leq A' \quad B \leq B'}{A \otimes B \leq A' \otimes B'} \leq\otimes \qquad \frac{A \leq A' \quad B \leq B'}{A' \to B \leq A \to B'} \leq\to$$

$$\frac{S \subseteq T \quad \{A_\ell \leq B_\ell\}_{\ell \in S}}{\oplus\{\ell : A_\ell\}_{\ell \in S} \leq \oplus\{\ell : B_k\}_{k \in T}} \leq\oplus \qquad \frac{T \subseteq S \quad \{A_k \leq B_k\}_{k \in T}}{\&\{\ell : A_\ell\}_{\ell \in S} \leq \&\{\ell : B_k\}_{k \in T}} \leq\&$$

$$\frac{}{A \leq A}\ \text{refl} \qquad \frac{A \leq B \quad B \leq C}{A \leq C}\ \text{trans}$$

Figure 2.5: SAX Subtyping

2.3 Operational Metatheory

In this section, we introduce the operational semantics of SAX in terms of *configurations* of processes and the futures with which they communicate. In preparation for the operational metatheory of the subsequent chapters, we establish syntactic type soundness of *configuration typing* with respect to *configuration reduction* as well as termination of the latter.

2.3.1 Configuration Reduction and Typing

Definition 2.3. *Configurations* $\mathcal{C}, \ldots$ are multisets generated by the following grammar.

$\mathcal{C} :=$	$\mathbf{proc}\, P$	process P
$\mid$	$!\mathbf{future}\, a\, S$	future at address a with contents S
$\mid$	$\cdot$	empty configuration
$\mid$	$\mathcal{C}, \mathcal{C}'$	union of two configurations

A configuration $\mathcal{F}$ is *final* iff it only contains futures.

$$
\begin{array}{lcll}
!\mathbf{future}\, a\, S, \mathbf{proc}\,(\mathtt{copy}\, b\, a) & \mapsto & !\mathbf{future}\, b\, S & \\
\mathbf{proc}\,(x \leftarrow P\,(x)\,; Q\,(x)) & \mapsto & \mathbf{proc}\,(P\,(a))\,, \mathbf{proc}\,(Q\,(a)) & (a \text{ fresh}) \\
!\mathbf{future}\, a\, V, \mathbf{proc}\,(\mathtt{read}\, a\, K) & \mapsto & \mathbf{proc}\,(V \triangleright K) & \\
!\mathbf{future}\, a\, K, \mathbf{proc}\,(\mathtt{read}\, a\, V) & \mapsto & \mathbf{proc}\,(V \triangleright K) & \\
\mathbf{proc}\,(\mathtt{write}\, a\, S) & \mapsto & !\mathbf{future}\, a\, S & \\
\mathbf{proc}\,(f\,(\overline{a})) & \mapsto & \mathbf{proc}\,(P\,(\overline{a})) & (f\,(\overline{x}) = P\,(\overline{x}))
\end{array}
$$

$$
\begin{array}{lcllll}
\langle\rangle & \triangleright & (\langle\rangle \Rightarrow P) & = & P & (\mathbf{1}) \\
\langle a, b\rangle & \triangleright & (\langle x, y\rangle \Rightarrow P\,(x, y)) & = & P\,(a, b) & (\otimes/\rightarrow) \\
k \cdot a & \triangleright & \{\ell \cdot x \Rightarrow P_\ell\,(x)\}_{\ell \in S} & = & P_k\,(a) & (\oplus/\&, k \in S)
\end{array}
$$

Figure 2.6: SAX Configuration Reduction and Principal Cut Reduction

Configuration reduction ($\mapsto$) is defined by the *multiset rewriting rules* [CS06] in Figure 2.6, which replace any subset of a configuration matching the left-hand side with the right-hand side; ! indicates objects that persist across reduction. All rules are standard, with reads being mediated by *principal cut reduction* $V \triangleright K$, justifying the computational interpretation of the logical rules from the previous section. Before we elaborate on the metatheory, we extend non-bidirectional process typing to configurations. The *configuration typing* judgment $\Sigma \vdash \mathcal{C} \div \Delta$ is inductively generated by the rules in Figure 2.7, which types the objects in $\mathcal{C}$ with sources in Σ and destinations in Δ. The peculiar design of the proc rule enables Lemma 2.2 in the next subsection.

2.3.2 Syntactic Type Soundness

In preparation for our proof of partial correctness in Chapter 4, which is typically established as a corollary of type soundness [JV21], we prove syntactic type soundness

$$\text{runtime contexts } \Sigma, \Delta := \cdot \mid \Sigma, a : A$$
$$A \leq B \text{ extends pointwise to } \Sigma \leq \Delta$$

$$\frac{\Sigma \leq \Sigma' \quad \Sigma' \vdash P \div (a : A') \quad A' \leq A}{\Sigma \vdash \mathbf{proc}\, P \div (\Sigma, a : A)}\ \text{proc} \qquad \frac{\Sigma \vdash \mathbf{proc}\, (\mathtt{write}\, a\, S) \div \Delta}{\Sigma \vdash\, !\mathbf{future}\, a\, S \div \Delta}\ \text{fut}$$

$$\frac{}{\Sigma \vdash (\cdot) \div \Sigma}\ \text{empty} \qquad \frac{\Sigma \vdash \mathcal{C} \div \Sigma' \quad \Sigma' \vdash \mathcal{C}' \div \Delta}{\Sigma \vdash \mathcal{C}, \mathcal{C}' \div \Delta}\ \text{join}$$

Figure 2.7: SAX Configuration Typing

of configuration typing with respect to reduction by a standard appeal to *progress* and *preservation* [WF94]. To simplify our argument, we first restrict the configuration typing derivations analyzed below.

Lemma 2.2 (SAX Configuration Typing Induction)**.** *Without loss of generality, it suffices to consider configuration typing derivations where:*

- Right-to-left induction*: every instance of the join rule has exactly one object in the right-hand configuration, because non-compliant such instances can be reassociated. Thus, induction can begin with the right-most object with the induction hypothesis applying to the remaining subconfiguration on the left.*

- Process typing inversion I*: the process typing derivation underneath any instance of the proc rule ends in a non-subsumption rule, because terminal instances of subsumption can be absorbed into the proc rule's subtyping premises using transitivity of subtyping. Thus, inversion on the typing derivations for processes writing to and reading from the same address*

end in right and left rule instances for the same type constructor, respectively. Because, from left to right, the writer and readers ascribe said address types $A \leq \ldots \leq B$ by transitivity of subtyping. Yet, if $A \leq B$, then A and B have the same head constructor (by a routine induction on the subtyping derivation).

Lemma 2.3 (SAX Progress)**.** *If $\cdot \vdash \mathcal{C} \div \Delta$ then either $\mathcal{C}$ is final or $\mathcal{C}$ steps.*

Proof. By right-to-left induction.

- If $\mathcal{C} = \cdot$, then it is trivially final.
- If $\mathcal{C} = \mathcal{C}_1, !\mathbf{future}\, a\, S$, then by the induction hypothesis, either $\mathcal{C}_1$ is final, in which case $\mathcal{C}$ is final, or $\mathcal{C}_1 \mapsto \mathcal{C}_1'$, in which case $\mathcal{C} \mapsto \mathcal{C}_1', !\mathbf{future}\, a\, S$.
- If $\mathcal{C} = \mathcal{C}_1, \mathbf{proc}\, P$, then by the induction hypothesis, either $\mathcal{C}_1 \mapsto \mathcal{C}_1'$, in which case $\mathcal{C} \mapsto \mathcal{C}_1', \mathbf{proc}\, P$, or $\mathcal{C}_1$ is final. Then, we proceed by cases of P.
  - If P is a cut, write (right rule) or definition call, then $\mathcal{C}$ steps by virtue of P.
  - If $P = \mathtt{copy}\, b\, a$, because $!\mathbf{future}\, a\, S \in \mathcal{C}_1$, we have $\mathcal{C} \mapsto \mathcal{C}, !\mathbf{future}\, b\, S$.
  - If P is typed by a positive left rule, for example $P = \mathtt{read}\, c\ (\langle x, y\rangle \Rightarrow P_1\,(x, y))$, then by inversion on the process typing derivation for $!\mathbf{future}\, c\, S \in \mathcal{C}_1$, we have $S = \langle a, b\rangle$, so $\mathcal{C} \mapsto \mathcal{C}_1, \mathbf{proc}\,(P_1\,(a, b))$.
  - If P is typed by a negative left axiom, for example $P = \mathtt{read}\, c\, \langle a, b\rangle$, then by inversion on the process typing derivation for $!\mathbf{future}\, c\, S \in \mathcal{C}_1$, we have $S = \langle x, y\rangle \Rightarrow P_1\,(x, y)$, so $\mathcal{C} \mapsto \mathcal{C}_1, \mathbf{proc}\,(P_1\,(a, b))$. □

Lemma 2.4 (SAX Preservation)**.** *If $\Sigma \vdash \mathcal{C} \div \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$, then $\Sigma \vdash \mathcal{C}' \div \Delta'$ for $\Delta' \supseteq \Delta$.*

Proof. By cases of the reduction step and then inversion on the configuration typing derivation.

- Identity: the process typing derivation for the source address is copied for the destination.
- Cut: the instance of the cut rule converts to one of the join rule.
- Read: the process typing derivation for the reading process is replaced by that of the continuation in question.
- Write: trivial, because typing of futures is defined via that of writes.
- Definition call: trivial, because typing of a call is defined via that of the body of the associated definition. □

Theorem 2.2 (SAX Type Soundness)**.** *A configuration C is safe iff it is final or, coinductively, $\mathcal{C} \mapsto \mathcal{C}'$ and $\mathcal{C}'$ is safe. Then, if $\cdot \vdash \mathcal{C} \div \Delta$, then $\mathcal{C}$ is safe.*

Proof. See [LG09] for coinductive characterizations of type soundness. We proceed by coinduction generalizing Δ; by Lemma 2.3, either $\mathcal{C}$ is final or $\mathcal{C} \mapsto \mathcal{C}'$. By Lemma 2.4, $\cdot \vdash \mathcal{C}' \div \Delta'$ for some $\Delta' \supseteq \Delta$, so we are done by the coinduction hypothesis. □

2.3.3 Termination

In preparation for our proof of termination for IRSAX in Chapter 3, we establish termination of SAX using the standard modification of Tait's method for Kripke logical relations [Plo73]. This simplifies the arguments given in [DPP20, SP22].

- Unlike [DPP20], we isolate the definition of *semantic types*, i.e., the interpretation of types generated by the logical relation (Definition 2.5). In short, they are world-indexed sets of terminating storables where the worlds are terminating configurations. In the next chapter, we show that recursive types of certain shapes are interpreted as fixed points of semantic type constructors. Therefore, they can be accurately described as (co)inductive types.

- *Kripke monotonicity* is used to extend the type ascriptions of subconfigurations to the entire configuration. Then, by establishing soundness of subtyping and process typing (Lemma 2.6 and Lemma 2.7) with respect to *semantic (sub)typing* (Definition 2.6), termination is entailed by the *fundamental theorem of the logical relation* induced by the interpretation of types (Theorem 2.3).

Definition 2.4. A *semantic type* $\mathscr{A}, \mathscr{B}, \mathscr{C}, \ldots$ is a pair of storables and final configurations, writing $\mathcal{F} \vDash S \in \mathscr{A}$ for $(S, F) \in \mathscr{A}$, such that:

- *Kripke monotonicity*: if $\mathcal{F} \vDash S \in \mathscr{A}$, then $\mathcal{F}, \mathcal{F}' \vDash S \in \mathscr{A}$ for all $\mathcal{F}'$.

Then, $\mathcal{F} \vDash a \in \mathscr{A}$ iff there exists $!\mathbf{future}\, a\, S \in \mathcal{F}$ such that $\mathcal{F} \vDash S \in \mathscr{A}$. Lastly, $\mathcal{C} \vDash a \in \mathscr{A}$ iff, inductively, for all reducts $\mathcal{C}'$ of $\mathcal{C}$, $\mathcal{C}' \vDash a \in \mathscr{A}$.

Definition 2.5 (Semantic Interpretation)**.** $[\![A]\!]$ is a semantic type defined by induction on A as follows, with the cases of positive and negative types respectively classifying values and continuations. Kripke monotonocity in each case is immediate. As is standard for Kripke semantics, we omit "$\mathcal{F} \vDash$" when its presence is obvious.

- $\langle\rangle \in [\![\mathbf{1}]\!]$ only.

- $\langle a, b\rangle \in [\![A \otimes B]\!]$ iff $a \in [\![A]\!]$ and $b \in [\![B]\!]$.
- $k \cdot a \in [\![\oplus \{\ell : A_\ell\}_{\ell \in S}]\!]$ iff $k \in S$ and $a \in [\![A_k]\!]$.
- $\mathcal{F} \vDash \langle x, y\rangle \Rightarrow P(x, y) \in [\![A \to B]\!]$ iff for all $\mathcal{F}' \supseteq \mathcal{F}$ such that $\mathcal{F}' \vDash a \in [\![A]\!]$, then $\mathcal{F}', \mathbf{proc}(P(a, b)) \vDash b \in [\![B]\!]$ for fresh a and b.
- $\mathcal{F} \vDash \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell \in S} \in [\![\& \{\ell : A_\ell\}_{\ell \in S}]\!]$ iff for all $\ell \in S$, then $\mathcal{F}, \mathbf{proc}(P_\ell(a)) \vDash a \in [\![A_\ell]\!]$ for fresh a.

Definition 2.6 (Semantic (Sub)typing)**.**

- $\mathcal{C} \vDash \Sigma \triangleq \mathcal{C} \vDash a \in [\![A]\!]$ for all $a : A \in \Sigma$.
- $\Sigma \vDash \mathcal{C} \div \Delta$ iff for all $\mathcal{F}$ such that $\mathcal{F} \vDash \Sigma$, then $\mathcal{F}, \mathcal{C} \vDash \Delta$.
- $A \subseteq B \triangleq [\![A]\!] \subseteq [\![B]\!]$, extending to $\Sigma \subseteq \Delta$ in the obvious way.

In addition to backward closure, semantic typing respects the monoidal structure of configurations as well as semantic subtyping.

Lemma 2.5 (Semantic Typing Properties)**.**

- Semantic empty*:* $\Sigma \vDash \cdot \div \Sigma$*.*
- Semantic join*: if* $\Sigma \vDash \mathcal{C} \div \Sigma'$ *and* $\Sigma' \vDash \mathcal{C}' \div \Delta$*, then* $\Sigma \vDash \mathcal{C}, \mathcal{C}' \div \Delta$*.*
- Semantic subsumption*: if* $\Sigma \subseteq \Sigma'$*,* $\Sigma' \vDash \mathcal{C} \div \Delta'$*, and* $\Delta' \subseteq \Delta$*, then* $\Sigma \vDash \mathcal{C} \div \Delta$*.*
- Backward closure*: if* $\Sigma \vDash \mathcal{C}' \div \Delta$ *for all reducts* $\mathcal{C}'$ *of* $\mathcal{C}$*, then* $\Sigma \vDash \mathcal{C} \div \Delta$*.*

Proof. Only the join rule is not straightforward—assume $\mathcal{F} \vDash \Sigma$, we want to show $\mathcal{F}, \mathcal{C}, \mathcal{C}' \vDash \Delta$. We have $\mathcal{F}, \mathcal{C} \vDash \Sigma'$ by the first assumption, i.e., $\mathcal{F}, \mathcal{C} \vDash a : A$ for each $a : A \in \Sigma'$. By induction, either $\mathcal{F}, \mathcal{C}$ is final and we are done by the second assumption, or for all $\mathcal{C}_1$ where $\mathcal{F}, \mathcal{C} \mapsto \mathcal{F}, \mathcal{C}_1$. □

Lemma 2.6 (SAX Subtyping Soundness)**.** *If $A \leq B$ then $A \subseteq B$.*

Proof. By a routine induction on the subtyping derivation D, we show some representative cases.

- *Eager pairs*: $D = \dfrac{\begin{array}{cc} \overset{D_1}{\vdots} & \overset{D_2}{\vdots} \\ A \leq A' & B \leq B' \end{array}}{A \otimes B \leq A' \otimes B'} \leq\otimes$

  Assuming $\langle a, b \rangle \in [\![A \otimes B]\!]$, i.e., $a \in [\![A]\!]$ and $b \in [\![B]\!]$, we want to show $a \in [\![A']\!]$ and $b \in [\![B']\!]$. It suffices to show $A \subseteq A'$ and $B \subseteq B'$, which we have by the induction hypotheses for D_1 and D_2.

- *Functions*: $D = \dfrac{\begin{array}{cc} \overset{D_1}{\vdots} & \overset{D_2}{\vdots} \\ A \leq A' & B \leq B' \end{array}}{A' \to B \leq A \to B'} \leq\to$

  Assuming $\mathcal{F} \vDash \langle x, y \rangle \Rightarrow P\,(x, y) \in [\![A \to B]\!]$, we want to show for all $\mathcal{F}' \supseteq \mathcal{F}$, if $\mathcal{F}' \vDash a \in [\![A]\!]$, then $\mathcal{F}', \mathbf{proc}\,(P(a, b)) \vDash b \in [\![B']\!]$ for fresh a and b. By the induction hypothesis for D_1, which gives $A \subseteq A'$, we have $\mathcal{F}' \vDash a \in [\![A']\!]$, so $\mathcal{F}', \mathbf{proc}\,(P(a, b)) \vDash b \in [\![B]\!]$. It suffices to show $B \subseteq B'$, which we have by the induction hypothesis for D_2. □

Lemma 2.7 (SAX Process Typing Soundness)**.** *If* $\Sigma \vdash P \div (a : A)$*, then* $\Sigma \vDash \mathbf{proc}\, P \div \Sigma, a : A$*.*

Proof. Assuming $\mathcal{F} \vDash \Sigma$ and letting $\mathcal{C} \triangleq \mathcal{F}, \mathbf{proc}\, P$, it suffices to show $\mathcal{C} \vDash \Sigma, a : A$ by induction on the process typing derivation.

- *Subsumption*: by the induction hypothesis and then semantic subsumption.
- *Definition call*: by the induction hypothesis on the body of the associated definition and then backward closure.
- *Identity*: $D = \dfrac{}{\Sigma', a : A \vdash \mathtt{copy}\, b\, a \div (b : A)}\ \mathrm{id}$

  Because $\mathcal{F} \vDash a : A$, we have $!\mathbf{future}\, a\, S \in \mathcal{F}$ such that $\mathcal{F} \vDash S \in [\![A]\!]$. Moreover, $\mathcal{F}'$ is the only reduct of $\mathcal{C}$ where $\mathcal{F}' \triangleq \mathcal{F}, !\mathbf{future}\, b\, S$. By backward closure, it suffices to show $\mathcal{F}' \vDash \Sigma, b : A$, which we have by definition of $\mathcal{F}' \vDash b : A$ and then Kripke monotonicity.
- *Cut*: $D = \dfrac{\begin{array}{cc} \overset{D_1}{\vdots} & \overset{D_2}{\vdots} \\ \Sigma \vdash P(x) \div (x : A) & \Sigma, x : A \vdash Q(x) \div \gamma \end{array}}{\Sigma \vdash x \leftarrow P(x)\,;\, Q(x) \div \gamma}\ \mathrm{cut}$

  Generalizing over $\mathcal{F}$, by induction on $[a/x]D_1$ and $[a/x]D_2$ for fresh a then semantic join, we have $\Sigma \vdash \mathbf{proc}\,(P(a))\,, \mathbf{proc}\,(Q(a)) \vDash \Sigma, \gamma$. Because $\mathbf{proc}\, P \mapsto \mathbf{proc}\,(P(a))\,, \mathbf{proc}\,(Q(a))$ is the only possible reduction, we are done by backward closure.
- *Right axiom*: for example, $D = \dfrac{}{\Sigma', a : A, b : B \vdash \mathtt{write}\, c\, \langle a, b \rangle \div (c : A \otimes B)}\ \otimes\mathrm{X}$.

Letting $\mathcal{F}' \triangleq \mathcal{F}, !\mathbf{future}\, c\, \langle a, b\rangle$, we have $\mathcal{F}' \vDash c : A \otimes B$ because $\mathcal{F}' \vDash a : A$ and $\mathcal{F}' \vDash b : B$ by Kripke monotonicity. Thus, $\mathcal{F}' \vDash \Sigma, c \in A \otimes B$ by Kripke monotonicity again. Because $\mathcal{F}'$ is the only reduct of $\mathcal{C}$, we are done by backward closure.

- *Positive left rule*: for example, $D = \dfrac{\begin{array}{c}D' \\ \vdots \\ \Sigma', c : A \otimes B, x : A, y : B \vdash P\,(x, y) \div \gamma\end{array}}{\Sigma', c : A \otimes B \vdash \mathtt{read}\, c\ (\langle x, y\rangle \Rightarrow P\,(x, y)) \div \gamma}\ \otimes\mathrm{L}$.

Because $\mathcal{F} \vDash c : A \otimes B$, we have $!\mathbf{future}\, c\, \langle a, b\rangle \in \mathcal{F}$ such that $\mathcal{F} \vDash a : A$ and $\mathcal{F} \vDash b : B$. By induction on $[a/x][b/y]D'$, we have $\mathcal{C}' \vDash \Sigma, \gamma$ where $\mathcal{C}' \triangleq \mathcal{F}, \mathbf{proc}\,(P\,(a, b))$. Because $\mathcal{C}'$ is the only reduct of $\mathcal{C}$, we are done by backward closure.

- *Negative right rule*: for example, let $D = \dfrac{\begin{array}{c}D' \\ \vdots \\ \Sigma, x : A \vdash P\,(x, y) \div (y : B)\end{array}}{\Sigma \vdash \mathtt{write}\, c\ (\langle x, y\rangle \Rightarrow P(x, y)) \div (c : A \to B)}\ \to\mathrm{R}$.

By backward closure then Kripke monotonicity, it suffices to show $\mathcal{F}' \vDash c : A \to B$ where $\mathcal{F}' \supseteq \mathcal{F}, !\mathbf{future}\, c\ (\langle x, y\rangle \Rightarrow P(x, y))$. Assuming $\mathcal{F}' \vDash a : A$, we have $\mathcal{F}' \vDash \Sigma, a : A$ by Kripke monotonicity, so $\mathcal{F}', \mathbf{proc}\,(P(a, b)) \vDash \Sigma, a : A, b : B$ by the induction hypothesis on $[a/x][b/y]D'$ for fresh a and b, as desired.

- *Left axiom*: for example, let $D = \dfrac{}{\Sigma', c : A \to B, a : A \vdash \mathtt{read}\, c\, \langle a, b\rangle \div (b : B)}\ \to\mathrm{L}$.

Because both $\mathcal{F} \vDash c : A \to B$ and $\mathcal{F} \vDash a : A$, we have $\mathcal{F}, \mathtt{read}\, c\, \langle a, b\rangle \vDash \Sigma, b : B$ by backward closure. □

Theorem 2.3 (SAX Fundamental Theorem)**.** *If* $\Sigma \vdash \mathcal{C} \div \Delta$, *then* $[\![\Sigma]\!] \vDash \mathcal{C} \div [\![\Delta]\!]$.

Proof. By induction, the empty and join cases are discharged by their semantic counterparts. The proc case and, by backward closure, the fut case are handled by Lemma 2.7, using Lemma 2.6 then semantic subsumption to factor through the subtyping premises. □

2.4 Related Work

SAX is an asynchronous process calculus with name passing (via eager pairs and functions) and branching/selection (via labelled sums and lazy records) with both futures-based and message-passing operational interpretations [PP21], complimenting previous work on assigning session types to an asynchronous π-calculus [DCPT12]. Moreover, the distinction between positive and negative types is related to call-by-push-value [Lev99]. Lastly, our syntax and semantics respectively draws from backwards bidirectional typing [Zei15, DK21] and Kripke logical relations [Plo73, BHN14].

Chapter 3

Index Refinements

...logic is not valid. — Jon Sterling

In this chapter, we develop Index Refined SAX (IRSAX) by adding index refinements to SAX. Aside from their application to termination checking as discussed in the introduction, index refinements introduce a limited form of type dependency to exclude illegal program states; consider the following example.

Example 3.1 (Index Refinements for Dimensional Analysis)**.** If our index domain is taken to be the mathematical structure generated by some system of base units, then physical quantities can be taken as values inhabiting dimension-indexed types: $x_0 : \mathbb{R}[\text{meters}]$ indicates a real-valued positional quantity x_0. As a result, there cannot be any errors due to implicit coercion between incompatible units. However, the user may hard-code, as part of a library, explicit coercions between units for the same physical phenomenon. For example, a value of $\mathbb{R}[\text{meters}]$ may be used as $\mathbb{R}[\text{inches}]$ through explicit conversion [Dun07].

With the principled addition of type and process recursion, IRSAX implements the first half of total correctness type refinements: termination checking via the sized type refinements of [SP22].

- In Section §3.1, we discuss the modifications made to the judgmental structure of SAX induced by an arithmetic index domain consisting of *propositions* (arithmetic formulas) and *expressions* (arithmetic terms).

- In Section §3.2, we adapt certain *property types* (*guarded*, *asserting*, *index universal*, and *index existential* types), originally introduced by Dunfield and Pfenning in natural deduction style [DP03], to SAX in preparation for the next section.

- In Section §3.3, we reformulate the sized type refinements of [SP22] in this framework and uniformly provide compositional termination checking for induction, coinduction, and mixed induction and coinduction. Like *op. cit.*, we make comprehensive use of *infinite proofs* to simplify our account of type and process recursion, as opposed to explicit formulations involving, e.g., fixed point rules. In particular, we initially take a mixed inductive-coinductive view of (sub)typing called IRSAX ∞, which is syntax-directed, moving then to a non-bidirectional system with constructive ω-rules [Yoc89] called IRSAX ω to establish the metatheory. As a sanity check, we show that our metatheory interprets recursive types as (co)inductive semantic types.

(all grammars and judgments from Figure 2.2 included)

$$\text{contexts } \Gamma := \ldots \mid \Gamma, i \mid \Gamma, \phi$$
$$\text{subtyping } \Gamma \vdash A \leq B$$

Figure 3.1: IRSAX Judgments

3.1 Judgmental Structure

Because the index domain is separate from SAX, the process typing judgment admits minimal modifications. In particular, contexts may now include *propositions* ϕ from the index domain as well as *index variables* i standing for elements of said domain. For the purposes of this chapter, we limit our attention to *arithmetic refinements* [DP20a, DP20b]. Lastly, index dependency of the new type constructors added in the next section requires subtyping to have access to the typing context. All modifications are given in Figure 3.1.

3.2 Property Types

The extended syntax for types, which include *index expressions* and propositions, is given in Figure 3.2 with full (sub)typing rules in Figure 3.3 and Figure 3.4. In short, the asserting type $\phi \wedge A$ classifies futures of type A while *asserting* ϕ, whereas the guarded type $\phi \supset A$ is the adjoint notion of implication: classifying futures of type A while *assuming* ϕ. On the other hand, the index universal and existential types correspond to standard first-order quantifiers over the index domain. Under a Curry-Howard interpretation, they are *index-dependent* types.

(all grammars from Figure 2.2 included)

$$\begin{array}{rl} \text{index expression } e, f := & i \mid n \in \mathbb{N} \mid e - f \mid \ldots \\ \text{proposition } \phi := & e > f \mid \ldots \end{array}$$

$$\begin{array}{rll} \text{type } A := & \ldots & \\ \mid & \phi \wedge A & \text{asserting type} \\ \mid & \phi \supset A & \text{guarded type} \\ \mid & \exists i.\, A\,(i) & \text{index existential} \\ \mid & \forall i.\, A\,(i) & \text{index universal} \end{array}$$

Figure 3.2: IRSAX Syntax

3.2.1 Property Types

To set the tone of our discussion, let us begin by developing typing rules for $\phi \wedge A$—because it classifies futures of type A where ϕ is true, its right rule ought to be the right rule for A along with a check that ϕ is true. Dually, the left rule should be that for A with ϕ as an assumption added to the context. Taken together, we have the following (sub)typing rules.

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \texttt{write}\, t\, S \div (t \Leftarrow A)}{\Gamma \vdash \texttt{write}\, t\, S \div (t \Leftarrow \phi \wedge A)}\ \wedge\mathrm{R} \qquad \frac{\Gamma, \phi, t \Rightarrow A \vdash \texttt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \phi \wedge A \vdash \texttt{read}\, t\, S \div \gamma}\ \wedge\mathrm{L}$$

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq \phi \wedge B}\ \leq\wedge\mathrm{R} \qquad \frac{\Gamma, \phi \vdash A \leq B}{\Gamma \vdash \phi \wedge A \leq B}\ \leq\wedge\mathrm{L}$$

Because the right rule types the process $\texttt{write}\, t\, S$, which scrutinizes the succedent t, the next rule in the derivation must be the right rule/axiom for A on t while asserting ϕ. This is analogous to the restriction of property type introduction rules to "checking intro. forms" in the sequent calculus-style system of Dunfield and Krishnaswami, which

itself corresponds to the *value restriction* in natural deduction [DP00]. Likewise, the left rule types $\texttt{read}\, t\, S$, deferring to the left rule/axiom for A on t with the assumption of ϕ. Curiously, this is analogous to restricting eliminations of property types in natural deduction to terms where the scrutinee is next in line to be evaluated. This is perhaps better demonstrated by example; consider the elimination rule for type union due to Dunfield and Pfenning below—the scrutinee e is guaranteed to be evaluated immediately because it is surrounded by an evaluation context E.

$$\frac{\Gamma \vdash e : A \vee B \quad \Gamma, x : A \vdash E[x] : C \quad \Gamma, x : B \vdash E[x] : C}{\Gamma \vdash E[e] : C} \vee\text{E}$$

Lastly, the subtyping rules read $\Gamma \vdash A \leq B$ as a sequent with exactly one antecedent and succedent, thus becoming singleton versions of these logical rules [Dun07]. Now, the rules for the guarded type $\phi \supset A$ follow symmetrically with the assumption and assertion of ϕ in opposite roles.

$$\frac{\Gamma, \phi \vdash \texttt{write}\, t\, S \div (t \Leftarrow A)}{\Gamma \vdash \texttt{write}\, t\, S \div (t \Leftarrow \phi \supset A)} \supset\text{R} \quad \frac{\Gamma \vdash \phi \quad \Gamma, t \Rightarrow A \vdash \texttt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \phi \supset A \vdash \texttt{read}\, t\, S \div \gamma} \supset\text{L}$$

$$\frac{\Gamma, \phi \vdash A \leq B}{\Gamma \vdash A \leq \phi \supset B} \leq\supset\text{R} \quad \frac{\Gamma \vdash \phi \quad \Gamma \vdash A \leq B}{\Gamma \vdash \phi \supset A \leq B} \leq\supset\text{L}$$

We seem to now have a recipe for developing the rules for property types:

- The right/left rule for a property type defers to the right/left rule for the underlying type refined
- Their subtyping rules are "singleton" versions of their logical rules

Let us apply this script to index existential and universal type constructors; there's a slight catch: the non-invertible rules seem to conjure an index term e out of nowhere!

$$\frac{\Gamma \vdash e \quad \Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow A\,(e))}{\Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow \exists i.\, A\,(i))}\ \exists\mathrm{R} \qquad \frac{\Gamma, i, t \Rightarrow A\,(i) \vdash \mathtt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \exists i.\, A\,(i) \vdash \mathtt{read}\, t\, S \div \gamma}\ \exists\mathrm{L}$$

$$\frac{\Gamma, i \vdash \mathtt{write}\, t\, S \div (t \Leftarrow A\,(i))}{\Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow \forall i.\, A\,(i))}\ \forall\mathrm{R} \qquad \frac{\Gamma \vdash e \quad \Gamma, t \Rightarrow A\,(e) \vdash \mathtt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \forall i.\, A\,(i) \vdash \mathtt{read}\, t\, S \div \gamma}\ \forall\mathrm{L}$$

$$\frac{\Gamma \vdash e \quad \Gamma \vdash A \leq B\,(e)}{\Gamma \vdash A \leq \exists i.\, B\,(i)}\ \leq\exists\mathrm{R} \qquad \frac{\Gamma, i \vdash A\,(i) \leq B}{\Gamma \vdash \exists i.\, A\,(i) \leq B}\ \leq\exists\mathrm{L}$$

$$\frac{\Gamma, i \vdash A \leq B\,(i)}{\Gamma \vdash A \leq \forall i.\, B\,(i)}\ \leq\forall\mathrm{R} \qquad \frac{\Gamma \vdash e \quad \Gamma \vdash A\,(e) \leq B}{\Gamma \vdash \forall i.\, A\,(i) \leq B}\ \leq\forall\mathrm{L}$$

Dunfield notes that, in practice, e would be replaced by a unification variable to be instantiated by an external solver or algorithmic typing, which is fleshed out in [DK19]. In the next section, we will permit process definitions to additionally abstract over index variables. Thus, the extended call rule can *output* types of the form $A\,(e)$ that contain index expressions, corresponding to the *contextual typing annotations* of Dunfield and Pfenning [DP04], which has the effect of instantiation. We now have the components required to discuss sized type refinements.

3.2.2 Typing Summary

To close this section, we tabulate the bidirectional typing presented into Figure 3.3 and Figure 3.4.

(all rules from Figure 2.4 included)

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow A)}{\Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow \phi \wedge A)}\ \wedge\mathrm{R} \qquad \frac{\Gamma, \phi, t \Rightarrow A \vdash \mathtt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \phi \wedge A \vdash \mathtt{read}\, t\, S \div \gamma}\ \wedge\mathrm{L}$$

$$\frac{\Gamma, \phi \vdash \mathtt{write}\, t\, S \div (t \Leftarrow A)}{\Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow \phi \supset A)}\ \supset\mathrm{R} \qquad \frac{\Gamma \vdash \phi \quad \Gamma, t \Rightarrow A \vdash \mathtt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \phi \supset A \vdash \mathtt{read}\, t\, S \div \gamma}\ \supset\mathrm{L}$$

$$\frac{\Gamma \vdash e \quad \Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow A\,(e))}{\Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow \exists i.\, A\,(i))}\ \exists\mathrm{R} \qquad \frac{\Gamma, i, t \Rightarrow A\,(i) \vdash \mathtt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \exists i.\, A\,(i) \vdash \mathtt{read}\, t\, S \div \gamma}\ \exists\mathrm{L}$$

$$\frac{\Gamma, i \vdash \mathtt{write}\, t\, S \div (t \Leftarrow A\,(i))}{\Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow \forall i.\, A\,(i))}\ \forall\mathrm{R} \qquad \frac{\Gamma \vdash e \quad \Gamma, t \Rightarrow A\,(e) \vdash \mathtt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \forall i.\, A\,(i) \vdash \mathtt{read}\, t\, S \div \gamma}\ \forall\mathrm{L}$$

Figure 3.3: IRSAX Typing

(all rules from Figure 2.5 modified to include Γ)

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq \phi \wedge B}\ \leq\wedge\mathrm{R} \qquad \frac{\Gamma, \phi \vdash A \leq B}{\Gamma \vdash \phi \wedge A \leq B}\ \leq\wedge\mathrm{L}$$

$$\frac{\Gamma, \phi \vdash A \leq B}{\Gamma \vdash A \leq \phi \supset B}\ \leq\supset\mathrm{R} \qquad \frac{\Gamma \vdash \phi \quad \Gamma \vdash A \leq B}{\Gamma \vdash \phi \supset A \leq B}\ \leq\supset\mathrm{L}$$

$$\frac{\Gamma \vdash e \quad \Gamma \vdash A \leq B\,(e)}{\Gamma \vdash A \leq \exists i.\, B\,(i)}\ \leq\exists\mathrm{R} \qquad \frac{\Gamma, i \vdash A\,(i) \leq B}{\Gamma \vdash \exists i.\, A\,(i) \leq B}\ \leq\exists\mathrm{L}$$

$$\frac{\Gamma, i \vdash A \leq B\,(i)}{\Gamma \vdash A \leq \forall i.\, B\,(i)}\ \leq\forall\mathrm{R} \qquad \frac{\Gamma \vdash e \quad \Gamma \vdash A\,(e) \leq B}{\Gamma \vdash \forall i.\, A\,(i) \leq B}\ \leq\forall\mathrm{L}$$

Figure 3.4: IRSAX Subtyping

3.3 Sized Type Refinements

Adding (co)inductive types and terminating recursion, including productive corecursive definitions, to any programming language is a non-trivial task, because only certain recursive programs constitute valid applications of (co)induction principles. Briefly, inductive

calls must occur on data smaller than the input and, dually, coinductive calls must be guarded by further codata output. In either case, we are concerned with the decrease of (co)data size—height of data and observable depth of codata—in a sequence of recursive calls. Because inferring this exactly is intractable, languages like Agda (before version 2.4) and Coq resort to conservative syntactic criteria like the *guardedness check*.

One solution that avoids syntactic checks is to track the flow of (co)data size at the type level with *sized types*, as pioneered by Hughes et al. [HPS96] and further developed by others [BFG+04, Bla04, Abe06, AP13]. Inductive and coinductive types are indexed by the height and observable depth of their data and codata, respectively. Considering a signature of mutually recursive indexed type definitions $X\,[i] = A\,[i]$, the example below adorns them with *sized type refinements*: $\mathtt{nat}[i]$ describes unary natural numbers less than i and $\mathtt{str}[i]$ describes infinite bitstreams that allow the first i elements to be observed before reaching undefined (e.g., potentially divergent) behavior. Note that we use the term "recursive type" as a catch all for (mixed) (co)inductive types.

Example 3.2 (Recursive Types)**.**

$$\mathtt{nat}[i] = i > 0 \wedge \oplus\{\mathtt{zero} : \mathbf{1}, \mathtt{succ} : \mathtt{nat}[i-1]\}$$

$$\mathtt{str}[i] = i > 0 \supset \&\{\mathtt{head} : \mathtt{bool}, \mathtt{tail} : \mathtt{str}[i-1]\}$$

The $\mathtt{succ}$ branch of $\mathtt{nat}\,[i]$ produces a natural number at height $i-1$ *asserting* $i > 0$ whereas the $\mathtt{tail}$ branch of $\mathtt{str}[i]$ can produce the remainder of the stream at depth $i-1$ *assuming* $i > 0$. Starting from $\mathtt{nat}[i]$, recurring *on*, for example, $\mathtt{nat}[i-1]$ ($i > 0$ is assumed

during *elimination* so that $i - 1$ is well-defined) produces the size sequence $i > i - 1 > i - 2 > \ldots$ that eventually terminates at 0, agreeing with the (strong) induction principle for natural numbers. Dually, starting from $\texttt{str}[i]$, recurring *into* $\texttt{str}[i - 1]$ (again, $i > 0$ is assumed during *introduction* so that $i - 1$ is well-defined) produces the same well-founded sequence of sizes, agreeing with the coinduction principle for streams. In either case, a recursive program terminates if its call graph generates a well-founded sequence of sizes in each code path. Moreover, index quantification can be used to abstract over size:

Example 3.3 (Full (Co)inductive Types)**.** $\exists i.\ \texttt{nat}[i]$ is the type of *all* natural numbers, whereas $\forall i.\ \texttt{str}[i]$ is the type of bitstreams of arbitrary depth, i.e., infinite streams.

In summary, sized type refinements combine the "good parts" of contemporary sized type systems:

- Like [Xi01], we view termination checking as a usage mode of the property types defined in the previous section, which affords both richer size arithmetic and more flexible size abstraction than most contemporary sized type systems (Examples 3.5 and 3.6).

- Like the bounded quantifiers of [Abe12], the dual behavior of asserting and guarded types during elimination and introduction encodes induction and coinduction, respectively, which not only avoids *type continuity* restrictions during recursion [Abe06], but also generalizes well to mixed induction and coinduction (Example 3.7). Like Vezzosi [Vez15], however, we use index quantification in lieu of infinite indices (ordinals) to abstract over index variables. We discuss some tradeoffs between finite and infinite interpretations of indices in the semantics in Section §3.3.3.

The remainder of this section discusses the addition of type and process recursion to IRSAX, starting with IRSAX ∞ and then moving to the ω system, with syntax in Figure 3.5.

$$A, B := \ldots \mid X\,[\bar{e}] \text{ where } X\,[\bar{i}] = A\,[\bar{i}]$$
$$P, Q := \ldots \mid f(\bar{i}, \bar{s}, t) \text{ where } f\left(\bar{i}, \overline{x : A}, y : B\right) = P\left(\bar{i}, \bar{x}, y\right)$$

Figure 3.5: IRSAX ∞ and ω Syntax: Recursive Types and Processes

3.3.1 Equirecursive Types

Viewing recursive types as similar to property types, they too are endowed with logical rules that explicitly unfold the left-hand side of a type equation to the right-hand side. As a result, they are *equirecursive* in that they appear to not impose any syntactic footprint at the level of processes. As before, there are two subtyping rules for recursive types that also unfold their definitions.

$$\frac{X\,[\bar{i}] = A\,[\bar{i}] \quad \Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow A\,[\bar{e}])}{\Gamma \vdash \mathtt{write}\, t\, S \div (t \Leftarrow X\,[\bar{e}])}\ \text{recR}$$

$$\frac{X\,[\bar{i}] = A\,[\bar{i}] \quad \Gamma, t \Rightarrow A\,[\bar{e}] \vdash \mathtt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow X\,[\bar{e}] \vdash \mathtt{read}\, t\, S \div \gamma}\ \text{recL}$$

$$\frac{X\,[\bar{i}] = B\,[\bar{i}] \quad \infty\,(\Gamma \vdash A \leq B\,[\bar{e}])}{\Gamma \vdash A \leq X\,[\bar{e}]}\ \leq\text{recR} \qquad \frac{X\,[\bar{i}] = A\,[\bar{i}] \quad \infty\,(\Gamma \vdash A\,[\bar{e}] \leq B)}{\Gamma \vdash X\,[\bar{e}] \leq B}\ \leq\text{recL}$$

To accommodate the infinitude of recursive types, following [DA10], subtyping in IRSAX ∞ is *mixed inductive-coinductive* at the meta level. The ∞ sign surrounding the premises of the subtyping rules above indicates a *coinductive* occurrence of the subtyping

judgment (with all other ones being inductive) [DA09, DA10]. That is, a subtyping derivation is a (potentially) infinitely deep tree where every infinite branch passes through an instance of this rule infinitely many times, representing the unfolding of a recursive type. Meta-level mixed induction and coinduction avoids the technical boilerplate of fixpoint rules, which explicitly manages meta-level coinduction hypotheses using a separate context, that are ordinarily required by recursive subtyping [BH97]. Below, we give a primer on this technique to those who are unfamiliar.

Remark 3.1 (Mixed Induction and Coinduction). Formally, if $F(X, Y)$ is a set operator representing the inference rules for the judgment J with its coinductive and inductive occurrences as X and Y, respectively, then J is generated by $\nu X.\, \mu Y.\, F(X, Y)$—a greatest fixed point surrounding a least fixed point. The proof principle of mixed induction and coinduction then refers to a lexicographic guarded coinduction (to prove/construct coinductive premises) prioritized over a structural induction (to deconstruct inductive premises, because guardedness does not change) [DA09].

In the next subsection, we give an example of an infinite typing derivation to give some intuition about this technique.

3.3.2 Recursive Processes

Following our treatment of subtyping, we also take a mixed inductive-coinductive view of typing recursive definition calls in IRSAX ∞ to avoid fixpoint rules (see [AP13] for an alternate solution using *measured types*). Crucially, the sequent is now tagged with the running value e of the *termination measure* M, a function of the definition called and the

index arguments, which is stepped down as checking continues. This crystallizes our intuition about termination in the presence of sized types: a recursive definition terminates when sizes decrease across each recursive call.

$$\frac{f\left(\bar{i}, \overline{x : A}, y : B\right) = P\left(\bar{i}, \bar{x}, y\right) \quad \bar{i}, \bar{\phi} \vdash e < M(f, \bar{e}') \quad \infty\left(\bar{i}, \bar{\phi}, \overline{s \Rightarrow A} \vdash^{M(f,\bar{e}')} P\left(\bar{e}', \bar{s}, t\right) \div (t \Leftarrow B)\right)}{\bar{i}, \bar{\phi}, \overline{s \Leftarrow A} \vdash^{e} f\left(\bar{e}', \bar{s}, t\right) \div (t \Rightarrow B)}\ \text{call}$$

Perhaps it is surprising that this rule (and the call rule from the previous chapter) need not explicitly mention substitution, weakening, contraction, etc. to make the interior judgment match the exterior judgment of a loop in an infinite proof because they are *admissible*. Because this discussion is quite abstract, let us first state the relevant substitution and weakening principles, then actually write down an infinite typing derivation for a toy process.

Lemma 3.1 (Assumptions about Index Domain / Substitution Principles)**.** *Along with substitution and weakening for both addresses and index data (variables and propositions), we have:*

- *Index substitution: if* $\Gamma \vdash e$*, then:*
  - *if* $\Gamma, i \vdash P \div \gamma$*, then* $[e/i](\Gamma \vdash P \div \gamma)$.
  - *if* $\Gamma, i \vdash A \leq B$*, then* $[e/i](\Gamma \vdash A \leq B)$.
- *Index cut: if* $\Gamma \vdash \phi$*, then:*

– *if* $\Gamma, \phi \vdash P \div \gamma$, *then* $\Gamma \vdash P \div \gamma$.

– *if* $\Gamma, \phi \vdash A \leq B$, *then* $\Gamma \vdash A \leq B$.

Moreover, both cases preserve (sub)typing derivation height.

Proof. By mixed induction and coinduction, assuming that the index domain admits cut and arithmetic substitution. □

Example 3.4 (Infinite Derivations)**.** Recall $\mathtt{nat}[i] = i > 0 \wedge \oplus\{\mathtt{zero} : \mathbf{1}, \mathtt{succ} : \mathtt{nat}[i-1]\}$. The process definition below traverses a unary natural number by induction to produce a unit.

$$\mathtt{eat}(i, x : \mathtt{nat}[i], y : \mathbf{1}) = \mathtt{read}\, x\ \left\{\mathtt{zero} \cdot x' \Rightarrow \mathtt{copy}\, y\, x', \mathtt{succ} \cdot x' \Rightarrow \mathtt{eat}(i-1, x', y)\right\}$$

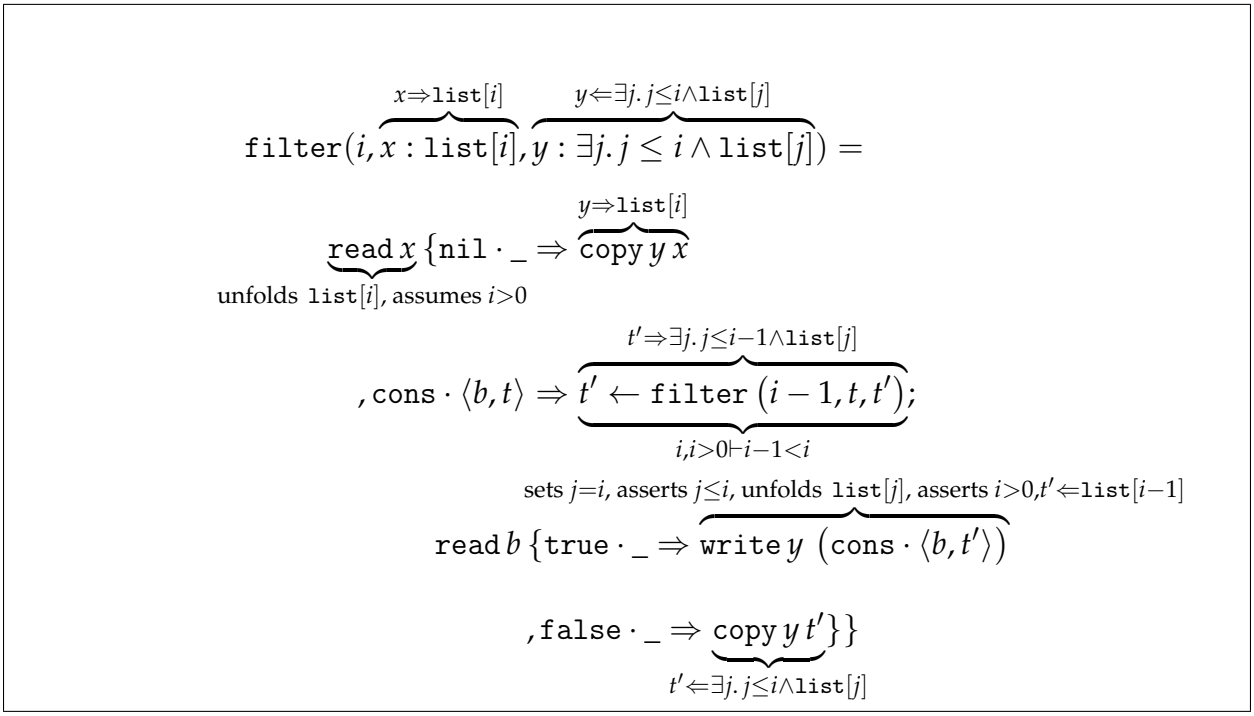
Now, we will write an abbreviated typing derivation for the body of this definition to show how mixed induction and coinduction operates at the meta level. If we take i to be the termination measure, then it suffices to check at the recursive call that $i - 1 < i$. Because that succeeds under the given assumption that $i > 0$, it suffices to reuse the derivation computed so far to complete it by appealing to the meta-level coinduction hypothesis. Arithmetic substitution, address substitution, and weakening of propositions is used to have the judgments at both ends of the loop match.

$$D = \dfrac{\dfrac{}{x' : \mathbf{1} \vdash^i y : \mathbf{1}}\ \text{idR} \quad \dfrac{\dfrac{i, i > 0 \vdash i - 1 < i \quad \infty\,(i, i > 0, x' : \mathtt{nat}[i-1] \vdash^{i-1} y : \mathbf{1})}{i, i > 0, x' : \mathtt{nat}[i-1] \vdash^i y : \mathbf{1}}\ \text{call}}{i, x : \mathtt{nat}[i] \vdash^i y : \mathbf{1}}\ \oplus\text{L}}$$

$$\dfrac{[(i-1)\,/i][x'/x]D,\ \text{weakening } i > 0}{\vdots}$$

The following examples verify that our formulation uniformly covers mutual (co)induction as well as mixed induction and coinduction. We also investigate some interesting properties of quantifier instantiation.

Example 3.5 (List Filter)**.** Let $\mathtt{list}[i] = i > 0 \wedge \oplus\{\mathtt{nil} : \mathbf{1}, \mathtt{cons} : \mathtt{bool} \otimes \mathtt{list}[i-1]\}$ be the type of bitlists of length less than i. The following definition filters false values out of a bitlist by induction where i decreases; because the size-change relationship between the input and output is not exact, we must use an index existential type to abstract over the output size. For convenience, we take the liberty to nest pattern matching and values [PP22] as well as use "_" to indicate a discarded variable. Lastly, we give inline comments detailing some parts of typechecking.



Example 3.6 (Even and Odd Substreams)**.** The definitions below project the even- and odd-indexed substreams y of some input stream x at half of the original depth [Sac14]. By imposing $\texttt{evens} < \texttt{odds}$, this signature terminates by lexicographic induction on i then the definition name; we outline the typechecking and termination checking process in-line. At the meta level, typing derivations for both are mutually mixed inductive-coinductive.

$$\texttt{evens}(i, \overbrace{x : \texttt{str}[2i]}^{x \Rightarrow \texttt{str}[2i]}, \overbrace{y : \texttt{str}[i]}^{y \Leftarrow \texttt{str}[i]}) =$$

$$\underbrace{\texttt{write}\, y}_{\text{unfolds } \texttt{str}[i],\ \text{assumes } i>0} \{ \overbrace{\texttt{head} \cdot h}^{h \Leftarrow \texttt{bool}} \Rightarrow \underbrace{\texttt{read}\, x\ (\texttt{head} \cdot h)}_{\text{unfolds } \texttt{str}[2i],\ \text{asserts } 2i>0}$$

$$, \underbrace{\texttt{tail} \cdot t}_{t \Leftarrow \texttt{str}[i-1]} \Rightarrow t' \leftarrow \underbrace{\texttt{read}\, x\ (\texttt{tail} \cdot t')}_{\ldots t' \Rightarrow \texttt{str}[2i-1]}; \overbrace{\texttt{odds}\ (i-1, t', t)}^{t' \Leftarrow \texttt{str}[2(i-1)+1]} \}$$

$$\texttt{odds}(i, \overbrace{x : \texttt{str}[2i+1]}^{x \Rightarrow \texttt{str}[2i+1]}, \overbrace{y : \texttt{str}[i]}^{y \Leftarrow \texttt{str}[i]}) = t \leftarrow \underbrace{\texttt{read}\, x\ (\texttt{tail} \cdot t)}_{\text{unfolds } \texttt{str}[2i+1],\ \text{asserts } 2i+1>0, t \Rightarrow \texttt{str}[2i+1-1]}; \overbrace{\texttt{evens}\ (i, t, y)}^{\texttt{evens} < \texttt{odds}, t \Leftarrow \texttt{str}[2i], y \Rightarrow \texttt{str}[i]}$$

Thanks to the flexibility of index refinements, we can also lift these definitions to the full coinductive type of streams $\forall k.\ \texttt{str}[k]$ via subsumption, which has the additional effect of hiding the exact size-change relationship between the input and output, which can be considered an implementation detail [Abe14]. The crucial subtyping checks are $i \vdash \forall k.\ \texttt{str}[k] \leq \texttt{str}[2i]$ and $i \vdash \forall k.\ \texttt{str}[k] \leq \texttt{str}[2i+1]$ which instantiate the quantifiers.

$$\texttt{evens}'(i, \overbrace{x : \forall k.\ \texttt{str}[k]}^{x \Rightarrow \forall k.\ \texttt{str}[k]}, \overbrace{y : \texttt{str}[i]}^{y \Leftarrow \texttt{str}[i]}) = \overbrace{\texttt{evens}(i, x, y)}^{x \Rightarrow \texttt{str}[2i]}$$

$$\texttt{odds}'(i, \overbrace{x : \forall k.\ \texttt{str}[k]}^{x \Rightarrow \forall k.\ \texttt{str}[k]}, \overbrace{y : \texttt{str}[i]}^{y \Leftarrow \texttt{str}[i]}) = \overbrace{\texttt{odds}(i, x, y)}^{x \Rightarrow \texttt{str}[2i+1]}$$

Example 3.7 (Projecting Left-Fair Streams)**.** Let us define the mixed inductive-coinductive type $\texttt{lfair}[i, j]$ of *left-fair* bitstreams [BH19]: infinite bitstreams where each element is separated by finitely many timeout labels named "later."

$$\texttt{lfair}[i,j] = \oplus\{\texttt{now} : i > 0 \supset \,\&\,\{\texttt{head} : \texttt{bool}, \texttt{tail} : \exists k.\ \texttt{lfair}[i-1,k]\}$$
$$,\texttt{later} : j > 0 \wedge \texttt{lfair}[i,j-1]\}$$

In particular, i bounds the observation depth of bitstream whereas j bounds the number of timeouts in between consecutive bits. Thus, this type is defined by lexicographic induction on (i,j). First, the provider may offer a bit, in which case the observation depth of the bitstream decreases from i to $i-1$ (in the coinductive branch, indicated by a guarded type). As a result, j may be "reset" as an arbitrary k using an existential. On the other hand, if a timeout is offered, then the depth i does not change. Rather, the number of timeouts j decreases to $j-1$ (in the inductive part, indicated by an asserting type). By using left-fair bitstreams, we can model processes that permit some timeout behavior but are eventually productive, because consecutive bits are interspersed with only finitely many timeouts. Armed with this type, we can define a *projection* operation that removes all of a left-fair bitstream timeouts, returning a plain bitstream.

$$\texttt{proj}(i, j, \overbrace{x : \texttt{lfair}[i,j]}^{x \Rightarrow \texttt{lfair}[i,j]}, \overbrace{y : \texttt{str}[i]}^{y \Rightarrow \texttt{str}[i]}) =$$

$$\overbrace{\texttt{read}\, x}^{\text{unfolds } \texttt{lfair}[i,j]} \{\texttt{now} \cdot x' \Rightarrow$$

$$\underbrace{\texttt{write}\, y}_{\text{unfolds } \texttt{str}[i],\ \text{assumes } i>0} \{\overbrace{\texttt{head} \cdot h}^{h \Leftarrow \texttt{bool}} \Rightarrow \underbrace{\texttt{read}\, x'}_{\text{asserts } i>0} (\texttt{head} \cdot h)\}\}$$

$$, \texttt{tail} \cdot t \Rightarrow \underbrace{t' \leftarrow \texttt{read}\, x' \ (\texttt{tail} \cdot t')}_{t' \Rightarrow \exists k.\ \texttt{lfair}[i-1,k]}; \overbrace{\texttt{proj}\ (i-1, j, t', t)}^{\text{checks } i>0 \vdash (i-1,k) < (i,j)}\}$$

$$, \underbrace{\texttt{later} \cdot x'}_{\text{assumes } j>0} \Rightarrow \underbrace{\texttt{proj}\ (i, j-1, x', y)}_{\text{checks } j>0 \vdash (i,j) < (i,j-1)}\}$$

3.3.3 Typing Summary and Metatheory

In the previous chapter, this section was used to define the non-bidirectional typing of SAX with which we established our operational metatheory. Before we can extend those results to sized type refinements, we note two complications of the current presentation:

- The logical relation defined previously would have to be modified to factor in the presence of arithmetic variables and propositions *despite* the fact that index refinements have no computational content (i.e., are *proof-irrelevant*).

- Because (sub)typing derivations can be infinitely deep, subtyping and process typing soundness would have to be proven using a lexicographic induction that includes some termination measure that steps down when a recursive type / call is unfolded. Thus, one wonders instead if the complexity of that induction can be

diffused by proving termination against the finite derivation prefixes that actually occur at runtime (otherwise, we would not have termination in the first place!).

These two complications are interconnected by a putative solution: if a process typing derivation had *no* free index variables anywhere, then the propositions that *do* appear would be evaluated in an empty index context. In particular, successive instances of the call rule would induce a chain of valuations of the termination measure. Such a chain must be finite, because the natural numbers are well-ordered. As far as the termination proof goes, inspecting the truncation of the typing derivation at the end of this chain is sufficient. Thus, our solution is to extend the non-bidirectional process typing for SAX to IRSAX ∞ as we normally would, but now keeping these observations in mind. The resulting system, IRSAX ω, is presented fully in figure 3.9 and Figure 3.10, which is then used to define IRSAX configuration typing in Figure 3.11. We show representative rules below.

$$\frac{\cdot \vdash \phi \quad \Gamma \vdash_\omega \mathtt{write}\, t\, S \div (t : A)}{\Gamma \vdash_\omega \mathtt{write}\, t\, S \div (t : \phi \wedge A)}\ \omega\wedge\mathrm{R} \qquad \frac{\Gamma, t : A \vdash_\omega \mathtt{read}\, t\, S \div \gamma \text{ if } \cdot \vdash \phi}{\Gamma, t : \phi \wedge A \vdash_\omega \mathtt{read}\, t\, S \div \gamma}\ \omega\wedge\mathrm{L}$$

$$\frac{\Gamma \vdash_\omega \mathtt{write}\, t\, S \div (t \Leftarrow A\,(n))}{\Gamma \vdash_\omega \mathtt{write}\, t\, S \div (t : \exists i.\, A\,(i))}\ \omega\exists\mathrm{R} \qquad \frac{\{\Gamma, t : A\,(n) \vdash_\omega \mathtt{read}\, t\, S \div \gamma\}_{n \in \mathbb{N}}}{\Gamma, t : \exists i.\, A\,(i) \vdash_\omega \mathtt{read}\, t\, S \div \gamma}\ \omega\exists\mathrm{L}$$

$$\frac{\cdot \vdash \phi \quad A \leq_\omega B}{A \leq_\omega \phi \wedge B}\ \omega{\leq}\wedge\mathrm{R} \qquad \frac{A \leq_\omega B \text{ if } \cdot \vdash \phi}{\phi \wedge A \leq_\omega B}\ \omega{\leq}\wedge\mathrm{L}$$

$$\frac{A \leq_\omega B\,(n)}{A \leq_\omega \exists i.\, B\,(i)}\ \omega{\leq}\exists\mathrm{R} \qquad \frac{\{A\,(n) \leq_\omega B\}_{n \in \mathbb{N}}}{\exists i.\, A\,(i) \leq_\omega B}\ \omega{\leq}\exists\mathrm{L}$$

$$\frac{f\,(\bar{i}, \bar{x}, y) = P\,(\bar{i}, \bar{x}, y) \quad \overline{s : A} \vdash P\,(\bar{n}, \bar{s}, t) \div (t : B)}{\overline{s : A} \vdash f\,(\bar{n}, \bar{s}, t) \div (t : B)}\ \omega\mathrm{call}$$

In particular, the logical rules for quantifiers do not introduce any index variables, opt-

ing instead for having infinitely many premises—one for each possible index value. One should think of these index *parameters* as distinct from index *variables*, as a derivation may constructively case on n finitely many times, whereas a variable i must be used *uniformly*. Note that in a classical setting, a derivation would be free to case on n infinitely many times [Yoc89]. As a result, the rules for guarded and asserting types using a "meta-level if" should be seen as having higher-order premises that map closed derivations of a proposition ϕ to one of IRSAX ω (recalling Zeilberger's *higher-order focusing* [Zei08]).

Finally, following our intuition about picking out the finite derivation prefix that actually occurs at runtime, we make the rule for recursive calls inductive. In sum, we have traded infinitely deep derivations for infinitely wide but finitely deep ones. As in Theorem 2.1, we establish soundness of ∞ typing with respect to ω typing but by a lexicographic induction involving termination measures. This proof depends on the following assumptions about the index domain's interaction with ∞ (sub)typing. However, completeness does not hold, because *some* ω-(sub)typing derivations can be non-uniform in the previous sense (by casing on n even finitely many times, etc.). For the result below, we presuppose that any recursive types discussed are *valid* according to the rules in Figure 3.6, which intuitively state that they are defined by induction on some *measure* M as a function of their index arguments (separately from measures on recursive calls). This is also necessary for Definition 3.1 to be well-defined.

Theorem 3.1 (Soundness of IRSAX ∞ (Sub)typing)**.** *Let* Γ *be free of index variables and propositions.*

- *if* $\cdot \vdash A \leq B$, *then* $A \leq_\omega B$

- *if* $\Gamma \vdash^n P \div J$, *then* $|\Gamma| \vdash_\omega |P| \div |J|$

Proof. For the first part, let D be the subtyping derivation; assuming $\cdot \vdash m$ valid A and $\cdot \vdash n$ valid B, the first part proceeds by a lexicographic induction on $((m, n), D)$ where (m, n) and are ordered simultaneously. Thus, either one of m or n must decrease, or they remain the same and D decreases. Here is an example of a base subtyping rule with the bolded quantities decreasing:

- $D = \dfrac{\begin{array}{cc} \vdots\,D_1 & \vdots\,D_2 \\ \cdot \vdash A_1 \leq B_1 & \cdot \vdash A_2 \leq B_2 \end{array}}{\cdot \vdash A_1 \otimes A_2 \leq B_1 \otimes B_2} \leq\otimes \quad \rightsquigarrow$

$$\dfrac{\begin{array}{cc} \mathrm{IH}((m,n), \mathbf{D_1}) & \mathrm{IH}((m,n), \mathbf{D_2}) \\ \vdots & \vdots \\ A_1 \leq_\omega B_1 & A_2 \leq_\omega B_2 \end{array}}{A_1 \otimes A_2 \leq_\omega B_1 \otimes B_2}\ \omega{\leq}\otimes$$

Then, the cases for property types substitute and cut out index variables and propositional assumptions, respectively:

- $D = \dfrac{\begin{array}{c} \vdots\,D' \\ \phi \vdash A \leq B \end{array}}{\cdot \vdash \phi \wedge A \leq B} \leq\wedge\mathrm{L} \rightsquigarrow \dfrac{\begin{array}{cc} \mathrm{IH}((m,n), \mathrm{cut}(E, \mathbf{D}')) & \vdots\,E \\ A \leq B & \text{if } \cdot \vdash \phi \end{array}}{\phi \wedge A \leq B}\ \omega{\leq}\wedge\mathrm{L}$
- $D = \dfrac{\begin{array}{c} \vdots\,D' \\ i \vdash A \leq B \end{array}}{\cdot \vdash \exists i.\, A(i) \leq B} \leq\exists\mathrm{L} \rightsquigarrow \dfrac{\begin{array}{c} \mathrm{IH}((m,n), [k/i]\mathbf{D}') \\ \vdots \\ \{A(k) \leq B\}_{k \in \mathbb{N}} \end{array}}{\exists i.\, A(i) \leq B}\ \omega{\leq}\exists\mathrm{L}$

Lastly, the case for recursive types witnesses m or n decreasing, allowing D to grow:

- $D = \dfrac{X\,[\vec{i}] = A\,[\vec{i}] \quad \begin{array}{c} \vdots\,D' \\ \cdot \vdash A\,[\bar{n}] \leq B \end{array}}{\cdot \vdash X\,[\bar{n}] \leq B} \leq\mathrm{recL} \rightsquigarrow \dfrac{X\,[\vec{i}] = A\,[\vec{i}] \quad \begin{array}{c} \mathrm{IH}((\mathbf{m}', n), D') \\ \vdots \\ \Gamma \vdash A\,[\bar{n}] \leq_\omega B \end{array}}{\Gamma \vdash X\,[\bar{n}] \leq_\omega B}\ \omega{\leq}\mathrm{recL}$

  where $\cdot \vdash m'$ valid $A[\overline{n}]$ and $m' < m$

- $D = \dfrac{\begin{array}{c} D' \\ \vdots \\ X\,[\vec{i}] = B\,[\vec{i}] \quad \cdot \vdash A \leq B\,[\bar{n}] \end{array}}{\cdot \vdash A \leq X\,[\bar{n}]}\,{\leq}\text{recR} \rightsquigarrow \dfrac{\begin{array}{c} \text{IH}((m, \mathbf{n}'), D') \\ \vdots \\ X\,[\vec{i}] = B\,[\vec{i}] \quad \Gamma \vdash A \leq_\omega B\,[\bar{n}] \end{array}}{\Gamma \vdash A \leq_\omega X\,[\bar{n}]}\,\omega{\leq}\text{recR}$

  where $\cdot \vdash n'$ valid $A[\overline{n}]$ and $n' < n$

The second part follows a similar script by lexicographic induction on (n, D)—in particular, n steps down at recursive calls, allowing D to grow once again. □

As before for IRSAX, we summarize (sub)typing rules for IRSAX ∞ in Figure 3.6, Figure 3.7, and Figure 3.8 as well as IRSAX ω in Figure 3.9 and Figure 3.10.

$$\frac{}{\Gamma \vdash m \text{ valid } \mathbf{1}} \quad \frac{\{\Gamma \vdash m \text{ valid } A_\ell\}_{\ell \in S}}{\Gamma \vdash m \text{ valid } \circ \{\ell : A_\ell\}_{\ell \in S}} \circ \in \{\oplus, \&\}$$

$$\frac{\Gamma \vdash m \text{ valid } A \quad \Gamma \vdash m \text{ valid } B}{\Gamma \vdash m \text{ valid } A \circ B} \circ \in \{\otimes, \to\}$$

$$\frac{\Gamma, \phi \vdash m \text{ valid } A}{\Gamma \vdash m \text{ valid } \phi \circ A} \circ \in \{\wedge, \supset\} \quad \frac{\Gamma, i \vdash m \text{ valid } A\,(i)}{\Gamma \vdash m \text{ valid } Qi.\, A\,(i)}\ Q \in \{\forall, \exists\}$$

$$\frac{\Gamma \vdash m > M\,(X, \bar{e}) \quad \infty\,(M\,(X, \bar{e}) \text{ valid } A\,[\bar{e}])}{\Gamma \vdash m \text{ valid } X\,[\bar{e}]}$$

Figure 3.6: IRSAX ∞ Type Validity

$$\frac{X\,[\bar{i}] = A\,[\bar{i}] \quad \Gamma \vdash \texttt{write}\, t\, S \div (t \Leftarrow A\,[\bar{e}])}{\Gamma \vdash \texttt{write}\, t\, S \div (t \Leftarrow X\,[\bar{e}])}\ \text{recR}$$

$$\frac{X\,[\bar{i}] = A\,[\bar{i}] \quad \Gamma, t \Rightarrow A\,[\bar{e}] \vdash \texttt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow X\,[\bar{e}] \vdash \texttt{read}\, t\, S \div \gamma}\ \text{recL}$$

(all rules from Figure 3.3 and above with e added to turnstile)

$$\frac{f\,(\bar{i}, \overline{x : A}, y : B) = P\,(\bar{i}, \bar{x}, y) \quad \bar{i}, \bar{\phi} \vdash e < M(f, \bar{e}') \quad \infty\left(\bar{i}, \bar{\phi}, \overline{s \Rightarrow A} \vdash^{M(f,\bar{e}')} P\,(\bar{e}', \bar{s}, t) \div (t \Leftarrow B)\right)}{\bar{i}, \bar{\phi}, \overline{s \Leftarrow A} \vdash^{e} f\,(\bar{e}', \bar{s}, t) \div (t \Rightarrow B)}\ \text{call}$$

Figure 3.7: IRSAX ∞ Typing

(all rules from Figure 3.4)

$$\frac{X\,[\bar{i}] = B\,[\bar{i}] \quad \infty\,(\Gamma \vdash A \leq B\,[\bar{e}])}{\Gamma \vdash A \leq X\,[\bar{e}]}\ \leq\text{recR} \qquad \frac{X\,[\bar{i}] = A\,[\bar{i}] \quad \infty\,(\Gamma \vdash A\,[\bar{e}] \leq B)}{\Gamma \vdash X\,[\bar{e}] \leq B}\ \leq\text{recL}$$

Figure 3.8: IRSAX ∞ Subtyping

(all rules from Definition 2.1 with subsumption utilizing Figure 3.10)

$$\frac{X\left[\bar{i}\right] = A\left[\bar{i}\right] \quad \Gamma \vdash \texttt{write}\, t\, S \div (t : A\left[\overline{n}\right])}{\Gamma \vdash \texttt{write}\, t\, S \div (t : X\left[\overline{n}\right])}\ \omega\text{recR}$$

$$\frac{X\left[\bar{i}\right] = A\left[\bar{i}\right] \quad \Gamma, t : A\left[\overline{n}\right] \vdash \texttt{read}\, t\, S \div \gamma}{\Gamma, t : X\left[\overline{n}\right] \vdash \texttt{read}\, t\, S \div \gamma}\ \omega\text{recL}$$

$$\frac{\cdot \vdash \phi \quad \Gamma \vdash_\omega \texttt{write}\, t\, S \div (t : A)}{\Gamma \vdash_\omega \texttt{write}\, t\, S \div (t : \phi \wedge A)}\ \omega\wedge\text{R} \qquad \frac{\Gamma, t : A \vdash_\omega \texttt{read}\, t\, S \div \gamma \text{ if } \cdot \vdash \phi}{\Gamma, t : \phi \wedge A \vdash_\omega \texttt{read}\, t\, S \div \gamma}\ \omega\wedge\text{L}$$

$$\frac{\Gamma, \phi \vdash_\omega \texttt{write}\, t\, S \div (t : A) \text{ if } \cdot \vdash \phi}{\Gamma \vdash_\omega \texttt{write}\, t\, S \div (t : \phi \supset A)}\ \omega\supset\text{R} \qquad \frac{\cdot \vdash \phi \quad \Gamma, t : A \vdash_\omega \texttt{read}\, t\, S \div \gamma}{\Gamma, t : \phi \supset A \vdash_\omega \texttt{read}\, t\, S \div \gamma}\ \omega\supset\text{L}$$

$$\frac{\Gamma \vdash_\omega \texttt{write}\, t\, S \div (t \Leftarrow A\,(n))}{\Gamma \vdash_\omega \texttt{write}\, t\, S \div (t : \exists i.\, A\,(i))}\ \omega\exists\text{R} \qquad \frac{\{\Gamma, t : A\,(n) \vdash_\omega \texttt{read}\, t\, S \div \gamma\}_{n \in \mathbb{N}}}{\Gamma, t : \exists i.\, A\,(i) \vdash_\omega \texttt{read}\, t\, S \div \gamma}\ \omega\exists\text{L}$$

$$\frac{\{\Gamma \vdash_\omega \texttt{write}\, t\, S \div (t : A\,(n))\}_{n \in \mathbb{N}}}{\Gamma \vdash_\omega \texttt{write}\, t\, S \div (t : \forall i.\, A\,(i))}\ \omega\forall\text{R} \qquad \frac{\Gamma, t \Rightarrow A\,(n) \vdash_\omega \texttt{read}\, t\, S \div \gamma}{\Gamma, t \Rightarrow \forall i.\, A\,(i) \vdash_\omega \texttt{read}\, t\, S \div \gamma}\ \omega\forall\text{L}$$

$$\frac{f\left(\bar{i}, \overline{x}, y\right) = P\left(\bar{i}, \overline{x}, y\right) \quad \overline{s : A} \vdash P\left(\overline{n}, \overline{s}, t\right) \div (t : B)}{\overline{s : A} \vdash f\left(\overline{n}, \overline{s}, t\right) \div (t : B)}\ \omega\text{call}$$

Figure 3.9: IRSAX ω Typing

(all rules from Figure 2.5)

$$\frac{X\left[\vec{i}\right] = B\left[\vec{i}\right] \quad \Gamma \vdash A \leq_\omega B\left[\bar{n}\right]}{\Gamma \vdash A \leq_\omega X\left[\bar{n}\right]}\ \omega{\leq}\text{recR}$$

$$\frac{X\left[\vec{i}\right] = A\left[\vec{i}\right] \quad \Gamma \vdash A\left[\bar{n}\right] \leq_\omega B}{\Gamma \vdash X\left[\bar{n}\right] \leq_\omega B}\ \omega{\leq}\text{recL}$$

$$\frac{\cdot \vdash \phi \quad A \leq_\omega B}{A \leq_\omega \phi \wedge B}\ \omega{\leq}\wedge\text{R} \qquad \frac{A \leq_\omega B \text{ if } \cdot \vdash \phi}{\phi \wedge A \leq_\omega B}\ \omega{\leq}\wedge\text{L}$$

$$\frac{A \leq B \text{ if } \cdot \vdash \phi}{A \leq_\omega \phi \supset B}\ \omega{\leq}\supset\text{R} \qquad \frac{\cdot \vdash \phi \quad A \leq_\omega B}{\phi \supset A \leq_\omega B}\ \omega{\leq}\supset\text{L}$$

$$\frac{\{A \leq_\omega B\,(n)\}_{n\in\mathbb{N}}}{A \leq_\omega \forall i.\, B\,(i)}\ \omega{\leq}\forall\text{R} \qquad \frac{A\,(n) \leq_\omega B}{\forall i.\, A\,(i) \leq_\omega B}\ \omega{\leq}\forall\text{L}$$

$$\frac{A \leq_\omega B\,(n)}{A \leq_\omega \exists i.\, B\,(i)}\ \omega{\leq}\exists\text{R} \qquad \frac{\{A\,(n) \leq_\omega B\}_{n\in\mathbb{N}}}{\exists i.\, A\,(i) \leq_\omega B}\ \omega{\leq}\exists\text{L}$$

Figure 3.10: IRSAX ω Subtyping

3.4 Termination

Thus, IRSAX configuration typing and reduction in Figure 3.11 and Figure 3.12 now inherit almost directly the termination result from the previous chapter (Lemma 2.6, Lemma 2.7, and Theorem 2.3), because property and recursive types are syntactically silent at the level of processes.

(all rules from Figure 2.7 excluding the proc rule)

$$\frac{\Sigma \leq_\omega \Sigma' \quad \Sigma' \vdash_\omega P \div (a : A') \quad A' \leq_\omega A}{\Sigma \vdash \mathbf{proc}\, P \div (\Sigma, a : A)}\ \text{proc}$$

Figure 3.11: IRSAX Configuration Typing

(all rules from Figure 2.6)

$$\mathbf{proc}\,(f\,(\overline{n},\overline{a},b)) \mapsto P\,(\overline{n},\overline{a},b) \;\text{ where } f\left(\overline{i},\overline{x},y\right) = P\left(\overline{i},\overline{x},y\right)$$

Figure 3.12: IRSAX Configuration Reduction

Definition 3.1 (IRSAX ω Semantic Interpretation)**.** Note that semantic types form a sub-lattice of the (complete) lattice of relations between runtime addresses and terminating configurations. Assuming $\cdot \vdash n$ valid A, we extend Definition 2.5 with the clauses below by lexicographic induction on (n, A). Thus, even though the type argument grows in the first clause, n decreases.

- $[\![X\,[\overline{n}]]\!] \triangleq [\![A\,[\overline{n}]]\!]$ where $X\left[\overline{i}\right] = A\left[\overline{i}\right]$, $\cdot \vdash n'$ valid $A[\overline{n}]$, and $M(X,\overline{n}) < n'$
- $[\![\phi \supset A]\!] \triangleq \begin{cases} [\![A]\!] & \cdot \vdash \phi \\ \top & \text{else} \end{cases}$
- $[\![\phi \wedge A]\!] \triangleq \begin{cases} [\![A]\!] & \cdot \vdash \phi \\ \bot & \text{else} \end{cases}$
- $[\![\forall i.\, A\,(i)]\!] \triangleq \bigcap_{n\in\mathbb{N}} [\![A\,(n)]\!]$ and $[\![\exists i.\, A\,(i)]\!] \triangleq \bigcup_{n\in\mathbb{N}} [\![A\,(n)]\!]$

Lemma 3.2 (IRSAX ω Subtyping Soundness)**.** *If $A \leq_\omega B$ then $A \subseteq B$.*

Proof. By induction on the ω subtyping derivation, inheriting the cases for SAX types from Lemma 2.6. Then, the cases for recursive types are trivial. Finally, the cases for property types are straightforward after establishing the following facts:

- $a \in [\![\phi \wedge A]\!]$ iff $a \in [\![A]\!]$ where $\cdot \vdash \phi$

- $a \in [\![\phi \supset A]\!]$ iff $\cdot \vdash \phi$ implies $a \in [\![A]\!]$
- $a \in [\![\exists i.\, A(i)]\!]$ iff $a \in [\![A(n)]\!]$ for some $n \in \mathbb{N}$
- $a \in [\![\forall i.\, A(i)]\!]$ iff $a \in [\![A(n)]\!]$ for all $n \in \mathbb{N}$

□

Lemma 3.3 (IRSAX ω Typing Soundness)**.** *If* $\Sigma \vdash_\omega P \div (a : A)$*, then* $\Sigma \vDash \mathbf{proc}\, P \div \Sigma, a : A$*.*

Proof. The additional cases for property and recursive types are trivial, because P is the same in the premise and conclusion of each rule. □

Theorem 3.2 (IRSAX ω Fundamental Theorem)**.** *If* $\Sigma \vdash \mathcal{C} \div \Delta$*, then* $[\![\Sigma]\!] \vDash \mathcal{C} \div [\![\Delta]\!]$*.*

Proof. Identical to that of Theorem 2.3. □

3.4.1 Semantics of Recursive Types

Recall at the beginning of the previous section, we promised that we would clarify the semantic status of recursive types, i.e., that they are actually modeled by (co)inductively-defined sets in the semantics. Thus, it suffices to determine whether they are denoted by least/greatest fixed points of the corresponding semantic type constructors. Our use of unbounded index quantification suggests that we reach for *Kleene's fixed point theorem* to make this characterization [LM14], requiring type constructors to be upper/lower semi-continuous. We want to note, however, that like Abel [Abe12], semi-continuity, monotonicity, etc. are *not* relevant to our proof of termination. Now, let $X[i] = i >$

$0 \wedge F(X[i-1])$ and $Y[i] = i > 0 \supset G(Y[i-1])$ where F and G are syntactic type constructors. By induction on n, it is immediate that $[\![X[n]]\!] = [\![F]\!]^n(\bot)$ and $[\![Y[n]]\!] = [\![G]\!]^n(\top)$. This leads us to the following results.

Theorem 3.3 (Kleene's Fixed Point Theorem [CC79])**.** *On a complete lattice, if $\mathscr{F}$ is upper resp. lower semi-continuous, then $\bigcup_{n\in\mathbb{N}} \mathscr{F}^n(\bot)$ resp. $\bigcap_{n\in\mathbb{N}} \mathscr{F}^n(\top)$ is the least resp. greatest fixed point of $\mathscr{F}$.*

Thus, if $[\![F]\!]$ and $[\![G]\!]$ satisfy the conditions above, then $\exists i.\, X[i]$ and $\forall i.\, Y[i]$ are denoted by their least and greatest fixed points, respectively. However, semi-continuity is decidely more restrictive than the usual monotonicity restriction in functional programming [Abe12], so this result explicitly rules out certain type constructors that intuitively inspect "an infinite amount of information" about their type inputs, like those defined by *typecase*.

3.5 Related Work

3.5.1 Index Refinements for Session Types

Session types are related to SAX, as it also has an asynchronous message passing interpretation [PP21]. Thus, our formulation of index refinements is related to [TCP11, GG13, TV19].

3.5.2 Sized Types and Inference

Sized types is related to size-change termination [LJB01] of rewrite systems [TG03, BR09]. Sized (co)inductive types [BFG+04, Bla04, Abe06, AP13] gave way to sized mixed inductive-

coinductive types [Abe12, AP13]. In parallel, linear size arithmetic for sized inductive types [CK01, Xi01, II06] was generalized to support coinductive types as well [Sac14]. We present, to our knowledge, the first sized type system to combine both features from above. As we mentioned earlier, we use unbounded quantification [Vez15] in lieu of transfinite sizes to represent (co)data of arbitrary height and depth. However, the state of the art [AP13] supports polymorphic and higher-kinded types, which is part of the future work.

IRSAX is closely related to the sequential functional language of [LR19], which utilizes circular typing derivations for a sized type system with mixed inductive-coinductive types and implicit quantification. In particular, their well-foundedness criterion on circular proofs corresponds to our validity condition on infinite proofs. However, they encode recursion using a fixed point combinator and utilize transfinite size arithmetic, both of which we intentionally avoid. Moreover, our adherence to a strict bidirectional typing discipline seems to obviate the need for choice operators in yielding mode-correct (sub)typing for quantifiers. Lastly, our metatheory, seems to be both simpler—by translation to ω typing—and more general because it does not have to explicitly rule out non-circular derivations.

Lastly, [Sac14] and [CLBed] consider *size inference*, which translates recursive programs with non-sized (co)inductive types to their sized counterparts when they are well-defined. Since our view is that sized types are a mode of use of more general index refinements, we do not consider size inference.

3.5.3 Sized Types and Termination Checking for π-calculi

The study of termination in the π-calculus was initiated by [YBH01, San06] by syntactically constraining type and/or term structure, which we intentionally avoid. Others assign numeric *levels* to each channel name and restrict communication such that a measure induced by said levels decreases consistently [DS06, DHS10, CH11]. While message passing is a different setting than ours, we are interested in the relationship between sizes and levels, because sized types have already been used to analyze parallel complexity in the π-calculus [BG22].

Severi et al. [SPTD16] give a mixed functional and concurrent programming language where corecursive definitions are typed with the later modality [Nak00]. Since Vezzosi [Vez15] gives an embedding of the later modality and its dual into sized types, we believe that a similar arrangement can be achieved in IRSAX. In any case, IRSAX supports recursion schemes more complex than structural (co)recursion [LM16].

3.5.4 Infinitary Proof Theory

Validity conditions of infinite proofs have been developed to keep cut elimination productive, which correspond to criteria like the guardedness check [BDS16, DP22]. Although we use infinite typing derivations, we explicitly avoid syntactic termination checking for its non-compositionality. Nevertheless, we are interested in implementing such validity conditions as uses of sized types as future work. Relatedly, cyclic termination proofs for separation logic programs can be automated [BBC08, TB20], although it is unclear how they could generalize to concurrent programs (in the setting of concurrent separation

logic) as well as codata.

Chapter 4

Dependent Refinements

> *We take papers on logic from the previous century and republish them in POPL.* —
> Karl Crary

In this chapter, we tackle the second half of our vision for total correctness type refinements by extending the type system for SAX to accommodate *dependent refinements* for partial correctness reasoning.

- In Section §4.1, we modify the SAX typing judgment by attaching *pre-* and *post-condition* assertions to types, which enable us to implement dependent refinements from a Hoare logic point of view. Notably, the bidirectional discipline collapses the design space for typing rules.

- In Section §4.2, we analyze a representative set of typing rules in DRSAX. In particular, we establish a correspondence with bidirectional typing for DRSAX and the rules of Hoare logic. This is precisely where bidirectional typing pays off. First, we reproduce standard path-sensitive reasoning for data types. By once again taking a

mixed inductive-coinductive view of (sub)typing in the presence of type and process recursion, the interaction between typing derivation circularity and subsumption uniformly treats inductive, coinductive, and mixed inductive-coinductive invariants. Lastly, bidirectionality enables a lightweight mechanism for the encapsulation of negatively typed continuations, where refinements may hide information about internal processes.

- In Section §4.3, we show how syntactic type soundness implies *observable partial correctness*, where data of purely positive type satisfy their postconditions directly. As indicated in the introduction, because our operational model is based on futures and not speculations [Har16, Chapter 38], our result is termination-oblivious.

4.1 Judgmental Structure

Following [Che22], we modify the SAX process typing judgment to include *preconditions* and *postconditions* ϕ on addresses as follows. First, let *subset types* $\mathbf{A}, \mathbf{B} := (A \mid \lambda x.\, \phi\,(x))$ where ϕ is an *assertion* from the assertion logic we define in the next section. Intuitively, an address s of subset type is of the underlying type such that $\phi(s)$ also holds [ROS98]. Then, judgments take on the form $s \Rightarrow \mathbf{A}$ and $t \Leftarrow \mathbf{A}$ with full details in Figure 4.1. In particular, the subtyping and process typing judgments allow types and assertions in the antecedents and the succedent to refer to address variables occurring to their left-hand side [de 91].

$$
\begin{aligned}
\text{judgments } J &:= (s \Rightarrow \mathbf{A}) \mid (t \Leftarrow \mathbf{A}) \\
\text{contexts } \Gamma &:= \cdot \mid \Gamma, J \\
\text{generic succedent } \gamma &:= t \Leftarrow \mathbf{A} \\
\text{process typing } \Gamma &\vdash P \div J \\
\text{subtyping } \Gamma &\vdash A \leq B
\end{aligned}
$$

Figure 4.1: DRSAX Judgments

4.2 Syntax and Bidirectional Typing

The full syntax for DRSAX types and processes are given in Figure 4.2: in addition to labelled sums and lazy records, eager pair and function types are made dependent on addresses. In all cases, certain type components are *subset types* that internalize pre-/post-conditions. Before we analyze a representative set of (sub)typing rules, listed in Figure 4.3 and Figure 4.4, we first describe the assertion logic underlying the refinement system.

$A :=$	$A^+ \mid A^- \mid X$ where $X = A$	
$\mathbf{A} :=$	$(A \mid \lambda x.\, \phi\,(x))$	subset type
$A^+ :=$	$\mathbf{1}$	
$\mid$	$\oplus \{\ell : A_\ell\}_{\ell \in S}$	labelled sum
$\mid$	$(x : \mathbf{A}) \otimes B\,(x)$	dependent eager pair
$A^- :=$	$\& \{\ell : \mathbf{A}_\ell\}_{\ell \in S}$	lazy record
	$(x : \mathbf{A}) \to \mathbf{B}\,(x)$	dependent function

Figure 4.2: DRSAX Syntax

4.2.1 The Assertion Logic of Axioms

Assertions, given by the grammar below, are drawn from the (classical) first-order theory of equality with uninterpreted functions.

$$\phi, \psi := \bot \mid \top \mid M \equiv N \mid \mathrm{is}_k\,(M) \mid \phi \wedge \psi \mid \phi \supset \psi \mid \phi \subset \psi \mid \forall x.\, \phi\,(x) \mid \ldots$$

$$M, N := s \mid \underbrace{\langle\rangle}_{\mathbf{1}\mathrm{R}} \mid \underbrace{\langle M, N\rangle}_{\otimes\mathrm{R}} \mid \underbrace{M\,'\,N}_{\to\mathrm{L}} \mid \underbrace{k \cdot M}_{\oplus\mathrm{R}} \mid \underbrace{M.k}_{\&\mathrm{L}}$$

We approximate process dynamics by a careful definition of first-order *terms* M, N. First, the indirection introduced by addresses is collapsed by treating address variables as term variables and runtime addresses as nullary function symbols (*not* constants, because unequal addresses do not necessarily have unequal referents). Thus, an address s pointing to SAX (co)data denoted by M is represented by the assertion $s \equiv M$. Finally, each axiom is assigned an uninterpreted function:

- *Right axioms*: $\langle\rangle$ is a unit value, $\langle M, N\rangle$ is a pair of values M and N, and $k \cdot M$ is a k-tagged value M. These function symbols are additionally subject to the first-order theory of *non-cyclic* data structures [Opp78]. Note that even with recursive types, values cannot be cyclic, because a non-allocating process cannot write to and read from the same address. For convenience, we assume the availability of the assertion $\mathrm{is}_k(M) \iff \exists x.\, M \equiv k \cdot x$.

- *Left axioms*: $M\,'\,N$ represents an application of the SAX function denoted by M to argument N and $M.k$ is the k^{th} projection of the record denoted by M. Note our

use of the phrase "the [continuation] denoted by M"—we do *not* directly encode function bodies into the assertion logic, as that could reveal information hidden by negative type refinements discussed in the next subsection. Instead, a continuation addressed by s is abstractly described by assertions about $s\,'\,N$ or $s.k$—which seems to achieve the effect of *copattern matching* [APTS13] via defunctionalized negative eliminations [Rey72].

4.2.2 Axioms as Assignments

Right axioms flow assertion information from right to left, corresponding to the assignment rule in Hoare logic, where the postcondition becomes the precondition by substitution. Left axioms seem to be dual, instead flowing information from left to right. Below, we show how SAX axioms transition to DRSAX ones via these processes.

$$\frac{}{\Gamma, s \Leftarrow A, t \Leftarrow B \vdash \mathtt{write}\, u\, \langle s, t\rangle \div (u \Leftarrow A \otimes B)}\ \otimes\mathrm{X} \rightsquigarrow$$

$$\frac{}{\Gamma, s \Leftarrow \mathbf{A}, t \Leftarrow B\,(s) \mid \lambda y.\, \phi\,(\langle s, y\rangle) \vdash \mathtt{write}\, u\, \langle s, t\rangle \div (u \Leftarrow (x : \mathbf{A}) \otimes B\,(x) \mid \lambda z.\, \phi\,(z))}\ \otimes\mathrm{X}$$

$$\frac{}{\Gamma, u \Rightarrow A \to B, s \Leftarrow A \vdash \mathtt{read}\, u\, \langle s, t\rangle \div (t \Rightarrow B)}\ \to\mathrm{X} \rightsquigarrow$$

$$\frac{}{\Gamma, u \Rightarrow (x : \mathbf{A}) \to \mathbf{B}\,(x) \mid \lambda z.\, \phi\,(z), s \Leftarrow \mathbf{A} \vdash \mathtt{read}\, u\, \langle s, t\rangle \div (t \Rightarrow \mathbf{B}\,(s))}\ \to\mathrm{X}$$

These axioms explain the particular way in which subset types are nested: a dependent eager pair type must output an assertion for its left component and a function type must output assertions for both input and output types. Axioms for labelled sum and record types follow symmetrically, with only the latter nesting subset types. Lastly, we comment on why we consider *this* particular choice of rules rather than others, especially

because the presence of assertions enlarges the design space. First, bidirectionality forces a certain flow of information, discounting, for example, the following rule, which is seemingly type-sound! Note that A is implicitly a subset type $A \mid \top$.

$$\frac{}{\Gamma, s \Leftarrow \mathbf{A}, t \Leftarrow B\,(s) \vdash \mathtt{write}\, u\, \langle s, t\rangle \div (u : (x : \mathbf{A}) \otimes B\,(x) \mid \lambda z.\, z \equiv \langle s, t\rangle)}\ \otimes\mathrm{X}$$

Dually, one wonders whether left axioms can reveal the identity of the destination in a similar way:

$$\frac{}{\Gamma, u \Rightarrow (x : \mathbf{A}) \to (B\,(x) \mid \lambda y.\, \psi\,(x, y)) \mid \lambda z.\, \phi\,(z)\,, s \Leftarrow \mathbf{A} \vdash \mathtt{read}\, u\, \langle s, t\rangle \div (t \Rightarrow B\,(s) \mid \lambda y.\, \psi\,(s, y) \wedge y \equiv u'\, s)}\ \to\mathrm{X}$$

However, this identity cannot be propagated in the corresponding right rule due to asynchrony: u would no longer be in scope in the premise. Thus, this rule would *not* be type-sound.

4.2.3 Snips as Composition

Viewing the assertion attached to the cut formula as a *midcondition*, snips correspond to the *composition rule* in Hoare logic.

$$\frac{\Gamma \vdash P(x) \div (x \Leftarrow \mathbf{A}) \quad \Gamma, x \Leftarrow \mathbf{A} \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma}\ \mathrm{snipR}$$

$$\frac{\Gamma \vdash P(x) \div (x \Rightarrow \mathbf{A}) \quad \Gamma, x \Rightarrow \mathbf{A} \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma}\ \mathrm{snipL}$$

4.2.4 Subsumption as Consequence

Introducing the auxiliary judgment $\Gamma \vdash \mathbf{A} \leq \mathbf{B}$, $\leq$pred is the standard *predicate subtyping* rule in systems with subset types [ROS98]. Otherwise, $\Gamma \vdash A \leq B$ in Figure 4.4 follows the usual script for subtyping dependent types [AC96]: subtyping premises for the second component of dependent eager pair and function types are fibered over the stronger subtype. Then, subtyping rules for labelled sums and lazy records are preserved from SAX. Because subsumption now utilizes predicate subtyping, the left and right subsumption rules respectively perform the *precondition weakening* and *postcondition strengthening* that is characteristic of the consequence rule in Hoare logic.

$$\frac{\Gamma \vdash \mathbf{A} \leq \mathbf{B} \quad \Gamma \vdash P \div (t \Rightarrow A)}{\Gamma \vdash P \div (t \Leftarrow \mathbf{B})} \leq\text{R} \qquad \frac{\Gamma \vdash \mathbf{A} \leq \mathbf{B} \quad \Gamma, s \Leftarrow \mathbf{B} \vdash P \div \gamma}{\Gamma, s \Rightarrow \mathbf{A} \vdash P \div \gamma} \leq\text{L}$$

4.2.5 Positive Left Rules as Conditionals

Positive left rules are *path-sensitive*: they explicitly add an equation revealing the identity of the scrutinized address. Below, the left rule for eager pairs scrutinizing u includes, in its premise, the equation $\lambda y.\, u \equiv \langle x, y \rangle$. This is reminiscent of the conditional rule in Hoare logic, where the value of the guard is made visible when verifying each branch.

$$\frac{\Gamma, u \Rightarrow A \otimes B, x \Rightarrow A, y \Rightarrow B \vdash P\,(x,y) \div \gamma}{\Gamma, u \Rightarrow A \otimes B \vdash \mathtt{read}\, u\ (\langle x,y\rangle \Rightarrow P\,(x,y)) \div \gamma}\ \otimes\mathrm{L} \rightsquigarrow$$

$$\frac{\Gamma, u \Rightarrow (x : \mathbf{A}) \otimes B\,(x) \mid \lambda z.\, \phi\,(z), x \Rightarrow \mathbf{A}, y \Rightarrow B\,(x) \mid \lambda y.\, u \equiv \langle x,y\rangle \vdash P\,(x,y) \div \gamma}{\Gamma, u \Rightarrow (x : \mathbf{A}) \otimes B\,(x) \mid \lambda z.\, \phi\,(z) \vdash \mathtt{read}\, u\ (\langle x,y\rangle \Rightarrow P\,(x,y)) \div \gamma}\ \otimes\mathrm{L}$$

$$\frac{\left\{\Gamma, t \Rightarrow \oplus\left\{\ell : A_\ell\right\}_{\ell\in S}, x : A_k \vdash P_k\,(x) \div \gamma\right\}_{k\in S}}{\Gamma, t \Rightarrow \oplus\left\{\ell : A_\ell\right\}_{\ell\in S} \vdash \mathtt{read}\, t\ \left\{\ell \cdot x \Rightarrow P_\ell\,(x)\right\}_{\ell\in S} \div \gamma}\ \oplus\mathrm{L} \rightsquigarrow$$

$$\frac{\left\{\Gamma, x \Rightarrow A_k \mid \lambda x.\, \phi\,(k \cdot x), t \Rightarrow \oplus\left\{\ell : A_\ell\right\}_{\ell\in S} \mid \lambda y.\, \phi\,(y) \wedge y \equiv k \cdot x \vdash P_k\,(x) \div \gamma\right\}_{k\in S}}{\Gamma, t \Rightarrow \oplus\left\{\ell : A_\ell\right\}_{\ell\in S} \mid \lambda y.\, \phi\,(y) \vdash \mathtt{read}\, t\ \left\{\ell \cdot x \Rightarrow P_\ell\,(x)\right\}_{\ell\in S} \div \gamma}\ \oplus\mathrm{L}$$

Path sensitivity is necessary to derive the following examples.

Example 4.1 (Eager Swap II)**.** Recall the following process that swaps a (non-dependent) pair addressed by z and writes it to w. Then, the following judgment is derivable, which expresses the action of swapping. In this and all subsequent examples, the judgment $x \Rightarrow A \mid \lambda y.\, \phi\,(y)$ will be abbreviated as $x \Rightarrow A \mid \phi\,(x)$ (and vice versa for $\Leftarrow$), i.e., the antecedent/succedent variable and bound variable of a pre-/post-condition may be conflated.

$$z \Rightarrow A \otimes B \vdash \mathtt{read}\, z \left(\overbrace{\langle x,y\rangle}^{y \Rightarrow B \mid z \equiv \langle x,y\rangle} \Rightarrow \underbrace{\mathtt{write}\, w\, \langle y,x\rangle}_{x \Leftarrow A \mid \phi(z,\langle y,x\rangle)} \right) \div (w \Leftarrow B \otimes A \mid \phi\,(z,w))$$

Here, $\phi\,(z,w) \triangleq \forall x,y.\, z \equiv \langle x,y\rangle \supset w \equiv \langle y,x\rangle$. The critical point of typechecking is when $x \Rightarrow A$ meets $x \Leftarrow A \mid \phi\,(z,\langle y,x\rangle)$ by subsumption, which depends on $y \Rightarrow B \mid z \equiv \langle x,y\rangle$ introduced by path sensitivity.

Example 4.2 (Negation II)**.** Recall that we can define Boolean negation of x, storing the result in y, as:

$$P \triangleq \texttt{read}\, x \left\{ \overbrace{\texttt{true} \cdot x'}^{x \Rightarrow \texttt{bool} \mid x \equiv \texttt{true} \cdot x'} \Rightarrow \underbrace{\texttt{write}\, y\ \left(\texttt{false} \cdot x'\right)}_{x' \Leftarrow \texttt{bool} \mid \phi(x, \texttt{false} \cdot x')}, \overbrace{\texttt{false} \cdot x'}^{x \Rightarrow \texttt{bool} \mid x \equiv \texttt{false} \cdot x'} \Rightarrow \underbrace{\texttt{write}\, y\ \left(\texttt{true} \cdot x'\right)}_{x' \Leftarrow \texttt{bool} \mid \phi(x, \texttt{true} \cdot x')} \right\}$$

Then, the judgment $x \Rightarrow \texttt{bool} \vdash P \div (y \Leftarrow \texttt{bool} \mid \phi(x,y))$ is derivable where $\phi(x,y) \triangleq (\text{is}_{\texttt{true}}(x) \supset \text{is}_{\texttt{false}}(y)) \wedge (\text{is}_{\texttt{false}}(x) \supset \text{is}_{\texttt{true}}(y))$. In the first branch, for example, the critical point of typechecking is when $x' \Rightarrow \texttt{bool}$ meets $x' \Leftarrow \texttt{bool} \mid \phi(x, \texttt{false} \cdot x')$ via subsumption—the assumption that $x \Rightarrow \texttt{bool} \mid x \equiv \texttt{true} \cdot x'$ due to path sensitivity is essential. The second branch follows symmetrically.

4.2.6 Negative Right Rules as Hoare-style Data Abstraction

Negative right rules are perhaps the most exotic part of DRSAX, because they must manually verify the postcondition χ attached to the type due to the lack of right-contraction. To protect the abstraction boundary of the continuation, it is discharged only with the information given by the assertions nested under the type, as opposed to verification against the continuation reified into higher-order logic [RP08]. Note that the ellipses are shorthand for the type of the address t scrutinized. This facility seems to be related to Hoare-style data abstraction [Hoa72], in which information hiding is admitted by *ad hoc* scoping restrictions.

$$\frac{\Gamma, x \Rightarrow A \vdash P\,(x,y) \div (y \Leftarrow B)}{\Gamma \vdash \mathtt{write}\, t\ (\langle x,y\rangle \Rightarrow P\,(x,y)) \div (t \Leftarrow A \to B)}\ {\to}\mathrm{R} \rightsquigarrow$$

$$\frac{\Gamma, x \Rightarrow A \mid \lambda x.\, \phi\,(x) \vdash P\,(x,y) \div (y \Leftarrow B \mid \lambda y\, \psi\,(y)) \quad \Gamma, t \Rightarrow \ldots \mid \forall x.\, \phi\,(x) \supset \psi\,(x, z'\,x) \vdash \chi\,(t)}{\Gamma \vdash \mathtt{write}\, t\ (\langle x,y\rangle \Rightarrow P\,(x,y)) \div (t \Leftarrow ((x : A \mid \lambda x.\, \phi\,(x)) \to B \mid \lambda y.\, \psi\,(x,y)) \mid \lambda z.\, \chi\,(z))}\ {\to}\mathrm{R}$$

$$\frac{\{\Gamma \vdash P_k\,(x) \div (x \Leftarrow A_\ell)\}_{\ell \in S}}{\Gamma \vdash \mathtt{write}\, t\ \{\ell \cdot x \Rightarrow P_\ell\,(x)\}_{\ell \in S} \div \left(t \Leftarrow \&\,\{\ell : A_\ell\}_{\ell \in S}\right)}\ \&\mathrm{R} \rightsquigarrow$$

$$\frac{\{\Gamma \vdash P_k\,(x) \div (x \Leftarrow A_\ell \mid \lambda x.\, \phi_\ell\,(x))\}_{\ell \in S} \quad \Gamma, t \Rightarrow \ldots \mid \lambda y.\ \textstyle\bigwedge_{\ell \in S} \phi_\ell\,(y.\ell) \vdash \psi\,(t)}{\Gamma \vdash \mathtt{write}\, t\ \{\ell \cdot x \Rightarrow P_\ell\,(x)\}_{\ell \in S} \div \left(t \Leftarrow \&\,\{\ell : A_\ell \mid \lambda x.\, \phi_\ell\,(x)\}_{\ell \in S} \mid \lambda y.\, \psi\,(y)\right)}\ \&\mathrm{R}$$

This enables an interesting example below where the nested refinements may engage in some information hiding.

Example 4.3 (Lazy Swap II)**.** Recall the following process that swaps the components of a lazy record.

$$P\,(x,y) \triangleq \mathtt{write}\, w\ \{\mathtt{fst} \cdot x \Rightarrow \mathtt{read}\, z\ (\mathtt{snd} \cdot x), \mathtt{snd} \cdot y \Rightarrow \mathtt{read}\, z\ (\mathtt{fst} \cdot y)\}$$

Curiously, the following judgment is *not* derivable because the initial type ascription to z does not reveal the values for its components.

$$z \Rightarrow \{\mathtt{fst} : A, \mathtt{snd} : B\} \vdash P \div (w \Leftarrow \{\mathtt{fst} : B \mid \lambda y.\, y \equiv z.\mathtt{snd}, \mathtt{snd} : A \mid \lambda x.\, x \equiv z.\mathtt{fst}\})$$

On the other hand, the following judgment is, because the values of the projection are exposed via shared knowledge of new addresses x and y.

$$x \Rightarrow A, y \Rightarrow B, z \Rightarrow \{\mathtt{fst} : A \mid \lambda x'.\, x' \equiv x, \mathtt{snd} : A \mid \lambda y'.\, y' \equiv y\}$$
$$\vdash P \div (w \Leftarrow \{\mathtt{fst} : B \mid \lambda y'.\, y' \equiv y, \mathtt{snd} : A \mid \lambda x'.\, x' \equiv x\})$$

However, utilizing this process would be cumbersome without designating x and y as *ghost variables*, because they are not operationally distinct from the first and second projections of x. We believe that such a facility could be implemented with *proof irrelevance* [LP09] in the future work.

4.2.7 Recursive Processes and Invariants

Lastly, the rule for typing recursive calls is taken directly from the previous chapter *sans* index refinements, which thus admits partial recursion. This generalizes Hoare's (inductive) rule for recursive procedure invocation [Hoa71], which was made coinductive by Bell and Chlipala in the context of *Hoare doubles* [BC16].

$$\frac{f\left(\overline{x : \mathbf{A}}, y : \mathbf{B}\right) = P\left(\overline{x}, y\right) \quad \infty\left(\overline{s \Rightarrow \mathbf{A}} \vdash P\left(\overline{s}, t\right) \div \left(t \Leftarrow \mathbf{B}\right)\right)}{\overline{s \Leftarrow \mathbf{A}} \vdash f\left(\overline{s}, t\right) \div \left(t \Rightarrow \mathbf{B}\right)}\ \text{call}$$

Now, recall that the mandatory change of phase between checking the body of a recursive definition and a recursive call triggers an instance of subsumption on both sides of the sequent. The assertions attached to a type signature then thus be viewed as *recursion invariants* as a generalization of both *loop invariants* and *adaptation* in Hoare logic [Hoa71]. The following examples show that we can uniformly consider inductive, coinductive, and mixed inductive-coinductive invariants with this single rule.

Example 4.4 (Addition)**.** In this example, we define addition on $\texttt{nat} = \oplus\{\texttt{zero} : \mathbf{1}, \texttt{succ} : \texttt{nat}\}$ and verify its correctness against addition axiomatized in our assertion logic as follows: $\forall x, y.\, \texttt{zero} \cdot x + y \equiv y$ and $\forall x, y.\, \texttt{succ} \cdot x + y \equiv \texttt{succ} \cdot (x + y)$.

$$\texttt{add}\,(x : \texttt{nat}, y : \texttt{nat}, z : \texttt{nat} \mid x + y \equiv z) =$$

$$\texttt{read}\, x\, \{\underbrace{\texttt{zero} \cdot x'}_{x \Rightarrow \texttt{nat} \mid x \equiv \texttt{zero} \cdot x'} \Rightarrow \overbrace{\texttt{copy}\, z\, y}^{y \Leftarrow \texttt{nat} \mid x+y \equiv y}, \underbrace{\texttt{succ} \cdot x'}_{x \Rightarrow \texttt{nat} \mid x \equiv \texttt{succ} \cdot x'} \Rightarrow \overbrace{z' \leftarrow \texttt{add}\,(x', y, z')}^{z' \Rightarrow \texttt{nat} \mid z' = x' + y}; \underbrace{\texttt{write}\, z\, (\texttt{succ} \cdot z')}_{z' \Leftarrow \texttt{nat} \mid \texttt{succ} \cdot z' \equiv x + y}\}$$

The base case is trivial due to path sensitivity: copying y to z flows $y \Leftarrow \texttt{nat} \mid x + y \equiv y$, yet the start of the definition flows $y \Rightarrow \texttt{nat}$. But because $x \equiv \texttt{zero} \cdot x'$, we have $(\texttt{nat} \mid \top) \leq (\texttt{nat} \mid x + y \equiv z)$ by the first axiom, resolving the tension by subsumption. Thus, of significance is checking the snip in the succ branch: $\texttt{add}(x', y, z')$ flows $z' \Rightarrow \texttt{nat} \mid z' = x' + y$ (the induction hypothesis from typing circularity) to the right but the write to z flows $z' \Leftarrow \texttt{nat} \mid \texttt{succ} \cdot z' \equiv x + y$ to the left (the induction step). Path sensitivity gives us $x \Rightarrow \texttt{nat} \mid x \equiv \texttt{succ} \cdot x'$, resolving the tension by subsumption.

Example 4.5 (List Filter II)**.** In this example, we establish total correctness of the list filter definition from the previous chapter by verifying that the list returned only contains true elements. Note that in IRSAX this can be done implicitly with *datasort refinement*, i.e., by modifying the type signature as $\texttt{filter}\,(x : \texttt{list}, y : \texttt{truelist})$ where $\texttt{truelist} = \oplus\{\texttt{nil} : \mathbf{1}, \texttt{cons} : (\oplus\{\texttt{true} : \mathbf{1}\}) \otimes \texttt{truelist}\}$. To utilize subset types in DR-SAX, we instead let $\texttt{truelist} = \oplus\{\texttt{nil} : \mathbf{1}, \texttt{cons} : (\texttt{bool} \mid \text{is}_{\texttt{true}}) \otimes \texttt{truelist}\}$, and the definition checks against this signature as-is. The next example details typechecking with

this style of refinement.

Example 4.6 (Stream Filter)**.** Let $\texttt{str} = \&\{\texttt{head} : \texttt{bool}, \texttt{tail} : \texttt{str}\}$ and $\texttt{truestr} = \&\{\texttt{head} : \texttt{bool} \mid \text{is}_{\texttt{true}}, \texttt{tail} : \texttt{truestr}\}$ be bitstreams, the latter restricted to having only true elements. The following partial function retains only the true elements of a bitstream—it is only productive when there are true elements in the input infinitely often. Yet, this definition is considered valid in DRSAX.

$$
\begin{aligned}
&\texttt{filter}\,(x : \texttt{str}, y : \texttt{truestr}) = \\
&\quad h \leftarrow \texttt{read}\, x\ (\texttt{head} \cdot h)\,; \\
&\quad t \leftarrow \texttt{read}\, x\ (\texttt{tail} \cdot t)\,; \\
&\quad\quad t' \leftarrow \texttt{filter}\,(t, t')\,; \\
&\quad\quad\quad \texttt{read}\, h\, \{\overbrace{\texttt{true} \cdot \_}^{h \Rightarrow \texttt{bool} \mid \text{is}_{\texttt{true}}} \Rightarrow \texttt{write}\, y\ \{\texttt{head} \cdot h' \Rightarrow \texttt{copy}\, h'\, h, \texttt{tail} \cdot t'' \Rightarrow \texttt{copy}\, t''\, t'\} \\
&\quad\quad\quad\quad\quad , \texttt{false} \cdot \_ \Rightarrow \texttt{copy}\, y\, t'\}
\end{aligned}
$$

Moreover, checking that it satisfies its type signature is straightforward due to path sensitivity: casing on h flows $h \Rightarrow \texttt{bool} \mid \text{is}_{\texttt{true}}$, which is then flowed to h' via $\texttt{copy}\, h'\, h$, as desired. Taking a step back, we folded the coinductive invariant into the definition of `truestr` [MPV22] as opposed to extending the assertion logic with support for (co)predicates as in [LM14]; in the next example, we use the same trick to encode a mixed inductive-coinductive invariant.

Example 4.7 (Left-Fair Stream Filter)**.** Recall the definition of left-fair bitstreams below;

we additionally define those which only have true elements.

$$\begin{aligned}
\texttt{lfair} &= \oplus\{\texttt{now} : \&\,\{\texttt{head} : \texttt{bool}, \texttt{tail} : \texttt{lfair}\} \\
&\qquad\quad ,\texttt{later} : \texttt{lfair}\} \\
\texttt{tlfair} &= \oplus\{\texttt{now} : \&\,\{\texttt{head} : \texttt{bool} \mid \mathrm{is}_{\texttt{true}}, \texttt{tail} : \texttt{tlfair}\} \\
&\qquad\quad ,\texttt{later} : \texttt{tlfair}\}
\end{aligned}$$

We will now define a process that filters false elements out of a left-fair stream by replacing them with later labels—note that it is also partial, because filtering a left-fair stream with infinitely many false elements will produce infinitely many later labels. The process of typechecking is similar to the previous example, because it is oblivious to the particular scheme of recursion (mixed induction and coinduction).

$$
\begin{array}{l}
\texttt{filter}\,(x : \texttt{lfair}, y : \texttt{tlfair}) = \\
\quad \texttt{read}\,x\,\{\texttt{now}\cdot x' \Rightarrow h \leftarrow \texttt{read}\,x'\,(\texttt{head}\cdot h)\,; t \leftarrow \texttt{read}\,x'\,(\texttt{tail}\cdot t)\,; \\
\qquad t' \leftarrow \texttt{filter}\,(t, t')\,; \\
\qquad \texttt{read}\,h\,\{\overbrace{\texttt{true}\cdot \_}^{h \Rightarrow \texttt{bool}\,|\,\text{is}_{\texttt{true}}} \Rightarrow \\
\qquad\quad s \leftarrow \texttt{write}\,s\,\{\texttt{head}\cdot h' \Rightarrow \texttt{copy}\,h'\,h, \texttt{tail}\cdot t'' \Rightarrow \texttt{copy}\,t''\,t'\}\,; \\
\qquad\quad \texttt{write}\,y\,(\texttt{now}\cdot s) \\
\qquad , \texttt{false}\cdot \_ \Rightarrow \texttt{write}\,y\,(\texttt{later}\cdot t')\} \\
\quad , \texttt{later}\cdot t \Rightarrow t' \leftarrow \texttt{filter}\,(t, t')\,; \texttt{write}\,y\,(\texttt{later}\cdot t')\}
\end{array}
$$

4.2.8 Typing Summary and Metatheory

As in previous chapters, we re-establish soundness and completeness of bidirectional typing to the corresponding non-bidirectional typing for processes.

Definition 4.1 (Non-Bidirectional Typing)**.** As before, let $\mathbf{\Gamma}$ be a context of judgments $\boldsymbol{J} := t : A$. Then, let $\mathbf{\Gamma} \vdash P \div \boldsymbol{J}$ be generated by rules identical to those in Figure 2.4, but with $(\Rightarrow)$ and $(\Leftarrow)$ replaced by $(:)$.

Theorem 4.1 (Soundness and Completeness of Bidirectional Typing)**.** *Let* $|J|$ *turn* $(\Rightarrow)$ *and* $(\Leftarrow)$ *to* $(:)$*. Extending* $|\cdot|$ *to* Γ *in the obvious way:*

- *if* $\Gamma \vdash P \div J$*, then* $|\Gamma| \vdash P \div |J|$

- *if* $\mathbf{\Gamma} \vdash P \div \boldsymbol{J}$, *then* $\Gamma \vdash P' \div J$ *where there exists an extension of the process definition signature such that* P' *unfolds to* P, $|\Gamma| = \mathbf{\Gamma}$, *and* $|J| = \boldsymbol{J}$.

Proof. Both are routine mixed induction and coinductions on the process typing derivation. □

Finally, we tabulate DRSAX (sub)typing rules and correspondences to Hoare logic in Figure 4.3, Figure 4.4, and Figure 4.5.

$$\frac{\Gamma \vdash \mathbf{A} \leq \mathbf{B} \quad \Gamma \vdash P \div (t \Rightarrow A)}{\Gamma \vdash P \div (t \Leftarrow \mathbf{B})} \leq \mathrm{R} \qquad \frac{\Gamma \vdash \mathbf{A} \leq \mathbf{B} \quad \Gamma, s \Leftarrow \mathbf{B} \vdash P \div \gamma}{\Gamma, s \Rightarrow \mathbf{A} \vdash P \div \gamma} \leq \mathrm{L}$$

$$\frac{\Gamma \vdash P(x) \div (x \Leftarrow \mathbf{A}) \quad \Gamma, x \Leftarrow \mathbf{A} \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma} \text{ snipR}$$

$$\frac{\Gamma \vdash P(x) \div (x \Rightarrow \mathbf{A}) \quad \Gamma, x \Rightarrow \mathbf{A} \vdash Q(x) \div \gamma}{\Gamma \vdash x \leftarrow P(x); Q(x) \div \gamma} \text{ snipL}$$

$$\frac{}{\Gamma, s \Leftarrow \mathbf{A} \vdash \mathtt{copy}\, t\, s \div (t \Leftarrow \mathbf{A})} \text{ idR} \qquad \frac{}{\Gamma, s \Rightarrow \mathbf{A} \vdash \mathtt{copy}\, t\, s \div (t \Rightarrow \mathbf{A})} \text{ idL}$$

$$\frac{\Gamma \vdash \phi(\langle\rangle)}{\Gamma \vdash \mathtt{write}\, t\, \langle\rangle \div (t \Leftarrow \mathbf{1} \mid \lambda x.\, \phi(x))} \mathbf{1}\mathrm{X} \qquad \frac{\Gamma, t \Rightarrow \mathbf{1} \mid \lambda x.\, \phi(x) \wedge x \equiv \langle\rangle \vdash P \div \gamma}{\Gamma, t \Rightarrow \mathbf{1} \mid \lambda x.\, \phi(x) \vdash \mathtt{read}\, t\ (\langle\rangle \Rightarrow P) \div \gamma} \mathbf{1}\mathrm{L}$$

$$\frac{}{\Gamma, s \Leftarrow \mathbf{A}, t \Leftarrow B(s) \mid \lambda y.\, \phi(\langle s, y\rangle) \vdash \mathtt{write}\, u\, \langle s, t\rangle \div (u \Leftarrow (x : \mathbf{A}) \otimes B(x) \mid \lambda z.\, \phi(z))} \otimes\mathrm{X}$$

$$\frac{\Gamma, u \Rightarrow (x : \mathbf{A}) \otimes B(x) \mid \lambda z.\, \phi(z), x \Rightarrow \mathbf{A}, y \Rightarrow B(x) \mid \lambda y.\, u \equiv \langle x, y\rangle \vdash P(x, y) \div \gamma}{\Gamma, u \Rightarrow (x : \mathbf{A}) \otimes B(x) \mid \lambda z.\, \phi(z) \vdash \mathtt{read}\, u\ (\langle x, y\rangle \Rightarrow P(x, y)) \div \gamma} \otimes\mathrm{L}$$

$$\frac{\Gamma, x \Rightarrow A \mid \lambda x.\, \phi(x) \vdash P(x, y) \div (y \Leftarrow B \mid \lambda y\, \psi(y)) \quad \Gamma, t \Rightarrow \ldots \mid \forall x.\, \phi(x) \supset \psi(x, z'\, x) \vdash \chi(t)}{\Gamma \vdash \mathtt{write}\, t\ (\langle x, y\rangle \Rightarrow P(x, y)) \div (t \Leftarrow ((x : A \mid \lambda x.\, \phi(x)) \to B \mid \lambda y.\, \psi(x, y)) \mid \lambda z.\, \chi(z))} \to\mathrm{R}$$

$$\frac{}{\Gamma, u \Rightarrow (x : \mathbf{A}) \to \mathbf{B}(x) \mid \lambda z.\, \phi(z), s \Leftarrow \mathbf{A} \vdash \mathtt{read}\, u\, \langle s, t\rangle \div (t \Rightarrow \mathbf{B}(s))} \to\mathrm{X}$$

$$\frac{}{\Gamma, s \Leftarrow A_k \mid \lambda x.\, \phi(k \cdot x) \vdash \mathtt{write}\, t\ (k \cdot s) \div \left(t \Leftarrow \oplus \{\ell : A_\ell\}_{\ell \in S} \mid \lambda y.\, \phi(y)\right)} \oplus\mathrm{X}, k \in S$$

$$\frac{\left\{\Gamma, x \Rightarrow A_k \mid \lambda x.\, \phi(k \cdot x), t \Rightarrow \oplus \{\ell : A_\ell\}_{\ell \in S} \mid \lambda y.\, \phi(y) \wedge y \equiv k \cdot x \vdash P_k(x) \div \gamma\right\}_{k \in S}}{\Gamma, t \Rightarrow \oplus \{\ell : A_\ell\}_{\ell \in S} \mid \lambda y.\, \phi(y) \vdash \mathtt{read}\, t\ \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell \in S} \div \gamma} \oplus\mathrm{L}$$

$$\frac{\{\Gamma \vdash P_k(x) \div (x \Leftarrow A_\ell \mid \lambda x.\, \phi_\ell(x))\}_{\ell \in S} \quad \Gamma, t \Rightarrow \ldots \mid \lambda y.\, \textstyle\bigwedge_{\ell \in S} \phi_\ell(y.\ell) \vdash \psi(t)}{\Gamma \vdash \mathtt{write}\, t\ \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell \in S} \div \left(t \Leftarrow \& \{\ell : A_\ell \mid \lambda x.\, \phi_\ell(x)\}_{\ell \in S} \mid \lambda y.\, \psi(y)\right)} \&\mathrm{R}$$

$$\frac{}{\Gamma, t \Rightarrow \& \{\ell : \mathbf{A}_\ell\}_{\ell \in S} \vdash \mathtt{read}\, t\ (k \cdot s) \div (s \Rightarrow \mathbf{A}_k)} \&\mathrm{X}, k \in S$$

$$\frac{f\left(\overline{x : \mathbf{A}}, y : \mathbf{B}\right) = P(\overline{x}, y) \quad \infty\left(\overline{s \Rightarrow \mathbf{A}} \vdash P(\overline{s}, t) \div (t \Leftarrow \mathbf{B})\right)}{\overline{s \Leftarrow \mathbf{A}} \vdash f(\overline{s}, t) \div (t \Rightarrow \mathbf{B})} \text{ call}$$

Figure 4.3: DRSAX Typing

$$\frac{\Gamma \vdash A \leq B \quad \Gamma, x \Rightarrow A \mid \phi\,(x) \vdash \psi\,(x)}{\Gamma \vdash A \mid \lambda x.\,\phi\,(x) \leq B \mid \lambda x.\,\psi\,(x)} \leq\text{pred}$$

$$\frac{\Gamma \vdash \mathbf{A} \leq \mathbf{A}' \quad \Gamma, x \Rightarrow \mathbf{A} \vdash \mathbf{B}\,(x) \leq \mathbf{B}'\,(x)}{\Gamma \vdash (x : \mathbf{A}') \to \mathbf{B}\,(x) \leq (x : \mathbf{A}) \to \mathbf{B}'\,(x)} \leq\!\to$$

$$\frac{\Gamma \vdash \mathbf{A} \leq \mathbf{A}' \quad \Gamma, x \Rightarrow \mathbf{A} \vdash B\,(x) \leq B'\,(x)}{\Gamma \vdash (x : \mathbf{A}) \otimes B\,(x) \leq (x : \mathbf{A}') \otimes B'\,(x)} \leq\!\otimes$$

$$\frac{S \subseteq T \quad \{\Gamma \vdash A_\ell \leq B_\ell\}_{\ell \in S}}{\Gamma \vdash \oplus\,\{\ell : A_\ell\}_{\ell \in S} \leq \oplus\,\{\ell : B_k\}_{k \in T}} \leq\!\oplus$$

$$\frac{T \subseteq S \quad \{\Gamma \vdash \mathbf{A}_k \leq \mathbf{B}_k\}_{k \in T}}{\Gamma \vdash \&\,\{\ell : \mathbf{A}_\ell\}_{\ell \in S} \leq \&\,\{\ell : \mathbf{B}_k\}_{k \in T}} \leq\!\&$$

$$\frac{X = B \quad \infty\,(\Gamma \vdash A \leq B)}{\Gamma \vdash A \leq X} \leq\text{recR} \qquad \frac{X = A \quad \infty\,(\Gamma \vdash A \leq B)}{\Gamma \vdash X \leq B} \leq\text{recL}$$

$$\frac{}{\Gamma \vdash A \leq A}\ \text{refl} \qquad \frac{\Gamma \vdash A \leq B \quad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C}\ \text{trans}$$

Figure 4.4: DRSAX Subtyping

DRSAX	Hoare Logic
Axiom	Assignment Rule
Cut	Composition
Subsumption	Consequence Rule
Positive Left Rule	Conditional Rule
Negative Right Rule	Data Abstraction
Call Rule	Recursive Procedure Invocation

Figure 4.5: Correspondence between Hoare Logic and DRSAX

4.3 Type Soundness and Observable Partial Correctness

DRSAX configuration typing in Figure 4.6, which modifies the proc rule to utilize (sub)typing like in the previous chapter, almost directly inherits Theorem 2.2 with respect to DRSAX

configuration reduction in Figure 4.7. In particular, a slight modification to Lemma 2.2 is sufficient along with a discussion of path sensitivity in the proof of type preservation. Then, the remaining development integrates our notion of observable partial correctness.

(all rules from Figure 2.7 excluding the proc rule)

$$\frac{\Sigma \leq \Sigma' \quad \Sigma' \vdash P \div (a : A') \quad \Sigma \vdash A' \leq A}{\Sigma \vdash \mathbf{proc}\, P \div (\Sigma, a : A)}\ \text{proc}$$

Figure 4.6: DRSAX Configuration Typing

(all rules from Figure 2.6)

Figure 4.7: DRSAX Configuration Reduction

Lemma 4.1 (DRSAX Configuration Typing Induction)**.** *Lemma 2.2 must be amended with the following additional clause for typing inversion.*

- Typing inversion II: *process typing derivations underneath any instance of the proc rule do not end in a rule instance belonging to a recursive type, because they can be absorbed into the proc rule's subtyping premises using transitivity of subtyping. Thus, if $A \leq B$ for A and B occurring in said process typing derivations, then A and B will still have the same head (non-recursive) type constructor.*

Lemma 4.2 (DRSAX Progress)**.** *If $\cdot \vdash \mathcal{C} \div \Delta$ then either $\mathcal{C}$ is final or $\mathcal{C}$ steps.*

Lemma 4.3 (DRSAX Preservation)**.** *If $\Sigma \vdash \mathcal{C} \div \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$, then $\Sigma \vdash \mathcal{C}' \div \Delta'$ for $\Delta' \supseteq \Delta$.*

Proof. The proof is otherwise identical to Lemma 2.4 except for processes that read values due to the path sensitivity of positive left rules. For example, when $\oplus$R meets $\oplus$L at

address b subject to $\phi(b)$, the k^{th} premise of $\oplus$L requires the type of b to be strengthened with the equality $b \equiv k \cdot a$ where a is somewhere to the left in D. In updating the typing derivation for the process providing b, a would be flowed $\phi(k \cdot a) \wedge k \cdot a \equiv k \cdot a$, which is subsumed by $\phi(k \cdot a)$ via $\leq$L. Thus, the readers of a see the same type ascription as before. To ensure that all readers of b except for the scrutinized instance of $\oplus$L see the same type ascription as before, we inductively update their left subtyping premises noting that $\phi(b) \wedge b \equiv k \cdot a$ implies $\phi(b)$ also using $\leq$L. □

For an alternate proof strategy of type preservation that grapples with this strong form of path sensitivity in a functional setting, see [RP08, Lemma 5]. Now, to prove observable partial correctness, we follow DeYoung et al. [DPP20] and refer to addresses occurring in values as *observable* with all else being *hidden*. As a result, final configurations of *purely positive type*, whose only constituents are value futures, only contain observable addresses.

Lemma 4.4 (*Final Configurations of Purely Positive Type*)**.** Purely *positive refined types* $\mathbf{A}^{++}$ *those that only contain positive type constructors. Extending this definition to* Δ *in the obvious way, if* $\cdot \vdash \mathcal{F} \div \Delta^{++}$*, then* $\mathcal{F}$ *only contains objects of the form* !**future** *a V (whose addresses are observable).*

Proof. By right-to-left induction on the configuration typing derivation, inversion on the process typing derivation for each future reveals a value. □

By taking care of the indirection that observable addresses introduce, we can determine when such a final configuration *satisfies* all of its postconditions.

Lemma 4.5 (DRSAX Observable Satisfaction)**.** *Satisfaction of a final configuration with respect to the postconditions in a context, $\mathcal{F} \vDash \Delta$, is inductively generated as follows.*

$$\frac{}{\cdot \vDash \cdot} \vDash \textit{empty} \quad \frac{\mathcal{F} \vDash \Delta \quad \Delta \vdash \phi\,(V)}{\mathcal{F}, !\mathbf{future}\, a\, V \vDash \Delta, a : A \mid \phi\,(\cdot)} \vDash \textit{value}$$

If $\cdot \vdash \mathcal{F} \div \Delta^{++}$, then $\mathcal{F} \vDash \Delta^{++}$.

Proof. By right-to-left induction on the configuration typing derivation; if $\mathcal{F}$ is empty, then we are done by $\vDash$ empty. Otherwise, by Lemma 4.4, we have $\mathcal{F} = \mathcal{F}_1, !\mathbf{future}\, a\, V$ and $\Delta^{++} = \Delta_1^{++}, a : A^{++} \mid \phi(\cdot)$ where $\mathcal{F}_1 \vDash \Delta_1^{++}$. Thus, it suffices to prove $\Delta_1^{++} \vdash \phi(V)$. By inversion on the process typing derivation for $!\mathbf{future}\, a\, V$, it suffices to only consider right axioms, in which case $\phi(V)$ is already assumed in Γ_1^{++} or is proved directly. For example, $\otimes$R assumes $\phi(\langle b, c\rangle)$ to type $\texttt{write}\, a\, \langle b, c\rangle$, whereas $\mathbf{1}$R has $\Gamma_1^{++} \vdash \phi(\langle\rangle)$ for $\texttt{write}\, a\, \langle\rangle$ in its premise. □

Thus, a well-typed configuration is observably partially correct—it either does not terminate or terminates at a final configuration where all of its purely positive subconfigurations observably satisfy their associated postconditions. We formalize this by the following corollary, which modifies our coinductive notion of configuration safety to include observable satisfaction [Cla77, GM99, MPR18].

Theorem 4.2 (DRSAX Type Soundness and Observable Partial Correctness)**.** *Let $\mathcal{F}$ be Σ-safe iff for all $\Sigma^{++} \subseteq \Sigma$, there exists $\mathcal{F}' \subseteq \mathcal{F}$ such that $\mathcal{F}' \vDash \Sigma^{++}$. Then, let $\mathcal{C}$ be Σ-safe iff, coinductively, $\mathcal{C} \mapsto \mathcal{C}'$ and $\mathcal{C}'$ is Σ-safe. Thus, if $\cdot \vdash \mathcal{C} \div \Sigma$, then $\mathcal{C}$ is Σ-safe.*

We finish by commenting on the generality of our partial correctness result—because

hidden addresses can be made observable by projecting or applying the continuations that hide them, we do not lose power by restricting our attention to observability.

4.4 Related Work

We view DRSAX on a spectrum between languages that *model* concurrency and/or parallelism without native support for them at one end and process calculi with dependent (session) types of varying expressivity at the other. Before we elaborate on this dichotomy, we note that our treatment of codata seems to be related to logical approaches to object encapsulation in the presence of mutable state [Hoa72, Hoa02, OYR09]. Moreover, refer to [BG16, BH19] for reasoning about terminating mixed inductive-coinductive programs.

4.4.1 Language-Based Verification, Concurrency, and Parallelism

Projects like SteelCore [SRF+20] and FCSL [NLSD14] implement a variation of *concurrent separation logic* [O'H04, JKJ+ed] in a metalanguage—in these cases, F* or Coq—in which various shared memory and message-passing constructs can be modeled. Similar efforts that do not use separation logic include that in Dafny [Lei17] and Why3 [SMV15]. Our interest is "one level up"—determining a core language that could, in theory, be embedded in the metalanguages mentioned, intersecting with our discussion of embedded session types below. One exception to this thread (pun intended) is Liquid Effects [KRBJ12], in which dependent refinements are retrofitted directly onto a parallel dialect of C.

4.4.2 Dependent and Embedded Session Types

Toninho et al. [TCP11] initiated the line of work on dependent session types by presenting a session-typed process calculus in Curry-Howard correspondence with first-order intuitionistic linear logic over a domain of non-linear proof terms. In particular, proof terms are not allowed to refer to the channels with which processes communicate in the linear layer. In their retrospective article ten years later [TCP21], they note that most subsequent developments [TY17, TV19, DP20a] have similar restrictions precisely because non-linear dependence on linear objects is problematic. As one exception to the rule, Toninho and Yoshida [TY18] allow proof terms to depend on quoted processes by way of a *contextual monad* [TCP13] related to the adjoint dependent modalities of dependent linear/non-linear logic [KPB15]. The relaxation of the restriction on type dependency comes at the cost of process/term-level duplication, because functional terms can be embedded faithfully into processes—DRSAX need not make this distinction.

Another line of work seeks to embed session type systems into existing dependent type theories, allowing meta-level reasoning about processes and the exploitation of existing language infrastructure [BH10, WX17, dBV19, SYB19, HBK20, MO22]. Embedded implementation is certainly not opposed by DRSAX nor the line of work above, but moving the burden of proof to the meta level requires explicit reasoning about the typing and operational semantics of programs to an extent determined by the embedding depth.

4.4.3 Dependent Call-by-Push-Value

The distinction between positive and negative types in (DR)SAX suggests a comparison to systems based on *dependent call-by-push-value* [PT19], recalling that we are restricted to first-order type dependency (that is, types are not communicable (co)data). In particular, Niu et al.'s cost-aware logical framework [NSGH22] enforces a generalized phase distinction [SH21] between the intensional/cost and extensional/behavioral aspects of a program, the former of which is the core value proposition and technical complication of parallelism. Lastly, sized type refinements seem to be related to their use of recurrences as termination metrics for non-structurally inductive functions, although we are also able to handle coinduction and mixed induction and coinduction.

Chapter 5

Modelling Asynchronous Reactive Programming

Are you working on Artificial Intelligence? — My Second Aunt

In this chapter, we "put it all together" by using IRSAX and DRSAX to model *asynchronous reactive programming* and, consequently, stretch both type refinement systems to their expressive limits. *Reactive programming* is a paradigm where a certain class of values admits time dependence on other such values. For example, if $a = b + c$ and the value of b is updated after the initialization of a, then a changes accordingly. In the *synchronous* formulation of *functional reactive programming*, such values are modeled as infinite streams called *signals*. Thus, $a = b + c$ defines a signal where subsequent *samples* (elements) of a evolve in lockstep with those of b and c. However, this forces all signals to recompute according to the fastest changing one, which is inefficient in some natural applications. For example, a bit-valued signal corresponding to whether a GUI element is clicked ought

to only provide samples when the element's state physically changes. As a result, various authors [GSK21, BM23] have introduced *asynchronous* formalisms based on *modal type theory* as follows:

- Signals of type A are simply streams defined via *modal possibility* $\diamond A$ indicating a value with the potential of asynchronous delay
- There is a synchronization primitive "select" that, given two values of modal type, returns the one that is populated first

Because all types in SAX classify futures—objects that arrive with the potential of asynchronous delay—the type of signals *seems* to coincide with our usual type of streams provided that, at runtime, processes are scheduled in a way that signal samples are computed in temporal (sequential) order [CFPP14]. Because our operational metatheory applies to *all* configuration schedules, this detail is inessential to typability. However, there is a catch: we cannot implement "select," because all reads in SAX are blocking. We can instead opt to model *signal combinators*—many-to-one signal functions—that utilize this form of synchronization without directly adding such a construct to SAX. Our view is that selection as a primitive deserves a deeper treatment via *temporal refinements*. Thus, this chapter is structured as follows:

- In Section §5.1, we define a mixed inductive-coinductive *representation* of signal combinators with selection-based synchronization. We compare its expressive power, as a model, to the calculi of [GSK21, BM23].
- In Section §5.2, we use IRSAX to verify *causality* (a refinement of termination relevant to this domain) for a *zipping* signal combinator.

- In Section §5.3, we use DRSAX to verify that zipping preserves pointwise specifications along the input signals in the output signal, like monotonicity.

In sum, we verify total correctness by the decomposition advocated in the introduction of this dissertation both at the conceptual and linguistic levels. At the end of the chapter, we identify strengths and weaknesses of our approach, leaving more domain-specific properties (e.g., space-leak freedom [GSK21, BM23]) to future work.

5.1 Signal Processors

In this section, we define and work with the type of *signal processors*, our model and representation of signal combinators.

Definition 5.1 (Signal Processors)**.** Let $A \,\&\, B \triangleq \&\,\{\texttt{force} : A \otimes B\}$; we will use this rather odd encoding of conjunction to define dependent records in the last section. The mixed inductive-coinductive type of *signal processors* representing S-ary signal-to-signal functions is defined as follows.

$$\texttt{sp} = \oplus\,\{\texttt{get} : \oplus\,\{\ell : A_\ell\}_{\ell \in S} \to \texttt{sp}, \texttt{put} : C \,\&\, \texttt{sp}\}$$

Given an S-indexed set of signals, a signal processor consists of an infinite series of `put` actions (the coinductive part) corresponding to the output signal with finitely many `get` actions (the inductive part) between consecutive `puts` corresponding to selection-based synchronization among the input signals. In particular, an event of type A_ℓ is the one received first temporally out of all of the input signals. This type is identical to that of

stream processors due to Ghani et al. [GHP09], except that `get` has an asynchronous reading due to the sum type's occurrence.

We will gauge the expressive power of this representation by modeling some common signal combinators that utilize the power of selection-based synchronization. First, let us define some syntactic sugar.

Definition 5.2 (Notation)**.** Below, we define some convenient object-oriented/monadic notation for writing signal processors.

$$(\langle x, y\rangle \leftarrow z.\mathtt{get}; P\,(x, y)) \triangleq z' \leftarrow \mathtt{write}\, z'\ (\langle x, y\rangle \Rightarrow P\,(x, y))\,;\mathtt{write}\, z\ \left(\mathtt{get} \cdot z'\right)$$

$$(x \leftarrow z.\mathtt{put}\,(s)\,; P\,(x)) \triangleq z' \leftarrow \mathtt{write}\, z'\ \{\mathtt{force} \cdot y \Rightarrow x \leftarrow P\,(x)\,;\mathtt{write}\, y\,\langle s, x\rangle\}\,;$$

$$\mathtt{write}\, z\ \left(\mathtt{put} \cdot z'\right)$$

From now on, the sum type of input events we will consider is $A \oplus B \triangleq \oplus\{\mathtt{l} : A, \mathtt{r} : B\}$ for some fixed A and B. For the sake of argument, let $Sig(A)$ be the type of signals with events of type A in an asynchronous functional setting; it is common to implement the interleaving of two signals as a function of type $Sig(A) \to Sig(B) \to Sig(A \oplus B)$. In our type of signal processors, input events *already* arrive interleaved in this fashion, obviating the need for such a combinator. However, the following combinator is not supported under the mixed inductive-coinductive interpretation of said type.

Definition 5.3 (Failure of Dynamic Switching)**.** The following mutually recursive set of definitions forms a signal processor on input events of type $A \oplus B$ and output events of

type $C \triangleq A \oplus B$ such that events of type A are outputted until an event of type B is encountered; then, *only* events of type B are outputted.

$$\texttt{sw}(z : \texttt{sp}) = \langle x, y\rangle \leftarrow z.\texttt{get}; y' \leftarrow y.\texttt{put}\,(x)\,;\texttt{read}\,x\ \{\texttt{l}\cdot\_ \Rightarrow \texttt{sw}\,(y')\,,\texttt{r}\cdot\_ \Rightarrow \texttt{right}\,(y')\}$$

$$\texttt{right}(z : \texttt{sp}) = \langle x, y\rangle \leftarrow z.\texttt{get};\texttt{read}\,x\ \{\texttt{l}\cdot\_ \Rightarrow \texttt{right}(y),\texttt{r}\cdot x' \Rightarrow y' \leftarrow y.\texttt{put}\,(x)\,;\texttt{right}(y')\}$$

It is not valid under the mixed inductive-coinductive interpretation of `sp` because the second definition could wait forever to receive an event tagged `r` (and therefore issue an infinite number of gets) despite belonging to the inductive part of `sp`. Adding a variant of `get` for every nonempty subset of input signals would avoid this *busy-waiting* behavior, but given the difficulty of refining such a type as the number of input signals grows, one would be better off just implementing selection into SAX directly.

On the other hand, our representation is able to encode *zipping*: combining events along the input signals into pairs of events along the output.

Definition 5.4 (Signal Zipping)**.** Starting with events $x_0 : A$ and $y_0 : B$ and taking output events to be of type $C \triangleq A \otimes B$, the following process definition models the zipping signal combinator.

$$\texttt{zip}(x_0 : A, y_0 : B, z : \texttt{sp}) = \langle p, z' \rangle \leftarrow z.\texttt{get};$$

$$\texttt{read}\, p \,\{\texttt{l} \cdot x \Rightarrow q \leftarrow \texttt{write}\, q\, \langle x, y_0 \rangle; w \leftarrow z'.\texttt{put}\, (q)\,; \texttt{zip}\, (x, y_0, w)$$

$$\texttt{r} \cdot y \Rightarrow q \leftarrow \texttt{write}\, q\, \langle x_0, y \rangle; w \leftarrow z'.\texttt{put}\, (q)\,; \texttt{zip}\, (x_0, y, w)\}$$

Thus, our representation is strictly more expressive than that of [GSK21], which cannot implement zipping, but less expressive than that of [BM23], which can implement both zipping and dynamic switching. We are now ready to initiate the process of verification, starting with *causality*.

5.2 Sized Type Refinements for Causality

A productive/terminating function f on streams is *causal* iff the i^{th} output element of f only depends on at most the first i input elements. For example, mapping an element-wise function to an entire stream is causal (the i^{th} output element depends only on the i^{th} input element), whereas the tail projection of a stream is *not* (the i^{th} output element depends on the $i + 1^{\text{th}}$ input element). Because we view events along a signal as occurring in temporal order, verifying causality on top of termination is necessary to ensure that signals are physically realistic and not "looking into the future." Below, we given an index refinement of signal processors that classifies only representations of causal signal combinators.

Definition 5.5 (Causal Signal Processors)**.** The above notion of causality is equivalent to

requiring the i^{th} `put` issued by signal processor to succeed no more than i `gets`. Thus, we can refine the type of signal processors into $\mathtt{sp}[i,j,k]$ where i is the (coinductive) depth of the output signal (number of `puts`), j bounds the number of `gets` between consecutive `puts` (inductive part, reset on each `put`), and k is the difference between the number of `puts` from `gets` seen so far (letting k take on integer values). Thus, on top of the usual sized type refinements—a lexicographic induction on (i,j)—we only need to additionally assert that $k \geq -1$ when a `put` is issued to verify causality.

$$\mathtt{sp}[i,j,k] = \oplus\{\mathtt{put} : i > 0 \supset \left(C \,\&\, \exists j'.\, k \geq -1 \wedge \mathtt{sp}[i-1,j',k+1]\right),$$
$$\mathtt{get} : j > 0 \wedge \left(\oplus\,\{\ell : A_\ell\}_{\ell \in S} \to \mathtt{sp}[i,j-1,k-1]\right)\}$$

Example 5.1 (Causality of Signal Zipping)**.** Because signal zipping alternates `get` and `put` in lockstep, we ascribe it the refined type $\mathtt{zip}\,(i, x_0 : A, y_0 : B, z : \mathtt{sp}[i,1,0])$. That is, we have arbitrarily many `puts` indicated by i with exactly one `get` between consecutive `puts`. Let us review the original source code under this type ascription.

$$\mathtt{zip}(i, x_0 : A, y_0 : B, z : \mathtt{sp}[i,1,0]) = \overbrace{\langle p, z'\rangle \leftarrow z.\mathtt{get};}^{\text{unfolds } \mathtt{sp}[i,1,0],\ \text{asserts } (j=1)>0, z' \Rightarrow \mathtt{sp}[i,0,-1]}$$
$$\mathtt{read}\, p\, \{\mathtt{l} \cdot x \Rightarrow q \leftarrow \mathtt{write}\, q\, \langle x, y_0\rangle; \underbrace{w \leftarrow z'.\mathtt{put}\,(q)}_{\text{unfolds } \mathtt{sp}[i,0,-1],\ \text{assumes } i>0,\ \text{sets } j'=1,\ \text{asserts } (k=-1)\geq -1}; \overbrace{\mathtt{zip}\,(i-1, x, y_0, w)}^{w \Leftarrow \mathtt{sp}[i-1,1,0],\ \text{checks } i>0 \vdash i-1<i}$$
$$\mathtt{r} \cdot y \Rightarrow q \leftarrow \mathtt{write}\, q\, \langle x_0, y\rangle; \underbrace{w \leftarrow z'.\mathtt{put}\,(q)}_{\text{identical to above}}; \overbrace{\mathtt{zip}\,(i-1, x_0, y, w)}^{\text{identical to above}}\}$$

We proceed by induction on i. The `get` issued to z produces $z' \Rightarrow \mathtt{sp}[i, 0, -1]$: both the `get` height j and the causality indicator k are decremented to account for the `get`. Then, the `put` issued to z' produces $w \Rightarrow \mathtt{sp}[i-1, 1, 0]$: the `put` depth i is decremented, the `get` height j is reset (due to the lexicographic induction on (i, j) at the level of types), and the indicator k is incremented to account for the `put`. Thus, the recursive call is able to proceed at $i - 1 < i$ and at the same type $\mathtt{sp}[i, 1, 0]$. In short, signal zipping typechecks and is therefore causal.

5.3 Verifying Recursive Refinements

Switching gears, an interesting class of partial correctness properties on signal processors operate pointwise: given some invariant on consecutive events along the input signal, we expect the represented combinator to preserve this invariant over consecutive events along the output signal. The idea of encoding these invariants via *recursive refinements* is due to [VRJ13]. Below, we define a refinement of the signal processor type along these lines and then verify that signal zipping preserves monotonicity given some ordering on events.

Definition 5.6 (Recursively-Refined Signal Processors)**.** Let $(x : \mathbf{A}) \,\&\, B \triangleq \,\&\, \{\mathtt{force} : (x : \mathbf{A}) \otimes B\}$. Imagining a straightforward extension of DRSAX with indexed types, the type of stream processors is refined according to some $\phi(p, p')$ applying to consecutive events along the input and $\psi(q, q')$ along the output by a bit of technical complication: starting with "plain" `sp`, we define further types $\mathtt{sp}_1$ and $\mathtt{sp}_2$ depending on what combination of two consecutive commands (each either a "get" or "put") were issued until a running input

and output event can be maintained ($\mathtt{sp}_3$).

$$\mathtt{sp} = \oplus\left\{\mathtt{get} : \left(p : \oplus\left\{\ell : A_\ell\right\}_{\ell\in S}\right) \to \mathtt{sp}_1[p], \mathtt{put} : (q : C)\ \&\ \mathtt{sp}_2[q]\right\}$$

$$\mathtt{sp}_1[p : \oplus\left\{\ell : A_\ell\right\}_{\ell\in S}] = \oplus\{\mathtt{get} : \left(p' : \oplus\left\{\ell : A_\ell\right\}_{\ell\in S} \mid \phi\left(p, p'\right)\right) \to \mathtt{sp}_1[p']$$

$$, \mathtt{put} : (q : C)\ \&\ \mathtt{sp}_3[p, q]\}$$

$$\mathtt{sp}_2[q : C] = \oplus\{\mathtt{get} : \left(p : \oplus\left\{\ell : A_\ell\right\}_{\ell\in S}\right) \to \mathtt{sp}_3[p, q]$$

$$, \mathtt{put} : \left(q' : C \mid \psi\left(q, q'\right)\right)\ \&\ \mathtt{sp}_2[q']\}$$

$$\mathtt{sp}_3[p : \oplus\left\{\ell : A_\ell\right\}_{\ell\in S}, q : C] = \oplus\{\mathtt{get} : \left(p' : \oplus\left\{\ell : A_\ell\right\}_{\ell\in S} \mid \phi\left(p, p'\right)\right) \to \mathtt{sp}_3[p', q]$$

$$, \mathtt{put} : \left(q' : C \mid \psi\left(q, q'\right)\right)\ \&\ \mathtt{sp}_3[p, q']\}$$

All types above are simply-typed equivalent to the original type ($\mathtt{sp}$) so as to not affect typability of our previously defined combinator(s).

Finally, we establish our desired property of zipping below.

Example 5.2 (Monotonicity of Signal Zipping)**.** Let $\phi(p, p') \triangleq p \leq p'$ where $\leq$ is the coproduct preorder defined on elements of $A \oplus B$ generated by two separate orders on elements of A and B. That is, we have $\forall x, x'.\, x \leq x' \subset\supset \mathtt{l} \cdot x \leq \mathtt{l} \cdot x$, $\forall y, y'.\, y \leq y' \subset\supset \mathtt{r} \cdot y \leq \mathtt{r} \cdot y'$, and $\forall x, y.\, \neg\left(\mathtt{l} \cdot x \leq \mathtt{r} \cdot y\right) \wedge \neg\left(\mathtt{r} \cdot y \leq \mathtt{l} \cdot x\right)$. Then, let $\psi\left(q, q'\right) \triangleq q \leq q'$ be the usual product/pairwise order on pairs of events of type $A \otimes B$. We aim to verify that $\mathrm{zip}(x_0 : A, y_0 : B, z : \mathtt{sp})$ is well-typed. After unfolding the definition once, each recursive call must check $w \Leftarrow \mathtt{sp}_3[p, q]$ where p is the event received from the input signals and q is the pair of events written along the output. Let us examine the original source code

under this type ascription.

$$\texttt{zip}(x_0 : A, y_0 : B, z : \texttt{sp}) = \overbrace{\langle p, z' \rangle \leftarrow z.\texttt{get};}^{\text{unfolds } \texttt{sp}, z' \Rightarrow \texttt{sp}_1[p]}$$

$$\texttt{read}\, p \,\{\underbrace{\texttt{l} \cdot x}_{p \Rightarrow A \oplus B \mid p \equiv \texttt{l} \cdot x} \Rightarrow q \leftarrow \texttt{write}\, q\, \langle x, y_0 \rangle; \underbrace{w \leftarrow z'.\texttt{put}\,(q)}_{w \Rightarrow \texttt{sp}_3[p,q]}; \texttt{zip}\,(x, y_0, w)$$

$$\underbrace{\texttt{r} \cdot y}_{p \Rightarrow A \oplus B \mid p \equiv \texttt{r} \cdot y} \Rightarrow q \leftarrow \texttt{write}\, q\, \langle x_0, y \rangle; \underbrace{w \leftarrow z'.\texttt{put}\,(q)}_{\text{identical to above}}; \texttt{zip}\,(x_0, y, w)\}$$

Checking each recursive call requires another unfolding of the definition of `zip`, because the type of w in either case must be $\texttt{sp}_3[p, q]$. Moreover, the type of y_0 and y must be specialized to propagate the equalities $q \equiv \langle x, y_0 \rangle$ and $q \equiv \langle x_0, y \rangle$. This presents a slight but not insurmountable technical issue: because `zip` takes on a different type signature at these calls, we would have to factor out them out into a separate definition $\texttt{zip}'$ analogous to $\texttt{evens}'$ and $\texttt{odds}'$ in Example 3.6. Because we are already quite deep into analyzing this definition, we will elide this formality. Instead, let us look at the unfolding of $\texttt{zip}\,(x, y_0, w)$ under the new type ascription with address variables p and q in scope; the second call follows symmetrically. Note that the listing below is not a definition (say, of $\texttt{zip}'$), but a definitional equality.

$$\texttt{zip}(x : A, \overbrace{y_0 : B \mid q \equiv \langle x, y_0\rangle}^{y_0 \Rightarrow B \mid q \equiv \langle x, y_0\rangle}, w : \texttt{sp}_3[p,q]) = \overbrace{\langle p', w'\rangle \leftarrow w.\texttt{get};}^{\text{unfolds } \texttt{sp}_3[p,q], p' \Rightarrow A \oplus B \mid p \leq p', w' \Rightarrow \texttt{sp}_3[p',q]}$$

$$\texttt{read}\, p' \{ \quad \underbrace{\texttt{l} \cdot x'}_{x' \Rightarrow A \mid p \leq \texttt{l} \cdot x'} \Rightarrow q' \leftarrow \underbrace{\texttt{write}\, q'\, \langle x', y_0\rangle}_{y_0 \Leftarrow B \mid q \leq \langle x', y_0\rangle}; \underbrace{w'' \leftarrow w'.\texttt{put}\, (q')}_{q' \Leftarrow C \mid q \leq q', w'' \Rightarrow \texttt{sp}_3[p',q']}; \texttt{zip}\, (x', y_0, w'')$$

$$\underbrace{\texttt{r} \cdot y'}_{y' \Rightarrow B \mid p \leq \texttt{r} \cdot y'} \Rightarrow q' \leftarrow \underbrace{\texttt{write}\, q'\, \langle x_0, y'\rangle}_{y' \Leftarrow B \mid q \leq \langle x_0, y'\rangle}; \underbrace{w'' \leftarrow w'.\texttt{put}\, (q')}_{\text{identical to above}}; \texttt{zip}\, (x_0, y', w'') \}$$

To complete the typechecking process, we must close the derivational loops between $\texttt{zip}\, (x : A, y_0 : B \mid q \equiv \langle x, y_0\rangle, w : \texttt{sp}[p,q])$ and its children $\texttt{zip}(x' : A, y_0 : B \mid q' \equiv \langle x', y_0\rangle, w'' : \texttt{sp}_3[p',q'])$ and $\texttt{zip}\, (x_0 : A, y' : B \mid q' \equiv \langle x_0, y'\rangle, w'' : \texttt{sp}[p',q'])$. This hinges on resolving the tension between type ascriptions for y_0 and y'. In particular, we discharge the following subtyping obligations, which have the force of establishing monotonicity.

- For y_0: $p \Rightarrow A \oplus B \mid p \equiv \texttt{l} \cdot x, x' \Rightarrow A \mid p \leq \texttt{l} \cdot x' \vdash (B \mid q \equiv \langle x, y_0\rangle) \leq (B \mid q \leq \langle x', y_0\rangle)$ due to $p \equiv \texttt{l} \cdot x, p \leq \texttt{l} \cdot x', q \equiv \langle x, y_0\rangle \vdash q \leq \langle x', y_0\rangle$ in the assertion logic, which is straightforward
- For y': $p \Rightarrow A \oplus B \mid p \equiv \texttt{l} \cdot x, y_0 \Rightarrow B \mid q \equiv \langle x, y_0\rangle \vdash (B \mid p \leq \texttt{r} \cdot y') \leq (B \mid q \leq \langle x_0, y'\rangle)$ due to $p \equiv \texttt{l} \cdot x, q \equiv \langle x, y_0\rangle, p \leq \texttt{r} \cdot y' \vdash q \leq \langle x_0, y'\rangle$ in the assertion logic, which is vacuously true from the implied antecedent $\texttt{l} \cdot x \leq \texttt{r} \cdot y'$

This completes our typechecking obligations, establishing monotonicity of zipping.

This chapter has been highly illustrative: on the one hand, it shows us that IRSAX and DRSAX are respectively capable of verifying complex termination and partial correctness

properties in a completely orthogonal manner as claimed by the thesis statement. On the other hand, it reveals some practical concerns regarding the expressive power of the underlying process calculus SAX (e.g., the ability to express reactive/select-based synchronization), the annotation burden of using refinements in the form of factoring definitions out, the need for indexed types, etc. Thus, we use the next chapter to reflect on what we have achieved and also what remains to be done.

Chapter 6

Conclusion and Future Work

> *How would you implement the type refinement systems proposed in your thesis?* —
>
> Brigitte Pientka

In this dissertation, we have evidenced the following thesis statement:

A proof-theoretic investigation of type refinements within the semi-axiomatic sequent calculus enables the verification of total correctness, decomposed into partial correctness and termination, for communicating processes.

To elaborate, we developed index and dependent type refinement systems that are capable of encoding and verifying termination and partial correctness specifications for SAX processes as sophisticated as those modelling asynchronous reactive programming. We have also successfully produced a theoretical framework for their design centered on proof theory, incorporating bidirectional typing, infinite proofs, and Hoare logic. Keeping in mind the limitations highlighted in the previous chapter, in the long term, we envision SAX as a core language for language-based process verification, thus requiring the follow-

ing features from contemporary sequential languages: further modes of type refinement, rich type structure, resource control, and effects. We consider the following avenues of future work to be the most critical.

- *Unifying index and dependent refinements*: Like Dunfield [Dun07], this dissertation treats index and (dependent) datasort refinement separately. However, Jaber and Riba [JR21] show how dependent refinements with assertions drawn from Jacobs' many-sorted coalgebraic modal logic [Jac01] can encode both safety (partial correctness) and liveness (termination/productivity) specifications orthogonally, raising the question: is a separate facility for termination checking actually necessary? The answer is *maybe not*: in a variation on SAX, DeYoung and Pfenning [DP23] replace certain addresses (essentially, those that are associated to types in output mode under bidirectional typing) with *projections* from which one can calculate the layout of data in memory. Curiously, projections along with the defunctionalized left axioms discussed in the previous section seem to correspond to the indices of Jacobs' modalities. Thus, we wonder whether said logic can be adapted to SAX, thus allowing index refinements as a subsystem for liveness specifications to be subsumed by dependent refinements.

- *Surface language and implementation*: in personal communication, DeYoung noticed that projections seem to be dual to Krishnaswami's patterns for interpreting pattern matching in the focused sequent calculus [Kri09]. Thus, a variation of said calculus may be a suitable surface language for SAX, provided that copattern matching for negative types is integrated [APTS13]. Moreover, a restriction of the bidirectional

presentation of SAX to *cyclic proofs* as a subsystem of infinite proofs generated by *regular coinduction* [Dag21] may enable decidable typechecking.

- *Modality and substructural typing*: adjoint SAX introduces a family of modalities that control occurrences of address weakening and contraction (address reuse and disposal) [PP21]. The notions of resource control induced by substructural typing could also be used to model *contention*, which is core to concurrency. Notably, Zeilberger's backwards bidirectional typing [Zei15] gives a deterministic algorithmic typing for the multiplicative fragment of the linear λ-calculus, obviating the need for *I/O contexts* [CHP96]. Because our bidirectional formulation of SAX has both conventional and backwards aspects, we wonder whether its linear reading is also suitably algorithmic even in the presence of additive connectives.

- *Asynchronous effects*: we conjecture that Ahman and Bauer's calculus for asynchronous effects [AP21] can be embedded into SAX, allowing the expression of effectful programs without, ideally, reaching for extralogical machinery.

- *Higher type structure*: while dependent type refinement enables type dependency on processes, process and type dependency on *types* (polymorphism and type nesting, respectively) are desirable for type abstraction. The combination of both features in the setting of session types was explored by Das et al. [DDMP21], which we suspect is adaptable to SAX.

Bibliography

[Abe06] Andreas Abel. Semi-Continuous Sized Types and Termination. In *Proceedings of the 20th International Conference on Computer Science Logic*, CSL'06, pages 72–88, Berlin, Heidelberg, September 2006. Springer-Verlag.

[Abe12] Andreas Abel. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. In Dale Miller and Zoltán Ésik, editors, *8th Workshop on Fixed Points in Computer Science (FICS 2012)*, volume 77 of *EPTCS*, pages 1–11, Tallinn, Estonia, March 2012.

[Abe14] Andreas Abel. Productive Infinite Objects via Copatterns and Sized Types in Agda. January 2014.

[AC96] David Aspinall and Adriana Compagnoni. Subtyping Dependent Types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, page 86, USA, July 1996. IEEE Computer Society.

[AND92] JEAN-MARC ANDREOLI. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297–347, June 1992.

[AP13] Andreas M. Abel and Brigitte Pientka. Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 185–196, New York, NY, USA, September 2013. Association for Computing Machinery.

[AP21] Danel Ahman and Matija Pretnar. Asynchronous Effects. *Proceedings of the ACM on Programming Languages*, 5(POPL):24:1–24:28, January 2021.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38, New York, NY, USA, 2013. Association for Computing Machinery.

[BBC08] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic Proofs of Program Termination in Separation Logic. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 101–112, New York, NY, USA, January 2008. Association for Computing Machinery.

[BC16] Christian J. Bell and Adam Chlipala. A Coinduction Proof Rule for Hoare Doubles. In *The 2nd International Workshop on Coq for PL (CoqPL 2016)*, 2016.

[BDS16] David Baelde, Amina Doumane, and Alexis Saurin. Infinitary Proof Theory: The Multiplicative Additive Case. In Jean-Marc Talbot and Laurent Regnier,

editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BFG+04] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, February 2004.

[BG16] Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 327–336, New York, NY, USA, July 2016. Association for Computing Machinery.

[BG22] Patrick Baillot and Alexis Ghyselen. Types for Complexity of Parallel Computation in Pi-calculus. *ACM Transactions on Programming Languages and Systems*, 44(3):15:1–15:50, July 2022.

[BH97] Michael Brandt and Fritz Henglein. Coinductive Axiomatization of Recursive Type Equality and Subtyping. In Philippe de Groote and J. Roger Hindley, editors, *Third International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, pages 63–81, Berlin, Heidelberg, 1997. Springer.

[BH10] Edwin Brady and Kevin Hammond. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Us-

age Protocols. *Fundamenta Informaticae*, 102(2):145–176, January 2010.

[BH19] Henning Basold and Helle Hvid Hansen. Well-definedness and observational equivalence for inductive–coinductive programs. *Journal of Logic and Computation*, 29(4):419–468, June 2019.

[BHN14] Nick Benton, Martin Hofmann, and Vivek Nigam. Abstract Effects and Proof-Relevant Logical Relations. *ACM SIGPLAN Notices*, 49(1):619–631, January 2014.

[Bla04] Frédéric Blanqui. A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems. In Vincent van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, pages 24–39, Berlin, Heidelberg, 2004. Springer.

[BM23] Patrick Bahr and Rasmus Ejlers Møgelberg. Asynchronous Modal FRP. *Proceedings of the ACM on Programming Languages*, 7(ICFP):205:476–205:510, August 2023.

[BR09] Frédéric Blanqui and Cody Roux. On the Relation between Sized-Types Based Termination and Semantic Labelling. In Erich Grädel and Reinhard Kahle, editors, *20th International Workshop on Computer Science Logic, Held as Part of the 15th Annual Conference of the EACSL*, Lecture Notes in Computer Science, pages 147–162, Berlin, Heidelberg, 2009. Springer.

[CC79] Patrick Cousot and Radhia Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 82(1):43–57, January 1979.

[CFPP14] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 361–372, New York, NY, USA, January 2014. Association for Computing Machinery.

[CH11] Ioana Cristescu and Daniel Hirschkoff. Termination in a Pi-calculus with Subtyping. In *Proceedings of the 18th International Workshop on Expressiveness in Concurrency*, volume 64, pages 44–58, August 2011.

[Che22] Zilin Chen. A Hoare Logic Style Refinement Types Formalisation. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2022, pages 1–14, New York, NY, USA, September 2022. Association for Computing Machinery.

[CHP96] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient Resource Management for Linear Logic Proof Search. In Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors, *Extensions of Logic Programming*, Lecture Notes in Computer Science, pages 67–81, Berlin, Heidelberg, 1996. Springer.

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computation*, 14(2):261–300, September 2001.

[Cla77] Edmund M. Clarke. Program Invariants as Fixed Points (Preliminary Reports). In *18th Annual Symposium on Foundations of Computer Science*, pages 18–29, Providence, Rhode Island, USA, 1977. IEEE Computer Society.

[CLBed] Jonathan Chan, Yufeng Li, and William J. Bowman. Is sized typing for Coq practical? *Journal of Functional Programming*, 33:e1, 2023/ed.

[CP10] Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, pages 222–236, Berlin, Heidelberg, 2010. Springer.

[CS06] Iliano Cervesato and Andre Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Electronic Notes in Theoretical Computer Science*, 165:145–176, November 2006.

[CSW14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining Proofs and Programs in a Dependently Typed Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 33–45, New York, NY, USA, January 2014. Association for Computing Machinery.

[DA09] Nils Anders Danielsson and Thorsten Altenkirch. Mixing Induction and Coinduction, October 2009.

[DA10] Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, Declaratively. In Claude Bolduc, Jules Desharnais, and Béchir Ktari, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 100–118, Berlin, Heidelberg, 2010. Springer.

[Dag21] Francesco Dagnino. Foundations of Regular Coinduction. *Logical Methods in Computer Science*, Volume 17, Issue 4, October 2021.

[dBV19] Jan de Muijnck-Hughes, Edwin C. Brady, and Wim Vanderbauwhede. Value-Dependent Session Design in a Dependently Typed Language. In Francisco Martins and Dominic Orchard, editors, *Proceedings of The 11th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software*, volume 291 of *EPTCS*, pages 47–59, Prague, Czechia, 2019.

[DCPT12] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 228–242, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[DDMP21] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Subtyping on Nested Polymorphic Session Types, March 2021.

[de 91] N. G. de Bruijn. Telescopic Mappings in Typed Lambda Calculus. *Information and Computation*, 91(2):189–204, April 1991.

[DHP18] Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel Complexity Analysis with Temporal Session Types. *Proceedings of the ACM on Programming*

Languages, 2(ICFP):91:1–91:30, July 2018.

[DHS10] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in Impure Concurrent Languages. In Paul Gastin and François Laroussinie, editors, *Proceedings of the 21st International Conference on Concurrency Theory*, Lecture Notes in Computer Science, pages 328–342, Berlin, Heidelberg, 2010. Springer.

[DK19] Jana Dunfield and Neelakantan R. Krishnaswami. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *Proceedings of the ACM on Programming Languages*, 3(POPL):9:1–9:28, January 2019.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. *ACM Computing Surveys*, 54(5):98:1–98:38, May 2021.

[DP00] Rowan Davies and Frank Pfenning. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 198–208, New York, NY, USA, September 2000. Association for Computing Machinery.

[DP03] Jana Dunfield and Frank Pfenning. Type Assignment for Intersections and Unions in Call-by-Value Languages. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 250–266, Berlin, Heidelberg, 2003. Springer.

[DP04] Jana Dunfield and Frank Pfenning. Tridirectional Typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 281–292, New York, NY, USA, 2004. Association for Computing Machinery.

[DP20a] Ankush Das and Frank Pfenning. Session Types with Arithmetic Refinements. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[DP20b] Ankush Das and Frank Pfenning. Verified Linear Session-Typed Concurrent Programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20, pages 1–15, New York, NY, USA, September 2020. Association for Computing Machinery.

[DP22] Farzaneh Derakhshan and Frank Pfenning. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *Logical Methods in Computer Science*, Volume 18, Issue 2, May 2022.

[DP23] Henry DeYoung and Frank Pfenning. Data Layout from a Type-Theoretic Perspective. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 1 - Proceedings of MFPS XXXVIII, February 2023.

[DPP20] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-Axiomatic Sequent Calculus. In Zena M. Ariola, editor, *5th International Conference on For-*

mal Structures for Computation and Deduction (FSCD 2020), volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[DS06] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Information and Computation*, 204(7):1045–1082, July 2006.

[Dun07] Jana Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007.

[FP91] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. Association for Computing Machinery.

[GG13] Dennis Griffith and Elsa L. Gunter. LiquidPi: Inferrable Dependent Session Types. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 185–197, Berlin, Heidelberg, 2013. Springer.

[GHP09] Neil Ghani, Peter Hancock, and Dirk Pattinson. Representations of Stream Processors Using Nested Fixed Points. *Logical Methods in Computer Science*, Volume 5, Issue 3, September 2009.

[GM99] Joseph A. Goguen and Grant Malcolm. Hidden coinduction: Behavioural correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.

[GSK21] Christian Uldal Graulund, Dmitrij Szamozvancev, and Neel Krishnaswami. Adjoint Reactive GUI Programming. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 289–309, Cham, 2021. Springer International Publishing.

[Hal85] Robert H. Halstead. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2 edition, April 2016.

[HBK20] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-Type Based Reasoning in Separation Logic. In *Proceedings of the ACM on Programming Languages*, volume 4, pages 6:1–6:30, 2020.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa71] C. A. R. Hoare. Procedures and Parameters: An Axiomatic Approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics, pages 102–116, Berlin, Heidelberg, 1971. Springer.

[Hoa72] C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, December 1972.

[Hoa02] C. A. R. Hoare. Towards a Theory of Parallel Programming. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 231–244. Springer, New York, NY, 2002.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 410–423, New York, NY, USA, January 1996. Association for Computing Machinery.

[II06] Frédéric Blanqui (INRIA) and Colin Riba (INPL). Combining Typing and Size Constraints for Checking the Termination of Higher-Order Conditional Rewrite Systems. In Miki Hermann and Andrei Voronkov, editors, *13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 105–119, Berlin, Heidelberg, 2006. Springer.

[Jac01] Bart Jacobs. Many-Sorted Coalgebraic Modal Logic: A Model-Theoretic Study. *RAIRO - Theoretical Informatics and Applications*, 35(1):31–59, January 2001.

[JKJ+ed] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018/ed.

[JR21] Guilhem Jaber and Colin Riba. Temporal Refinements for Guarded Recursive Types. In Nobuko Yoshida, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 548–578, Cham, 2021. Springer International Publishing.

[JV21] Ranjit Jhala and Niki Vazou. Refinement Types: A Tutorial. *Foundations and Trends® in Programming Languages*, 6(3–4):159–317, October 2021.

[KPB15] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Linear and Dependent Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 17–30, New York, NY, USA, January 2015. Association for Computing Machinery.

[KRBJ12] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic Parallelism via Liquid Effects. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 45–54, New York, NY, USA, 2012. Association for Computing Machinery.

[Kri09] Neelakantan R. Krishnaswami. Focusing on Pattern Matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 366–378, New York, NY, USA, January 2009. Association for Computing Machinery.

[KS08] Naoki Kobayashi and Davide Sangiorgi. A Hybrid Type System for Lock-Freedom of Mobile Processes. *ACM Transactions on Programming Languages and Systems*, 32(5):16:1–16:49, May 2008.

[LDD+22] Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Polarized Subtyping. In Ilya Sergey, editor, *31st European Symposium on Programming*, Lecture Notes in Computer Science, pages 431–461, Cham, 2022. Springer International Publishing.

[LDM06] Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A Sequent Calculus for Type Theory. In Zoltán Ésik, editor, *Computer Science Logic*, Lecture Notes in Computer Science, pages 441–455, Berlin, Heidelberg, 2006. Springer.

[Lei17] K. Rustan M. Leino. Modeling Concurrency in Dafny. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *School on Engineering Trustworthy Software Systems (SETTS 2017)*, Lecture Notes in Computer Science, pages 115–142, Cham, 2017. Springer International Publishing.

[Lev99] Paul Blain Levy. Call-by-Push-Value: A Subsuming Paradigm. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA '99, pages 228–242, Berlin, Heidelberg, April 1999. Springer-Verlag.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, February 2009.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proceedings of the 28th ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '01, pages 81–92, New York, NY, USA, January 2001. Association for Computing Machinery.

[LM14] K. Rustan M. Leino and Michał Moskal. Co-induction Simply. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *Formal Methods*, Lecture Notes in Computer Science, pages 382–398, Cham, 2014. Springer International Publishing.

[LM16] Sam Lindley and J. Garrett Morris. Talking Bananas: Structural Recursion for Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 434–447, New York, NY, USA, September 2016. Association for Computing Machinery.

[LP09] William Lovas and Frank Pfenning. Refinement Types as Proof Irrelevance. In Pierre-Louis Curien, editor, *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, pages 157–171, Berlin, Heidelberg, 2009. Springer.

[LR19] Rodolphe Lepigre and Christophe Raffalli. Practical Subtyping for Curry-Style Languages. *ACM Transactions on Programming Languages and Systems*, 41(1):5:1–5:58, February 2019.

[MO22] Daniel Marshall and Dominic Orchard. Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types. In Marco Carbone and Rumyana Neykova, editors, *Proceedings of the 13th In-*

ternational Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, volume 356 of *EPTCS*, pages 1–11, Munich, Germany, 2022.

[MPR18] Brandon Moore, Lucas Peña, and Grigore Rosu. Program Verification by Coinduction. In Amal Ahmed, editor, *Proceedings of the 27th European Symposium on Programming*, Lecture Notes in Computer Science, pages 589–618, Cham, 2018. Springer International Publishing.

[MPV22] Lykourgos Mastorou, Nikolaos Papaspyrou, and Niki Vazou. Coinduction Inductively: Mechanizing Coinductive Proofs in Liquid Haskell. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, Haskell 2022, pages 1–12, New York, NY, USA, September 2022. Association for Computing Machinery.

[Nak00] H. Nakano. A Modality for Recursion. In *Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science*, pages 255–266, June 2000.

[NLSD14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In Zhong Shao, editor, *Proceedings of the 23rd European Symposium on Programming*, Lecture Notes in Computer Science, pages 290–310, Berlin, Heidelberg, 2014. Springer.

[NSGH22] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages*,

6(POPL):9:1–9:31, January 2022.

[O'H04] Peter W. O'Hearn. Resources, Concurrency and Local Reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory*, Lecture Notes in Computer Science, pages 49–67, Berlin, Heidelberg, 2004. Springer.

[Opp78] Derek C. Oppen. Reasoning about Recursively Defined Data Structures. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 151–157, New York, NY, USA, January 1978. Association for Computing Machinery.

[OYR09] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and Information Hiding. *ACM Transactions on Programming Languages and Systems*, 31(3):11:1–11:50, April 2009.

[Plo73] Gordon Plotkin. Lambda-Definability and Logical Relations. Edinburgh University, 1973.

[PP21] Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. *Journal of Logical and Algebraic Methods in Programming*, 120:100637, April 2021.

[PP22] Klaas Pruiksma and Frank Pfenning. Back to futures. *Journal of Functional Programming*, 32:e6, January 2022.

[PT00] Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

[PT19] Pierre-Marie Pédrot and Nicolas Tabareau. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. *Proceedings of the ACM on Programming Languages*, 4(POPL):58:1–58:28, December 2019.

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, August 1972. Association for Computing Machinery.

[Rey00] John C. Reynolds. The Meaning of Types - From Intrinsic to Extrinsic Semantics. *BRICS Report Series*, 7(32), June 2000.

[ROS98] J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[RP08] Yann Régis-Gianas and François Pottier. A Hoare Logic for Call-by-Value Functional Programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Proceedings of the 9th International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 305–335, Berlin, Heidelberg, 2008. Springer.

[Sac14] Jorge Luis Sacchini. Linear Sized Types in the Calculus of Constructions. In Michael Codish and Eijiro Sumii, editors, *12th International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science, pages 169–185, Cham, 2014. Springer International Publishing.

[San06] Davide Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 16(1):1–39, February 2006.

[Sch77] Kurt Schütte. *Proof Theory*. Grundlehren Der Mathematischen Wissenschaften. Springer Berlin, Heidelberg, 1 edition, 1977.

[SH21] Jonathan Sterling and Robert Harper. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *Journal of the ACM*, 68(6):41:1–41:47, October 2021.

[Smu69] Raymond M. Smullyan. Analytic cut. *The Journal of Symbolic Logic*, 33(4):560–564, January 1969.

[SMV15] César Santos, Francisco Martins, and Vasco Thudichum Vasconcelos. Deductive Verification of Parallel Programs using Why3. In Sophia Knight, Ivan Lanese, Alberto Lluch-Lafuente, and Hugo Torres Vieira, editors, *Proceedings 8th Interaction and Concurrency Experience*, volume 189 of *EPTCS*, pages 128–142, Grenoble, France, 2015.

[SP22] Siva Somayyajula and Frank Pfenning. Type-Based Termination for Futures. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:21, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[SP23] Siva Somayyajula and Frank Pfenning. Dependent Type Refinements for Futures. In *39th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIX)*, Bloomington, Indiana, USA, June 2023.

[SPTD16] Paula Severi, Luca Padovani, Emilio Tuosto, and Mariangiola Dezani-Ciancaglini. On Sessions and Infinite Data. In Alberto Lluch Lafuente and José Proença, editors, *18th IFIP WG 6.1 International Conference on Coordination Models and Languages, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques (DisCoTec 2016)*, Lecture Notes in Computer Science, pages 245–261, Cham, 2016. Springer International Publishing.

[SRF+20] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proceedings of the ACM on Programming Languages*, 4(ICFP):121:1–121:30, August 2020.

[SYB19] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Effpi: Verified Message-Passing Programs in Dotty. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, Scala '19, pages 27–31, New York, NY, USA, July 2019. Association for Computing Machinery.

[TB20] Gadi Tellez and James Brotherston. Automatically Verifying Temporal Properties of Pointer Programs with Cyclic Proof. *Journal of Automated Reasoning*, 64(3):555–578, March 2020.

[TCP11] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent Session Types via Intuitionistic Linear Type Theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, pages 161–172, New York, NY, USA, July 2011. Association for Computing Machinery.

[TCP13] Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Proceedings of the 22nd European Symposium on Programming*, ESOP'13, pages 350–369, Berlin, Heidelberg, March 2013. Springer-Verlag.

[TCP21] Bernardo Toninho, Luís Caires, and Frank Pfenning. A Decade of Dependent Session Types. In *23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2021, pages 1–3, New York, NY, USA, October 2021. Association for Computing Machinery.

[TG03] René Thiemann and Jürgen Giesl. Size-Change Termination for Term Rewriting. In Robert Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, pages 264–278, Berlin, Heidelberg, 2003. Springer.

[TV19] Peter Thiemann and Vasco T. Vasconcelos. Label-Dependent Session Types. *Proceedings of the ACM on Programming Languages*, 4(POPL):67:1–67:29, December 2019.

[TY17] Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. *Journal of Logical and Algebraic Methods in Programming*, 90:61–83, August 2017.

[TY18] Bernardo Toninho and Nobuko Yoshida. Depending on Session-Typed Processes. In Christel Baier and Ugo Dal Lago, editors, *21st International Conference on Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 128–145, Cham, 2018. Springer International Publishing.

[Vez15] Andrea Vezzosi. Total (Co)Programming with Guarded Recursion. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, pages 77–78, Tallinn, Estonia, 2015.

[VRJ13] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract Refinement Types. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 209–228, Berlin, Heidelberg, 2013. Springer.

[VSJ+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, August 2014. Association for Computing Machinery.

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A Concurrent Logical Framework: The Propositional Fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *3rd Annual Workshop of the Types Working Group*, Lecture Notes in Computer Science, pages 355–377, Berlin, Heidelberg, 2004. Springer.

[WF94] A.K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, November 1994.

[WX17] Hanwen Wu and Hongwei Xi. Dependent Session Types, April 2017.

[Xi01] Hongwei Xi. Dependent Types for Program Termination Verification. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 231–242, June 2001.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, January 1999. Association for Computing Machinery.

[YBH01] N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π-Calculus. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 311–322, June 2001.

[Yoc89] Serge Yoccoz. Recursive ω-Rule for Proof Systems. *Information Processing Letters*, 31(6):291–294, June 1989.

[Zei08] Noam Zeilberger. Focusing and Higher-Order Abstract Syntax. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 359–369, New York, NY, USA, January 2008. Association for Computing Machinery.

[Zei15] Noam Zeilberger. Balanced polymorphism and linear lambda calculus. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, Tallinn, Estonia, May 2015.