Type Refinements for Asynchronously Communicating Processes

Thesis Proposal

Siva Somayyajula

November 12, 2022

Contents

1 Introduction

Formal verification of concurrent programs involves:

1. A specification of processes and process interaction
2. A logic to reason about such interactions

Then, program analyses are phrased as (dis)proving theorems in the logic. While the role of formal logic is clear in the latter part, representing processes and their interactions as logical phenomena is an area of active research. To that end, Watkins et al. [99] identify two paradigms with respect to a logical theory T:

	formulas-as-processes	proofs-as-processes
part 1/process	formula ϕ in T	proof of ϕ
part 1/process interaction	search for proof of ϕ	proof reduction
part 2/reasoning	meta-logic of T	T

While formulas-as-processes typically demands a separate meta-logic to reason about process interaction, proofs-as-processes implies that formulas are types. Following the Curry-Howard tradition, a theorem about a process interaction can be written as a type and proved by writing a process of said type. In this paradigm, dependent types are desirable for fine-grained reasoning. Contemporary dependent type theories with concurrency [89, 95] typically focus on the message-passing dynamics reflected by session-typed processes. In a survey of dependent session types, Toninho et al. [92] identify three avenues for future work:

1. Processes as proofs—the state-of-the-art [95] permits (linear) session dependency on functional values as well as (non-linear) functional dependency on processes. Since one sub-system embeds into the other, is it possible to remove the distinction between processes and proofs?
2. Encoding nested finite and infinite behaviors classified by mixed inductive-coinductive types
3. Enabling the implementation of concurrent systems (which use shared state and non-determinism effects)

In general, a concurrent type theory would also abstract over operational semantics, e.g., uniformly treat shared memory and message passing dynamics. The goal of this thesis is to work within this context and develop a unifying solution to the questions of type dependency and recursion. In this proposal, we aim to:

1. Identify a suitable core language for (asynchronously) communicating processes

2. Extend it with mixed inductive-coinductive and dependent types

3. Develop the requisite metatheory

This document is structured as follows.

- Section 2 recaps DeYoung et al. [40]'s semi-axiomatic sequent calculus (SAX), a core language for asynchronous communication with both futures-based [47] and message-passing dynamics [72] as well as structural and substructural typing disciplines.[1] Moreover, sequential call-by-need and call-by-value strategies are identified as different schedules. This proposal focuses on the structural variant with futures-based semantics (goal 1).

- Section 3 reviews the futures-based dynamics of SAX. As a warm-up to dealing with recursion, we also prove termination of well-typed programs [81] (goal 3).

- Section 4 develops Refined SAX (RSAX), an extension of SAX with type refinements from an arbitrary logical theory. With arithmetic refinements, RSAX encodes sized type refinements for mixed inductive-coinductive programming [81] (goal 2). Then, we extend our termination result to RSAX (goal 3).

- Section 5 proposes Dependent RSAX (DRSAX), an extension of RSAX with dependent types (goal 2). DRSAX simulates process interaction in the refinement theory by exposing modal necessity from dynamic logic [70], enabling unlimited type dependency on processes (goal 1). As a case study, we show how to reason about observational equality [5, 11] of processes in the presence of mixed inductive-coinductive types.

- Section 6 summarizes the pending and future work

In sum, we evidence the following thesis statement:

> **Thesis**: As a unifying principle, type refinements for asynchronously communicating processes enables mixed inductive-coinductive programming as well as program reasoning with dependent types.

2 Semi-Axiomatic Sequent Calculus (SAX)

In its full generality, SAX enjoys mixed modes of substructurality as well as both futures-based and message-passing semantics. Due to the technical issues surrounding substructural dependent types, this thesis is focused on the original, structural type system,

[1]While linear futures and message passing are weakly bisimilar [72], weakening and contraction are respectively interpreted as shared memory in the futures model, whereas in message passing, they are interpreted as service cancellation, replication, and message multicast [71].

which naturally has a futures-based interpretation. Logically, SAX follows from observations about type polarity [6, @Levy04]. 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 noninvertible, 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. As a result, we define types stratified by their polarity (without shifts, unlike call-by-push-value [58]).

Definition 1 (Types)**.** Positive and negative types are defined below.

$$\begin{aligned} A^+, B^+ &:= \oplus\{\ell : A_\ell\}_{\ell \in S} \mid \mathbf{1} \mid A \otimes B \\ A^-, B^- &:= \&\{\ell : A_\ell\}_{\ell \in S} \mid A \to B \\ A, B, C &:= A^+ \mid B^- \end{aligned}$$

There are positive ($\otimes$, $\mathbf{1}$) and negative ($\&$) conjunctions, disjunction ($\oplus$), and implication ($\to$).

The key innovation of SAX 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.

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

Because the axiomatic rules drop the premises of their sequent calculus counterparts, cut reduction corresponds to asynchronous communication just as the standard sequent calculus models synchronous communication [21]. In the futures-based interpretation of SAX, the sequent becomes the typing judgment[2]:

$$\overbrace{x : A, y : B, \ldots}^{\Gamma} \vdash P(x, y, \ldots, z) :: (z : C)$$

which reads "the *process* P reads from *source* addresses $x, y, \ldots$ and writes to the *destination* address z." P blocks if it reads from $x, y, \ldots$ before they are initialized. Dually, P performs a non-blocking write to z exactly once, representing the initialization of a future [47]. Moreover, the types $A, B, \ldots, C$ restrict the shape of the contents of the memory addressed by $x, y, \ldots, z$. We will now examine the rules for each connective and justify their computational interpretation via cut reduction. First, let us review the syntax for addresses and processes.

[2]The labeled succedent $z : C$ is typical of sequent calculi [20] and allows presentational symmetry. For example, the continuations typed by $\otimes$L and $\to$R are identical.

Definition 2 (Address)**.** Addresses $s, t := x \mid a$ are either address variables $x, y, z, \ldots$ or address values $a, b, c, \ldots$. The latter are only relevant at runtime (discussed in the next section).

Definition 3 (Address variable binding conventions)**.** SAX evinces a trichotomy of values V, processes P, and continuations K subject to the following conventions for binding address variables.

- A value V may contain address variables, the set of which is denoted $\mathrm{vars}(V)$, but may not bind them
- A process P may bind an address variable x with the syntax $x.\,P(x)$ where $P(x)$ indicates that x may occur in P (cf. abstract binding trees [cite:@Harper2016PFPL]). The substitution of an address t for x in P is written $P(t)$.
- A continuation K of the form $V \mapsto P(\mathrm{vars}(V))$, where V *only* contains address variables, binds $\mathrm{vars}(V)$

Definition 4 (Process)**.** The abstract syntax for processes is given by the following grammar, taken up-to renaming of address variables.

$P, Q :=$	$s \to t$	copy contents of s to t
$\mid$	$(x.\,P(x)) \| (y.\,Q(y))$	for fresh a, spawn $P(a)$ to write to a and concurrently proceed as $Q(a)$, which may read from a
$\mid$	$t \circ V$	pass V to continuation stored in t, or write value V to t
$\mid$	t **match** K	pass value stored in t to continuation K, or write continuation K to t

The first two kinds of processes correspond to the identity and cut rules; the concrete syntax for the latter is $x \leftarrow P(x); Q(x)$ to indicate that the bound address variables in P and Q are instantiated with the same address value at runtime. The remaining constructs correspond to logical right and left rules (shown in the table below to the left). Each such rule distinguishes an address variable x as well as a value V or continuation K, specified on a per-type basis (shown in the table below to the right).

polarity	right rule	left rule
positive	$x \circ V$	x **match** K
negative	x **match** K	$x \circ V$

type(s)	value V	continuation K
$\mathbf{1}$	$\langle\rangle$	$\langle\rangle \mapsto P$
$\otimes, \to$	$\langle s, t\rangle$	$\langle y, z\rangle \mapsto P(y, z)$
$\&, \oplus$	$\ell \cdot t$	$\{\ell \cdot y \mapsto P_\ell(y)\}_{\ell \in S}$

Let us begin with the identity and cut rules. We will show cut reductions via the relation $\rightsquigarrow$.

Identity and Cut. The cut rule drives our futures-based model. The process $x \leftarrow P(x); Q(x)$ at once spawns a process $P(x)$ which populates x as well as a process $Q(x)$ which may perform a blocking read from x. Based on the following initial cut reductions, the identity rule $x \rightarrow y$ copies the contents of x to y.

$$\frac{}{\Gamma, x : A \vdash x \rightarrow y :: (y : A)}\ \text{id} \qquad \frac{\Gamma \vdash P(x) :: (x : A) \quad \Gamma, x : A \vdash Q(x) :: (z : C)}{\Gamma \vdash x \leftarrow P(x); Q(x) :: (z : C)}\ \text{cut}$$

$$y \leftarrow (x \rightarrow y)\,; Q(y) \rightsquigarrow Q(x)$$

$$x \leftarrow P(x); (x \rightarrow y) \rightsquigarrow P(y)$$

Disjunction and Negative Conjunction. Disjunction $\oplus\{\ell : A_\ell\}_{\ell \in S}$ corresponds to the (eager) variant record type, so its right rule writes a tagged address y of type A_k for some $k \in S$ whereas its left rule matches on the choice of k then proceeds with the continuation process $P_k(y)$. Viewing such a pattern matching continuation as a record, negative conjunction corresponds to the lazy record type with the roles of its rules reversed.

$$\frac{k \in S}{\Gamma, y : A_k \vdash x \circ k \cdot y :: (x : \oplus\{\ell : A_\ell\}_{\ell \in S})}\ \oplus\text{R} \qquad \frac{\{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S}, y : A_k \vdash P_k(y) :: (z : C)\}_{k \in S}}{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S} \vdash x\, \textbf{match}\, \{\ell \cdot y \mapsto P_\ell(y)\}_{\ell \in S} :: (z : C)}\ \oplus\text{L}$$

$$\frac{\{\Gamma \vdash P(y) :: (y : A_\ell)\}_{\ell \in S}}{\Gamma \vdash x\, \textbf{match}\, \{\ell \cdot y \mapsto P_\ell(y)\}_{\ell \in S} :: (x : \&\{\ell : A_\ell\}_{\ell \in S})}\ \&\text{R} \qquad \frac{k \in S}{\Gamma, x : \&\{\ell : A_\ell\}_{\ell \in S} \vdash x \circ k \cdot y :: (y : A_k)}\ \&\text{L}$$

$$x \leftarrow x \circ k \cdot y; x\, \textbf{match}\, \{\ell \cdot y \mapsto Q_\ell(y)\}_{\ell \in S} \rightsquigarrow Q_k(y)$$

$$x \leftarrow x\, \textbf{match}\, \{\ell \cdot y \mapsto P_\ell(y)\}_{\ell \in S}; x \circ k \cdot y \rightsquigarrow P_k(y)$$

Positive Conjunction and Implication. Positive conjunction corresponds to the eager product type, so its right rule writes a pair of sources $\langle s, t\rangle$ to some destination x. Dually, its left rule reads the pair $\langle y, z\rangle$ from x and passes it to the continuation's process $Q(y, z)$. The rules for $\mathbf{1}$ degenerate accordingly.

$$\frac{}{\Gamma, y : A, z : B \vdash x \circ \langle y, z\rangle :: (x : A \otimes B)}\ \otimes\text{R} \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \vdash P(y, z) :: (w : C)}{\Gamma, x : A \otimes B \vdash x\, \textbf{match}\, (\langle y, z\rangle \mapsto P(y, z)) :: (w : C)}\ \otimes\text{L}$$

$$\frac{}{\Gamma \vdash x \circ \langle\rangle :: (x : \mathbf{1})}\ \mathbf{1}\mathrm{R} \qquad \frac{\Gamma, x : \mathbf{1} \vdash P :: (z : C)}{\Gamma, x : \mathbf{1} \vdash x\ \mathbf{match}\ (\langle\rangle \mapsto P) :: (z : C)}\ \mathbf{1}\mathrm{L}$$

$$x \leftarrow x \circ \langle y, z\rangle; x\ \mathbf{match}\ (\langle y, z\rangle \mapsto Q(y,z)) \rightsquigarrow Q(y,z)$$

$$x \leftarrow x \circ \langle\rangle; x\ \mathbf{match}\ (\langle\rangle \mapsto Q) \rightsquigarrow Q$$

Dual to positive conjunction, implication types destination-passing style functions. Its right rule writes the continuation $\langle y, z\rangle \mapsto P(y,z)$ which may read from the source/argument y *and* must write the result of function application to the destination z. Thus, the left rule performs this style of application by passing it a suitable source/argument and destination.

$$\frac{\Gamma, y : A \vdash P(y,z) :: (z : B)}{\Gamma \vdash x\ \mathbf{match}\ (\langle y, z\rangle \mapsto P(y,z)) :: (x : A \to B)}\ {\to}\mathrm{R} \qquad \frac{}{\Gamma, x : A \to B, y : A \vdash x \circ \langle y, z\rangle :: (z : B)}\ {\to}\mathrm{L}$$

$$x \leftarrow x\ \mathbf{match}\ (\langle y, z\rangle \mapsto P(y,z)); x \circ \langle y, z\rangle \rightsquigarrow P(y,z)$$

Destination-passing style is essential to futures-based parallelism—one must be able to refer to the result of function application even if it has not yet been initialized.

Before discussing the metatheory for simply-typed SAX, please see appendix B of [40] for example programs—we reproduce one below.

Example 1 (SAX Program)**.** Let us write a program with the signature $x : A \to B, y : B \to C \vdash P :: (z : A \to C)$; P is defined as follows.

$$P \triangleq z\ \mathbf{match}\ (\langle u, w\rangle \mapsto u' \leftarrow x \circ \langle u, u'\rangle; y \circ \langle u', w\rangle)$$

The first function call, to x, produces the intermediate result u' of type B. The subsequent call, to y, yields the final result w of type C.

2.1 Related Work

As a polarized language capable of expressing call-by-need and call-by-value evaluation strategies, SAX is comparable extended call-by-push-value [62], which has an additional construct for call-by-need.

3 Futures-Based Semantics

Parallel dynamics of SAX processes is assigned to configurations of processes and their mechanism of communication (future cells or messages) [40, 72, 71]. In this section, we will review the futures-based semantics of configurations and, in anticipation of introducing recursion, prove that well-typed configurations terminate.

Definition 5 (Configuration)**.** A configuration $\mathcal{C}$ is defined by the following grammar.

$C := \cdot$	empty configuration
$\mid \operatorname{proc} a\, P$	process P writing to cell addressed by a
$\mid !\mathrm{cell}\, a\, W$	future (persistent, marked with !) cell addressed by a with contents $W := V \mid K$
$\mid \mathcal{C}, \mathcal{C}$	concatenation of two configurations

$\mathcal{C}$ denotes a multiset of objects (processes and cells), so $(\cdot)$ and $(,)$ form a commutative monoid. We also require that a runtime address refers to at most one object in $\mathcal{C}$. Lastly, a configuration $\mathcal{F}$ is final iff it only contains future cells.

Now, let Σ and Δ be runtime contexts that associate runtime addresses to types (as opposed to address variables). By convention, substitution of address variables for runtime addresses must be well-typed, i.e., $[a : A, \ldots, b : B / x : A, \ldots, y : B]P$ is well-defined. Then, the configuration typing judgment given in figure 1, $\Sigma \vdash \mathcal{C} :: \Delta$, means that the objects in $\mathcal{C}$ are well-typed with sources in Σ and destinations in Δ.[3]

$$\frac{\Gamma \vdash P :: (z : C)}{\Sigma \vdash \operatorname{proc} a\,([\Sigma, c : C / \Gamma, z : C]P) :: (\Sigma, c : C)}\ \mathrm{proc}$$

$$\frac{\Gamma \vdash z \circ V :: (z : C)}{\Sigma \vdash !\mathrm{cell}\, c\,([\Sigma, c : C / \Gamma, z : C]V) :: (\Sigma, c : C)}\ !\mathrm{cell}_V \qquad \frac{\Gamma \vdash z\ \mathbf{match}\ K :: (z : C)}{\Sigma \vdash !\mathrm{cell}\, a\,([\Sigma, c : C / \Gamma, z : C]K) :: (\Sigma, c : C)}\ !\mathrm{cell}_K$$

$$\frac{}{\Sigma \vdash \cdot :: \Sigma}\ \mathrm{empty} \qquad \frac{\Sigma \vdash \mathcal{C} :: \Sigma' \quad \Sigma' \vdash \mathcal{C}' :: \Delta}{\Sigma \vdash \mathcal{C}, \mathcal{C}' :: \Delta}\ \mathrm{join}$$

Figure 1: Configuration Typing

Configuration reduction $\to$ is given as multiset rewriting rules [22] in figure 2, which replace any subset of a configuration matching the left-hand side with the right-hand side. Formally, a configuration of objects can be represented by a multiplicative conjunction of linear logic propositions representing each object. Then, rewrite rules are linear

[3] The typing rules preserve the invariant $\Sigma \subseteq \Delta$ because future cells are persistent.

implications and fresh runtime addresses are generated by existential quantification over a sort of names [68]. Principal cuts encountered in a configuration are resolved by passing a value to a continuation also given in figure 2 as the relation $V \triangleright K = P$ (relatedly, see K-machines in [48]).

$$
\begin{aligned}
!\mathsf{cell}\, a\, W, \mathsf{proc}\, b\, (a \to b) &\to\ !\mathsf{cell}\, b\, W \\
\mathsf{proc}\, c\, (x \leftarrow P(x); Q(x)) &\to \\
\mathsf{proc}\, a\, (P(a)), \mathsf{proc}\, c\, (Q(a)) &\text{ where } a \text{ is fresh} \\
!\mathsf{cell}\, a\, K, \mathsf{proc}\, c\, (a \circ V) &\to \mathsf{proc}\, c\, (V \triangleright K) \\
!\mathsf{cell}\, a\, V, \mathsf{proc}\, c\, (a\ \mathbf{match}\ K) &\to \mathsf{proc}\, c\, (V \triangleright K) \\
\mathsf{proc}\, a\, (a \circ V) &\to\ !\mathsf{cell}\, a\, V \\
\mathsf{proc}\, a\, (a\ \mathbf{match}\ K) &\to\ !\mathsf{cell}\, a\, K
\end{aligned}
\qquad
\begin{aligned}
\langle\rangle \triangleright \langle\rangle \mapsto P &= P \\
\langle a, b\rangle \triangleright (\langle x, y\rangle \mapsto P(x, y)) &= P(a, b) \\
k \cdot a \triangleright \{\ell \cdot x \mapsto P_\ell(x)\}_{\ell \in S} &= P_k(a)
\end{aligned}
$$

Figure 2: Operational Semantics

The first rule for $\to$ corresponds to the identity rule and copies the contents of one cell into another. The second rule, which is for cut, models computing with futures [47]: it allocates a new cell to be populated by the newly spawned P. Concurrently, Q may read from said new cell, which blocks if it is not yet initialized. The third and fourth rules resolve principal cuts by passing a value to a continuation. Lastly, the final two rules perform the action of writing to a cell.

3.1 Termination

Now, we are ready to prove termination of configuration reduction. In contrast to the original version of this proof [40], we explicitly model types as sets of terminating configurations [85, 69, 82]. In the further proposed work, we aim to reason about semantic (co)inductive types using this model. This sub-section is outlined as follows:

- We define semantic types: sets of terminating configurations with the ancillary properties to complete the termination proof.
- We show that semantic versions of the syntactic typing rules of processes, configuration objects, and configurations are admissible in this model.
- We prove the fundamental theorem that every well-typed configuration is in the model. Normalization (the existence of *a* normal form, i.e., final configuration reduct) for *closed* configurations (where $\cdot \vdash \mathcal{C} :: \Delta$) is a corollary.
- Termination of *open* configurations (where $\Sigma \vdash \mathcal{C} :: \Delta$) is a corollary of the fundamental theorem and the diamond property [7].

Now, let us begin with the definition of semantic type.

Definition 6 (Semantic type)**.** A semantic type $\mathscr{A}, \mathscr{B}, \ldots$ is a set of pairs of addresses and final configurations, writing $\mathcal{F} \in [a : \mathscr{A}]$ for $(a, \mathcal{F}) \in \mathscr{A}$, such that if $\mathcal{F} \in [a : \mathscr{A}]$, then:

1. Inversion: $!\mathrm{cell}\, a\, W \in \mathcal{F}$ for some W.
2. Contraction: $\mathcal{F}, !\mathrm{cell}\, b\, W \in [b : \mathscr{A}]$ for fresh b (W is from point 1).
3. Weakening: $\mathcal{F}, \mathcal{F}' \in [a : \mathscr{A}]$ for all $\mathcal{F}'$.

Let $\to^*$ be multi-step reduction and $\mathcal{C} \in [\![a : \mathscr{A}]\!] :\Leftrightarrow \mathcal{C} \to^* \mathcal{F}$ and $\mathcal{F} \in [a : \mathscr{A}]$.

Conditions 1 and 2 are required to reproduce the identity rule semantically, whereas condition 3 aggregates the semantic type ascriptions of different sub-configurations. In the next definition, we define each semantic type in **boldface** based on its syntactic counterpart.

Definition 7 (Semantic types)**.**

- $\mathcal{F} \in [a : \mathbb{1}] \triangleq \mathcal{F} = \mathcal{F}', !\mathrm{cell}\, a\, \langle\rangle$.
- $\mathcal{F} \in [c : \mathscr{A} \boldsymbol{\otimes} \mathscr{B}] \triangleq \mathcal{F} = \mathcal{F}', !\mathrm{cell}\, c\, \langle a, b\rangle$ where $\mathcal{F}' \in [a : \mathscr{A}]$ and $\mathcal{F}' \in [b : \mathscr{B}]$.
- $\mathcal{F} \in [c : \mathscr{A} \boldsymbol{\to} \mathscr{B}] \triangleq !\mathrm{cell}\, c\, K \in \mathcal{F}$ for some K and $\mathcal{F}, \mathcal{F}', \mathrm{proc}\, b\, (c \circ \langle a, b\rangle) \in [\![b : \mathscr{B}]\!]$ for all $\mathcal{F}'$ such that $\mathcal{F}, \mathcal{F}' \in [a : \mathscr{A}]$ where a and b are fresh.
- $\mathcal{F} \in [b : \boldsymbol{\oplus}\{\ell : \mathscr{A}_\ell\}_{\ell \in S}] \triangleq \mathcal{F} = \mathcal{F}', !\mathrm{cell}\, b\, (k \cdot a)$ and $\mathcal{F}' \in [a : \mathscr{A}_k]$ for some $k \in S$.
- $\mathcal{F} \in [b : \boldsymbol{\&}\{\ell : \mathscr{A}_\ell\}_{\ell \in S}] \triangleq !\mathrm{cell}\, b\, K \in \mathcal{F}$ for some K and $\mathcal{F}, \mathrm{proc}\, a\, (b \circ k \cdot a) \in [\![a : \mathscr{A}_k]\!]$ for all $k \in S$ and where a is fresh.

Positive semantic types are defined by structure—the contents of a particular cell—whereas negative semantic types are defined by behavior—how interacting with a configuration produces the desired result (see also the semantics of CBPV [58]). Analogously for the λ-calculus, the semantic positive product is defined as containing pairs of normalizing terms, whereas the semantic function space contains all terms that normalize under application [45, 4]. For example, let us verify that $\boldsymbol{\otimes}$ is a well-defined semantic type constructor.

Remark. Recall $F \in [c : \mathscr{A} \boldsymbol{\otimes} \mathscr{B}] \triangleq \mathcal{F} = \mathcal{F}', !\mathrm{cell}\, c\, \langle a, b\rangle$ where $\mathcal{F}' \in [a : \mathscr{A}]$ and $\mathcal{F}' \in [b : \mathscr{B}]$. We need to show that conditions 1 through 3 hold.

1. *Inversion*: Immediate, $W \triangleq \langle a, b\rangle$.
2. *Contraction*: For fresh d, we want to show $\mathcal{F}, !\mathrm{cell}\, d\, \langle a, b\rangle \in [d : \mathscr{A} \boldsymbol{\otimes} \mathscr{B}]$. Because $\mathcal{F}' \in [a : \mathscr{A}]$ and $\mathcal{F}' \in [b : \mathscr{B}]$, we have $\mathcal{F} \in [a : \mathscr{A}]$ and $\mathcal{F} \in [b : \mathscr{B}]$ by condition 3 of $\mathscr{A}$ and $\mathscr{B}$, as desired.

3. *Weakening*: For all $\mathcal{F}''$; $\mathcal{F}, \mathcal{F}'' = \mathcal{F}', \mathcal{F}'', !\mathrm{cell}\, c\, \langle a, b\rangle$ and $\mathcal{F}', \mathcal{F}'' \in [a : \mathscr{A}]$ as well as $\mathcal{F}', \mathcal{F}'' \in [b : \mathscr{B}]$ due to condition 3 of both $\mathscr{A}$ and $\mathscr{B}$.

Now, to state the semantic typing rule lemmas, we need to define the semantic typing judgment.

Definition 8 (Semantic typing judgment)**.** Let $\mathbf{\Sigma}$ and $\mathbf{\Delta}$ be contexts associating cell addresses to semantic types.

- $\mathcal{F} \in [\mathbf{\Sigma}] \triangleq \mathcal{F} \in [a : \mathscr{A}]$ for all $a : \mathscr{A} \in \mathbf{\Sigma}$.
- $\mathcal{C} \in [\![\mathbf{\Sigma}]\!] \triangleq \mathcal{C} \mapsto^* \mathcal{F}$ and $\mathcal{F} \in [\mathbf{\Sigma}]$.
- $\mathbf{\Sigma} \vDash \mathcal{C} :: \mathbf{\Delta} \triangleq$ for all $\mathcal{F} \in [\mathbf{\Sigma}]$, we have $\mathcal{F}, \mathcal{C} \in [\![\mathbf{\Delta}]\!]$.

In natural deduction, the analogous judgment $\mathbf{\Sigma} \vDash e : \mathscr{A}$ is defined by quantifying over all closing value substitutions σ with domain $\mathbf{\Sigma}$, then stating $\sigma(e) \in \mathscr{A}$. Similarly, we ask whether the configuration $\mathcal{C}$ terminates at the desired semantic type(s) when "closed" by a final configuration F providing all the sources from which $\mathcal{C}$ reads. Immediately, we reproduce the standard backwards closure result.

Lemma 1 (Backward closure)**.** If $\mathcal{C} \to^* \mathcal{C}'$ and $\mathbf{\Sigma} \vDash \mathcal{C}' :: \mathbf{\Delta}$, then $\mathbf{\Sigma} \vDash \mathcal{C} :: \mathbf{\Delta}$.

We are finally ready to prove a representative sample of semantic typing rule lemmas, all of which are in figure 3 (the dashed lines indicate that they are lemmas). Afterwards, we can tackle objects and configurations. As promised, conditions 1 and 2 are used for the admissibility of the identity rule.

Lemma 2 (id)**.** $\mathbf{\Sigma}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (a \to b) :: (\mathbf{\Sigma}, a : \mathscr{A}, b : \mathscr{A})$

Proof. Assuming $\mathcal{F} \in [\mathbf{\Sigma}, a : \mathscr{A}]$, we want to show $\mathcal{F}, \mathrm{proc}\, b\, (a \to b) \in [\![\mathbf{\Sigma}, a : \mathscr{A}, b : \mathscr{A}]\!]$. By condition 1, $!\mathrm{cell}\, a\, W \in \mathcal{F}$. By condition 2, $\mathcal{F}, !\mathrm{cell}\, b\, W \in [b : \mathscr{A}]$. By condition 3, $\mathcal{F}, !\mathrm{cell}\, b\, W \in [\mathbf{\Sigma}, a : \mathscr{A}, b : \mathscr{A}]$. Since $\mathcal{F}, \mathrm{proc}\, b\, (a \to b) \to \mathcal{F}, !\mathrm{cell}\, b\, W$, we are done by lemma 1. □

The reader may have noticed that each semantic type's definition encodes its own noninvertible rule lemma, making the admissibility of rules like ⊗R immediate. Invertible rule lemmas require more effort; consider ⊗L below.

Lemma 3 (⊗L)**.** If $\mathbf{\Sigma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B} \vdash \mathrm{proc}\, d\, (P(a, b)) :: \mathbf{\Delta}, d : \mathscr{C}$, then $\mathbf{\Sigma}, c : \mathscr{A} \otimes \mathscr{B} \vdash \mathrm{proc}\, d\, (c\; \mathbf{match}\; (\langle x, y\rangle \mapsto P(x, y))) :: \mathbf{\Delta}, d : \mathscr{C}$.

Proof. Assuming $\mathcal{F} \in [\mathbf{\Sigma}, c : \mathscr{A} \otimes \mathscr{B}]$, we want to show that $\mathcal{F}, \mathrm{proc}\, d\, (c\; \mathbf{match}\; (\langle x, y\rangle \mapsto P(x, y))) \in [\![\mathbf{\Delta}, d : \mathscr{C}]\!]$. Since $\mathcal{F} \in [c : \mathscr{A} \otimes \mathscr{B}]$, we have $\mathcal{F} = \mathcal{F}', !\mathrm{cell}\, c\, \langle a, b\rangle$ where $\mathcal{F}' \in [a : \mathscr{A}]$ and $\mathcal{F}' \in [b : \mathscr{B}]$. As a result, both $\mathcal{F} \in [a : \mathscr{A}]$ and $\mathcal{F} \in [b : \mathscr{B}]$ by condition 3. In sum, $\mathcal{F} \in [\mathbf{\Sigma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B}]$, so by the premise, $\mathcal{F}, \mathrm{proc}\, d\, (P(a, b)) \in [\![\mathbf{\Delta}, d : \mathscr{C}]\!]$. Since $\mathcal{F}, \mathrm{proc}\, d\, (c\; \mathbf{match}\; (\langle x, y\rangle \mapsto P(x, y))) \to \mathcal{F}, \mathrm{proc}\, d\, (P(a, b))$, we are done by lemma 1. □

$$\frac{}{\mathbf{\Sigma}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (a \to b) :: \mathbf{\Sigma}, a : \mathscr{A}, b : \mathscr{A}}\ \mathrm{id} \qquad \frac{\mathbf{\Sigma} \vDash \mathrm{proc}\, a\, (P(a)) :: \mathbf{\Sigma}', a : \mathscr{A} \quad \mathbf{\Sigma}', a : A \vDash \mathrm{proc}\, c\, (Q(a)) :: \mathbf{\Delta}, c : \mathscr{C}}{\mathbf{\Sigma} \vDash \mathrm{proc}\, c\, (x \leftarrow P(x); Q(x)) :: \mathbf{\Delta}, c : \mathscr{C}}\ \mathrm{cut}$$

$$\frac{}{\mathbf{\Sigma} \vDash \mathrm{!cell}\, a\, \langle\rangle :: \mathbf{\Sigma}, a : \mathbb{1}}\ \mathbb{1}\mathrm{R} \qquad \frac{\mathbf{\Sigma}, a : \mathbb{1} \vDash \mathrm{proc}\, c\, P :: \mathbf{\Delta}, c : \mathscr{C}}{\mathbf{\Sigma}, a : \mathbb{1} \vDash \mathrm{proc}\, c\, (a\, \mathbf{match}\, (\langle\rangle \mapsto P)) :: \mathbf{\Delta}, c : \mathscr{C}}\ \mathbb{1}\mathrm{L}$$

$$\frac{}{\mathbf{\Sigma}, a : \mathscr{A}, b : \mathscr{B} \vDash \mathrm{!cell}\, c\, \langle a, b\rangle :: \mathbf{\Sigma}, a : \mathscr{A}, b : \mathscr{B}, c : \mathscr{A} \otimes \mathscr{B}}\ \otimes\mathrm{R} \qquad \frac{\mathbf{\Sigma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B} \vdash \mathrm{proc}\, d\, (P(a, b)) :: \mathbf{\Delta}, d : \mathscr{C}}{\mathbf{\Sigma}, c : \mathscr{A} \otimes \mathscr{B} \vDash \mathrm{proc}\, d\, (c\, \mathbf{match}\, (\langle x, y\rangle \mapsto P(x, y))) :: \mathbf{\Delta}, d : \mathscr{C}}\ \otimes\mathrm{L}$$

$$\frac{\mathbf{\Sigma}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (P(a, b)) :: \mathbf{\Delta}, b : \mathscr{B}}{\mathbf{\Sigma} \vDash \mathrm{!cell}\, c\, (\langle x, y\rangle \mapsto P(x, y)) :: \mathbf{\Delta}, c : \mathscr{A} \boldsymbol{\rightarrow} \mathscr{B}}\ \boldsymbol{\rightarrow}\mathrm{R} \qquad \frac{}{\mathbf{\Sigma}, c : \mathscr{A} \boldsymbol{\rightarrow} \mathscr{B}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (c \circ \langle a, b\rangle) :: \mathbf{\Sigma}, c : \mathscr{A} \boldsymbol{\rightarrow} \mathscr{B}, a : \mathscr{A}, b : \mathscr{B}}\ \boldsymbol{\rightarrow}\mathrm{L}$$

$$\frac{k \in S}{\mathbf{\Sigma}, a : \mathscr{A}_k \vDash \mathrm{!cell}\, b\, (k \cdot a) :: \mathbf{\Sigma}, a : \mathscr{A}_k, b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S}}\ \oplus\mathrm{R} \qquad \frac{\{\mathbf{\Sigma}, b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S}, a : \mathscr{A}_k \vDash \mathrm{proc}\, c\, (P_k(a)) :: \mathbf{\Delta}, c : \mathscr{C}\}_{k \in S}}{\mathbf{\Sigma}, b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S} \vDash \mathrm{proc}\, c\, (b\, \mathbf{match}\, \{\ell \cdot x \mapsto P_\ell(x)\}_{\ell \in S}) :: \mathbf{\Delta}, c : \mathscr{C}}\ \oplus\mathrm{L}$$

$$\frac{\{\mathbf{\Sigma} \vDash \mathrm{proc}\, a\, (P_k(a)) :: \mathbf{\Delta}, a : \mathscr{A}_k\}_{k \in S}}{\mathbf{\Sigma} \vDash \mathrm{!cell}\, b\, \{\ell \cdot x \mapsto P_\ell(x)\}_{\ell \in S} :: \mathbf{\Delta}, b : \boldsymbol{\&}\{\ell : \mathscr{A}_\ell\}_{\ell \in S}}\ \boldsymbol{\&}\mathrm{R} \qquad \frac{k \in S}{\mathbf{\Sigma}, b : \boldsymbol{\&}\{\ell : \mathscr{A}_\ell\}_{\ell \in S} \vDash \mathrm{proc}\, b\, (c \circ k \cdot a) :: \mathbf{\Sigma}, b : \boldsymbol{\&}\{\ell : \mathscr{A}_\ell\}_{\ell \in S}, a : \mathscr{A}_k}\ \boldsymbol{\&}\mathrm{L}$$

Figure 3: Semantic Object Typing Rule Lemmas I

Ironically, the persistence of a cell from the conclusion to the premise of a rule, which encodes contraction, is justified via condition 3 (semantic weakening). On the other hand, the identity rule, which "bakes in" weakening, is justified via condition 2 (semantic contraction). Now, to develop the fundamental theorem, we need a semantic interpretation of syntactic types.

Definition 9 (Semantic interpretation I)**.** We define the semantic interpretation of $(\!|A|\!)$ by induction on A.

$$(\!|\mathbf{1}|\!) \triangleq \mathbb{1} \qquad (\!|A \otimes B|\!) \triangleq (\!|A|\!) \boldsymbol{\otimes} (\!|B|\!) \qquad (\!|A \multimap B|\!) \triangleq (\!|A|\!) \boldsymbol{\multimap} (\!|B|\!)$$

$$(\!|\oplus\{\ell : A_\ell\}_{\ell\in S}|\!) \triangleq \boldsymbol{\oplus}\{\ell : (\!|A_\ell|\!)\}_{\ell\in S} \qquad (\!|\&\{\ell : A_\ell\}_{\ell\in S}|\!) \triangleq \boldsymbol{\&}\{\ell : (\!|A_\ell|\!)\}_{\ell\in S}$$

$(\!|\cdot|\!)$ is then extended to runtime contexts Σ and Δ in the obvious way.

Lemma 4 (Semantic object typing I)**.**

1. If $\Gamma \vdash z \circ V :: (z : C)$, then $(\!|\Sigma|\!) \vDash !\mathrm{cell}\, a\, ([\Sigma, c : C/\Gamma, z : C]V) :: (\!|\Sigma|\!), c : (\!|C|\!)$.
2. If $\Gamma \vdash z\, \mathbf{match}\, K :: (z : C)$, then $(\!|\Sigma|\!) \vDash !\mathrm{cell}\, a\, ([\Sigma, c : C/\Gamma, z : C]K) :: (\!|\Sigma|\!), c : (\!|C|\!)$.
3. If $\Gamma \vdash P :: (z : C)$, then $(\!|\Sigma|\!) \vDash \mathrm{proc}\, a\, ([\Sigma, c : C/\Gamma, z : C]P) :: (\!|\Sigma|\!), c : (\!|C|\!)$.

Proof. Part 1 follows by case analysis on the process typing derivation D applying the relevant semantic typing rule lemmas, like $\boldsymbol{\otimes}$R for $\otimes\mathrm{R}^\omega$. We prove parts 2 and 3 simultaneously by lexicographic induction on D then the part number. That is, part 2 refers to part 3 on the typing subderivation for the process contained in K (like $\multimap\mathrm{R}^\omega$). In part 3, if P reads a cell (like $\multimap\mathrm{L}^\omega$ or $\otimes\mathrm{L}^\omega$), then we invoke the relevant semantic typing rule lemma. If P writes a continuation K, then $\mathrm{proc}\, a\, (a\, \mathbf{match}\, K) \to !\mathrm{cell}\, a\, K$, so we invoke part 2 on D and conclude by lemma 1. Writing a value follows symmetrically, invoking part 1. $\square$

Now that processes and objects have been resolved, it remains to derive the semantic configuration typing rule lemmas.

Lemma 5 (Semantic configuration typing)**.**

Empty $\boldsymbol{\Sigma} \vDash \cdot :: \boldsymbol{\Sigma}$

Join If $\boldsymbol{\Sigma} \vDash \mathcal{C} :: \boldsymbol{\Sigma}'$ and $\boldsymbol{\Sigma}' \vDash \mathcal{C}' :: \boldsymbol{\Delta}$, then $\boldsymbol{\Sigma} \vDash \mathcal{C}, \mathcal{C}' :: \boldsymbol{\Delta}$.

The previous lemmas establish the fundamental theorem, of which normalization of closed configurations is a corollary.

Theorem 1 (Fundamental theorem)**.** If $\Sigma \vdash \mathcal{C} :: \Delta$, then $(\!|\Sigma|\!) \vDash \mathcal{C} :: (\!|\Delta|\!)$.

Proof. By induction on the configuration typing derivation D, the empty and join cases are discharged by lemma 5. The object typing cases are covered by lemma 4. □

Termination follows from normalization and the diamond property [7]. Below, let $\mathcal{C}_1 \sim \mathcal{C}_2$ iff $\mathcal{C}_1$ is equal to $\mathcal{C}_2$ up-to renaming of addresses.

Theorem 2 (Diamond property [40])**.** Assume $\Sigma \vdash \mathcal{C} :: \Delta$, $\mathcal{C} \to \mathcal{C}_1$, and $\mathcal{C} \to \mathcal{C}_2$ where $\mathcal{C}_1 \not\sim \mathcal{C}_2$. Then, $\mathcal{C}_1 \to \mathcal{C}_1'$ and $\mathcal{C}_2 \to \mathcal{C}_2'$ such that $\mathcal{C}_1' \sim \mathcal{C}_2'$.

Theorem 3 (Termination)**.** If $\Sigma \vdash \mathcal{C} :: \Delta$, then there are no infinite reduction sequences beginning with $\mathcal{C}$.

4 Refined SAX (RSAX)

Now that we have covered simply-typed SAX, we introduce type refinements. Following Das et al. [32, 31, 33], we internalize assertion and assumption of constraints $\phi, \psi, \ldots$, formulas of some refinement theory, via the following types: constraint conjunction and implication, respectively.

$$A^+, B^+ := \ldots \mid \phi \land A$$
$$A^-, B^- := \ldots \mid \phi \supset A$$

Given a constraint assertion judgment $\Gamma \vdash \phi$, these types introduce values $[t]$ and continuations $[y] \mapsto P(y)$ subject to the following typing and operational rules.

$$\frac{\Gamma \vdash \phi}{\Gamma, y : A \vdash x \circ [y] :: (x : \phi \land A)} \land\text{R} \qquad \frac{\Gamma, x : \phi \land A, \phi, y : A \vdash P(y) :: (w : C)}{\Gamma, x : \phi \land A \vdash x\ \mathbf{match}\ ([y] \mapsto P(y)) :: (w : C)} \land\text{L}$$

$$\frac{\Gamma, \phi \vdash P(y) :: (y : A)}{\Gamma \vdash x\ \mathbf{match}\ ([y] \mapsto P(y)) :: (x : \phi \supset A)} \supset\text{R} \qquad \frac{\Gamma \vdash \phi}{\Gamma, x : \phi \supset A \vdash x \circ [y] :: (y : A)} \supset\text{L}$$

$$[a] \triangleright ([x] \mapsto P(x)) = P(a)$$

It is also useful to add the process $P := \ldots \mid$ **impossible** that indicates a dead code branch due to inconsistent constraints. As a result, it does not appear at run-time.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \mathbf{impossible} :: (x : A)}$$

While dependent type theories typically collapse the distinction between types and constraints via the Curry-Howard correspondence, we rely on an open-world assumption:

we may instantiate the refinement theory with elements that depart from SAX. Arithmetic and many-sorted first-order logic enable sized type refinements in RSAX and refinement reflection in DRSAX, respectively.

To get off the ground, we will begin with arithmetic refinements, adding such constraints as comparison of arithmetic terms $e_1 < e_2$, etc. Permitting Γ to have arithmetic variables $i, j, k, \ldots$, arithmetic quantifier types introduce values $\langle e, t \rangle$ and continuations $\langle i, y \rangle \mapsto P(y)$ subject to the following rules where the judgment $\Gamma \vdash e$ determines that e is well-defined under the constraints in Γ.

$$\frac{\Gamma \vdash e}{\Gamma, y : A(e) \vdash x \circ \langle e, y \rangle :: (x : \exists i.\, A(i))}\ \exists R \qquad \frac{\Gamma, x : \exists i.\, A(i), i, y : A(i) \vdash P(i,y) :: (z : C)}{\Gamma, x : \exists i.\, A(i) \vdash x\ \mathbf{match}\ (\langle i, y \rangle \mapsto P(i,y)) :: (z : C)}\ \exists L$$

$$\frac{\Gamma, i \vdash P(i,y) :: (y : A(i))}{\Gamma \vdash x\ \mathbf{match}\ (\langle i, y \rangle \mapsto P(i,y)) :: (x : \forall i.\, A(i))}\ \forall R \qquad \frac{\Gamma \vdash e}{\Gamma, x : \forall i.\, A(i) \vdash x \circ \langle e, y \rangle :: (y : A(e))}\ \forall L$$

$$\langle n, a \rangle \triangleright (\langle i, x \rangle \mapsto P(i,x)) = P(n,a)$$

Unsurprisingly, they act like constraint conjunction and implication. In the next subsection, we will detail our incorporation of mixed inductive-coinductive programming via arithmetic refinements. For other use cases, refer to [32].

4.1 Sized Type Refinements

Adding (co)inductive types and terminating recursion (including productive corecursive definitions) to any programming language is a non-trivial task, since 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. Since inferring this exactly is intractable, languages like Agda (before version 2.4) [4] and Coq [88] 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. [50] and further developed by others [10, 13, 2, 4]. Inductive and coinductive types are indexed by the height and observable depth of their data and codata, respectively. Consider the equirecursive type definitions in example 2 adorned with sized type refinements: nat$[i]$ describes unary natural numbers less than or equal to i and stream$_A[i]$ describes infinite A-streams that allow the first $i + 1$ elements to be observed before reaching potentially undefined or divergent behavior.

Example 2 (Recursive types)**.**

$$\begin{aligned}\mathsf{nat}[i] &= \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ}\ : i > 0 \wedge \mathsf{nat}[i-1]\}\\ \mathsf{stream}_A[i] &= \&\{\mathsf{head} : A, \mathsf{tail} : i > 0 \supset \mathsf{stream}_A[i-1]\}\end{aligned}$$

Note that $\mathsf{stream}_A[i]$ is not polymorphic, but is parametric in the choice of A for demonstrative purposes.

The succ branch of $\mathsf{nat}[i]$ produces a nat at height $i-1$ *when* $i > 0$ whereas the tail branch of $\mathsf{stream}_A[i]$ can produce the remainder of the stream at depth $i-1$ *assuming* $i > 0$. Starting from $\mathsf{nat}[i]$, recurring *on*, for example, $\mathsf{nat}[i-1]$ ($i > 0$ 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 $\mathsf{stream}_A[i]$, recurring *into* $\mathsf{stream}_A[i-1]$ (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. Most importantly, the behavior of constraint conjunction and implication during elimination resp. introduction encodes induction resp. coinduction.

Let us make this precise—fix signatures of mutually recursive type and process definitions of the form $X[\bar{i}] = A_X(\bar{i})$ and $y \leftarrow f\ \bar{i}\ \overline{x} = P_f(y, \bar{i}, \overline{x})$. As a result, we add to the type and process syntax:

$$\begin{aligned}A, B, C &:= \ldots \mid X[\overline{e}]\\ P, Q &:= \ldots \mid y \leftarrow f\ \overline{e}\ \overline{x}\end{aligned}$$

Types are equirecursive and are unfolded silently; Das et al. [32] consider a richer judgmental equality on types that respects equality of type indices (arithmetic terms) which we leave for the next section on dependent RSAX. We also add an explicit operational rule unfolding recursive definitions:

$$\mathsf{proc}\, a\, (a \leftarrow f\ \overline{n}\ \overline{b}) \to \mathsf{proc}\, a\, (P_f(\overline{n}, \overline{b}, a))$$

To perform termination checking, we modify the process typing judgment to keep track a vector of arithmetic data $\overline{e}$. In general, vectors of objects will be denoted with an overline.

$$\Gamma \vdash^{\overline{e}} P :: (x : A)$$

The existing typing rules factor $\overline{e}$ through. Similar to Levy's [58] infinitely deep call-by-push-value, we take a mixed inductive-coinductive view of the syntax to model recursion. To check recursive calls, we add the "call" rule below where ∞J indicates a *coinductive* occurrence of the judgment J.

$$\frac{\Gamma \vdash \overline{e'} < \overline{e} \quad y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y) \quad \infty(\overline{x} : \overline{A} \vdash^{\overline{e'}} P_f(\overline{e'}, \overline{x}, y) :: (y : A))}{\Gamma, \overline{x} : \overline{A} \vdash^{\overline{e}} y \leftarrow f\ \overline{e'}\ \overline{x} :: (y : A)}\ \text{call}$$

Informally, all valid typing derivations are trees whose infinite branches have an instances of the call rule occur infinitely often, representing the unfolding of a recursive process. At each unfolding, we check that the arithmetic arguments have decreased from $\overline{e}$ to $\overline{e'}$ lexicographically[4] for termination. 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 [27].

As a result, sized type refinements are *compositional*: since termination checking is reduced to typechecking, we avoid the brittleness of syntactic termination checking. For typechecking in finite time, we conjecture that restricting our type system to circular derivations, which can be represented as finite trees with loops, and decidable arithmetic (e.g., Presburger) is sufficient. In short, such a restricted system can be put in correspondence with a finitary system that detects said loops [17, 26, 72] and arithmetic assertions can be discharged mechanically [30]. In the next example, we show a simple inductive process definition as well as a hypothetical instance of typechecking. **Note: in all examples, we suppress explicit process terms for constrained types, because assumptions and assertions can be inferred in these circumstances.** [33].

Example 3 (Typechecking)**.** The process definition below, whose type signature is $i, x : \text{nat}[i] \vdash^i y \leftarrow \text{eat}\ \ i\ x :: (y : \mathbf{1})$, traverses a unary natural number by induction to produce a unit. Recall $\text{nat}[i] = \oplus\{\text{zero} : \mathbf{1}, \text{succ}\ : i > 0 \wedge \text{nat}[i-1]\}$.

$$y \leftarrow \text{eat}\, i\ x = x\ \mathbf{match}\ \{\text{zero}\, z \mapsto z \to y, \text{succ}\, z \mapsto y \leftarrow \text{eat}\, (i-1)\ z\}$$

Now, let us construct a typing derivation of its body below. Note that we use "$D \in J$" to indicate a derivation D of the judgment J.

$$\text{D} = \frac{\dfrac{}{z : \mathbf{1} \vdash^i (y : \mathbf{1})}\ \text{id} \qquad \dfrac{\dfrac{i, i > 0 \vdash i - 1 < i [(i-1)/i][z/x]D \in \infty(i, z : \text{nat}[i-1] \vdash^{i-1} (y : \mathbf{1}))}{i, i > 0, z : \text{nat}[i-1] \vdash^i (y : \mathbf{1})}\ \text{call}}{i, z : i > 0 \wedge \text{nat}[i-1] \vdash^i (y : \mathbf{1})}\ \wedge\text{L}, \text{cut}}{i; x : \text{nat}[i] \vdash^i (y : \mathbf{1})}\ \oplus\text{L}$$

For space, we omit the process terms. Of importance is the instance of the call rule for the recursive call to eat: the check $i > 0 \vdash i - 1 < i$ (discharged automatically) verifies that the process terminates and the loop $[(i-1)/i][z/x]D$ "ties the knot" on the typechecking process. Mutually recursive programs, then, are checked by circular typing derivations that are mutually recursive *in the metatheory*.

[4]If two vectors have different lengths, then zeroes are appended to the shorter one.

Some prior work, which is based on sequential functional languages, encode recursion via various fixed point combinators [2, 75] that make both mixed inductive-coinductive programming [11] and the option of substructural typing difficult, the latter requiring the use of the ! modality [98][5]. Thus, like $\mathsf{F}_\omega^{\mathsf{cop}}$ [4], we consider a signature of parametric recursive definitions. Unlike *op. cit.* and like Levy's infinitely-deep syntax [58], we make typing derivations for recursive programs infinitely deep by unfolding recursive calls *ad infinitum* [17, 57]. This accrues the following benefits:

- An elegant presentation of typing that need not deal directly with the complexity of checking mutually recursive definitions
- As we will see in the next subsection, a two-stage termination argument that translates these infinitely deep derivations to infinitely wide but finitely deep derivations eligible for induction. As a result, the original typing judgment can utilize arbitrarily rich recursion schemes as long as they can be translated.
- Recalling our goal of future-proofing, the option of introducing substructurality in correspondence with infinite proofs in linear logic [37, 38, 8]

With our feet now wet, let us explore an example of mutual coinduction.

Example 4 (Evens and odds)**.** The signature below describes definitions that project the even- and odd-indexed substreams (referred to by y) of some input stream (referred to by x) at half of the original depth.

$$\begin{aligned} i;\cdot;x : \mathsf{stream}_A[2i] &\vdash^i y \leftarrow \mathsf{evens}\ i\ x :: (y : \mathsf{stream}_A[i]) \\ i;\cdot;x : \mathsf{stream}_A[2i+1] &\vdash^i y \leftarrow \mathsf{odds}\ i\ x :: (y : \mathsf{stream}_A[i]) \end{aligned}$$

The even-indexed substream retains the head of the input, but its tail is the odd-indexed substream of the input's tail. The odd-indexed substream, on the other hand, is simply the even-indexed substream of the input's tail. Operationally, the heads and tails of both substreams are computed on demand similar to a lazy record. Unlike their sequential counterparts, however, the recursive calls proceed concurrently due to the nature of cut. Since our examples will keep constraints implicit, we indicate when constraints are assumed or asserted inline for clarity.

$$\begin{aligned} y \leftarrow \mathsf{evens}\ i\ x = y\ \mathbf{match}\ \{\, &\mathsf{head}\ h \mapsto x^{\mathrm{R}}.\mathsf{head}\ h, \\ &\underbrace{\mathsf{tail}\ y_t}_{i>0\ \text{assumed}} \mapsto x_t \leftarrow \underbrace{x^{\mathrm{R}}.\mathsf{tail}\ x_t}_{2i>0\ \text{asserted}}\ ; \underbrace{y_t \leftarrow \mathsf{odds}\ (i-1)\ x_t\}}_{i;i>0\vdash i-1<i\ \text{checked}} \end{aligned}$$

[5]The Y combinator, for example, has the type $!(!A \multimap A) \multimap A$, excluding linear logic without exponentials.

$$y \leftarrow \text{odds } i\ x = x_t \leftarrow \underbrace{x^{\mathrm{R}}.\,\text{tail}\,x_t}_{2i+1>0 \text{ asserted}}; y \leftarrow \text{evens } i\ x_t$$

By inlining the definition of odds in evens and vice versa, both programs terminate even though odds calls evens with argument i. We sketch an alternate termination argument for similar such definitions at the end of the next subsection that does not require inlining. An alternate typing scheme that hides the exact size change is shown below—given a stream of *arbitrary* depth, we may project its even- and odd-indexed substreams of arbitrary depth, too.

$$i; \cdot; x : \forall j.\ \text{stream}_A[j] \vdash^i y \leftarrow \text{evens } i\ x :: (y : \text{stream}_A[i])$$
$$i; \cdot; x : \forall j.\ \text{stream}_A[j] \vdash^i y \leftarrow \text{odds } i\ x :: (y : \text{stream}_A[i])$$

$\exists j.\ X[j]$ and $\forall j.\ X[j]$ denote *full* inductive and coinductive types, respectively, classifying (co)data of arbitrary size. In general, less specific type signatures are necessary when the exact size change is difficult to express at the type level [102]. For example, in relation to an input list of height i, the height j of the output list from a list filtering function may be constrained as $j \leq i$.

First, we define head and tail observations on streams of arbitrary depth. Since they are not recursive, we do not bother tracking the size superscript of the typing judgment, since they can be inlined. Moreover, we take the liberty to nest values (highlighted yellow), which can be expanded into SAX [72].

$$\cdot; \cdot; x : \forall j.\ \text{stream}_A[j] \vdash y \leftarrow \text{head}\,x :: (y : A)$$
$$y \leftarrow \text{head}\,x = x^{\mathrm{R}}.\ \langle 0, \text{head}\,y \rangle$$
$$\cdot; \cdot; x : \forall j.\ \text{stream}_A[j] \vdash y \leftarrow \text{tail}\,x :: (y : \forall j.\ \text{stream}_A[j])$$
$$y \leftarrow \text{tail}\,x = y\ \textbf{match}\ (\langle j, y' \rangle \mapsto \underbrace{x^{\mathrm{R}}.\ \langle j+1, \text{tail}\,y' \rangle}_{j+1>0 \text{ asserted}})$$

The implementation of odds and evens follows almost exactly as before with the above observations in place. Note that we use the abbreviation $y \leftarrow f\ \overline{e}\ \overline{x}; Q \triangleq y \leftarrow (y \leftarrow f\ \overline{e}\ \overline{x}); Q$ for convenience.

$$\begin{aligned} y \leftarrow \text{evens } i\ x = y\ \textbf{match}\ \{&\text{head } h \mapsto y \leftarrow \text{head}\,x, \\ &\text{tail}\,y_t \mapsto x_t \leftarrow \text{tail}\,x; y_t \leftarrow \text{odds}\,(i-1)\ x_t\} \\ y \leftarrow \text{odds } i\ x = x_t \leftarrow \text{tail}\,x; y &\leftarrow \text{evens } i\ x_t \end{aligned}$$

Ad hoc features for implementing size arithmetic in the prior work are subsumed by our more general arithmetic refinements that combine the "good parts" of modern sized type systems.

- First, the instances of constraint conjunction and implication to encode inductive resp. coinductive types in our system are similar to the bounded quantifiers in MiniAgda [3], which gave an elegant foundation for mixed inductive-coinductive functional programming, avoiding continuity checking [2].
- Unlike the prior work, however, we are able to modulate the specificity of type signatures: (slight variations of) those in example 4 are respectively given in $\mathsf{CIC}_{\widehat{\ell}}$ [75] and MiniAgda [3, 1]. Furthermore, we avoid transfinite indices in favor of permitting some unbounded quantification (following Vezzosi [97]), achieving the effect of somewhat complicated infinite sizes without leaving finite arithmetic.

Of course, a major application of sized types is mixed inductive-coinductive programming, so consider the following examples that demonstrate use cases in concurrency.

Example 5 (Stream processors)**.** The mixed inductive-coinductive type $\mathrm{sp}_{A,B}[i,j]$ of stream processors of input depth i and output depth j represents continuous (in the sense of [43]) functions from $\mathrm{stream}_A[i]$ to $\mathrm{stream}_B[j]$; we define it below. Ghani et al. [43] define it as the nested greatest-then-least fixed point $\nu X.\, \mu Y.\, (A \times Y) + (B \times X)$, but we adapt the version by Abel [3] to finite size arithmetic.

$$
\begin{aligned}
\mathrm{sp}_{A,B}[i,j] &= \oplus\{\mathrm{get} : A \to \mathrm{sp}^{\mu}_{A,B}[i,j], \mathrm{put} : \&\{\mathrm{now} : B, \mathrm{rest} : \mathrm{sp}^{\nu}_{A,B}[i,j]\}\} \\
\mathrm{sp}^{\mu}_{A,B}[i,j] &= i > 0 \land \forall j'.\, \mathrm{sp}_{A,B}[i-1,j'] \\
\mathrm{sp}^{\nu}_{A,B}[i,j] &= j > 0 \supset \mathrm{sp}_{A,B}[i,j-1]
\end{aligned}
$$

Such functions may consume finitely many elements of type A from the input stream (the inductive part $\mathrm{sp}^{\mu}_{A,B}[i]$) before outputting arbitrarily many elements of type B onto the output stream (the coinductive part $\mathrm{sp}^{\nu}_{A,B}[j]$). This requires a lexicographic induction on (i,j)—in the inductive part, the input depth decreases to $i-1$, so the new output depth j' may be arbitrary. In the coinductive part, i stays the same, so j must decrease (to $j-1$). Let us interpret stream processors as functions on streams via a concurrent "run" function (as opposed to the sequential version from prior work [3, 4]). Below, we give the type signature and code.

$$
\begin{array}{l}
i, j, p : \mathsf{sp}_{A,B}[i,j], x : \mathsf{stream}_A[i] \vdash y \leftarrow \mathsf{run}\ (i,j)\ (p,x) :: (y : \mathsf{stream}_B[j]) \\
y \leftarrow \mathsf{run}\ (i,j)\ (p,x) = \\
\quad p\ \mathbf{match}\ \{\mathsf{get}\, f \mapsto h \leftarrow x \circ \mathsf{head}\ h; \underbrace{p' \leftarrow f \circ \langle h, \langle j, p' \rangle\rangle}_{i>0 \text{ assumed}}; \\
\qquad\qquad\qquad t \leftarrow \underbrace{x \circ \mathsf{tail}\, t}_{i>0 \text{ asserted}}; \underbrace{y \leftarrow \mathsf{run}(i-1,j)\ (p',t)}_{i,j,i>0 \Vdash (i-1,j)<(i,j) \text{ checked}}, \\
\qquad\quad \mathsf{put}\, o \mapsto y\ \mathbf{match}\ \{\mathsf{head}\ h \mapsto o \circ \mathsf{now}\ h, \\
\qquad\qquad\qquad \underbrace{\mathsf{tail}\, t}_{j>0 \text{ assumed}} \mapsto p' \leftarrow \underbrace{o \circ \mathsf{rest}\, p'}_{j>0 \text{ asserted}}; \underbrace{t \leftarrow \mathsf{run}\ (i,j-1)\ (p',x)}_{i,j,j>0 \Vdash (i,j-1)<(i,j) \text{ checked}}\}\}
\end{array}
$$

If the processor issues a "get," then the head of the input stream is consumed, recursing on its tail. Otherwise, the output stream is constructed recursively, first issuing the element received from the processor. It is clear that the program terminates by lexicographic induction on (i, j).

Example 6 (Left-fair streams)**.** Let us define the mixed inductive-coinductive type $\mathsf{lfair}_{A,B}[i,j]$ of left-fair streams [11]: infinite A-streams where each element is separated by finitely many elements in B. Once again, these types are *not* polymorphic, but are parametric in the choice of A and B for demonstration.

$$
\begin{aligned}
\mathsf{lfair}_{A,B}[i,j] &= \oplus\{\mathsf{now} : \&\{\mathsf{head} : A, \mathsf{tail} : \mathsf{lfair}^{\nu}_{A,B}[i,j]\}, \mathsf{later} : B \otimes \mathsf{lfair}^{\mu}_{A,B}[i,j]\} \\
\mathsf{lfair}^{\nu}_{A,B}[i,j] &= i > 0 \supset \exists j'.\ \mathsf{lfair}_{A,B}[i-1,j'] \\
\mathsf{lfair}^{\mu}_{A,B}[i,j] &= j > 0 \land \mathsf{lfair}_{A,B}[i,j-1]
\end{aligned}
$$

In particular, i bounds the observation depth of the A-stream whereas j bounds the height of the B-list in between consecutive A elements. Thus, this type is defined by lexicographic induction on (i, j). First, the provider may offer an element of A, in which case the observation depth of the stream decreases from i to $i-1$ (in the coinductive part, $\mathsf{lfair}^{\nu}_{A,B}[i,j]$). As a result, j may be "reset" as an arbitrary j'. On the other hand, if an element of "padding" in B is offered, then the depth i does not change. Rather, the height of the B-list decreases from j to $j-1$ (in the inductive part, $\mathsf{lfair}^{\mu}_{A,B}[i,j]$). By using left-fair streams, we can model processes that permit some timeout behavior but are eventually productive, since consecutive elements of type A are interspersed with only finitely many timeout acknowledgements of type B. Armed with this type, we can define a *projection* operation [11] that removes all of a left-fair stream's timeout acknowledgements concurrently, returning an A-stream. For brevity, we nest patterns (highlighted yellow), which can be expanded into nested matches [72].

$$
\begin{aligned}
&i, j, x : \mathsf{lfair}_{A,B}[i,j] \vdash^{(i,j)} y \leftarrow \mathsf{proj}\;(i,j)\;x :: (y : \mathsf{stream}_A[i])\\
&y \leftarrow \mathsf{proj}\;(i,j)\;x =\\
&x\;\mathbf{match}\;(\mathsf{now}\,s \mapsto \mathbf{match}\,y^{\mathrm{W}}(\mathsf{head}\,h \mapsto s^{\mathrm{R}}.\,\mathsf{head}\,h,\\
&\qquad\qquad\qquad\qquad \underbrace{\mathsf{tail}\,t}_{i>0\text{ assumed}} \mapsto u \leftarrow s^{\mathrm{R}}.\,\mathsf{tail}\,u;\\
&\qquad\qquad\qquad\qquad\qquad\quad u\;\mathbf{match}\;(\underbrace{\langle j', x'\rangle}_{i>0\text{ asserted}} \mapsto \underbrace{t \leftarrow \mathsf{proj}\;(i-1,j')\;x'}_{i,j,j';i>0\vdash(i-1,j')<(i,j)\text{ checked}})),\\
&\qquad\quad \underbrace{\mathsf{later}\langle b, x'\rangle}_{j>0\text{ assumed}} \mapsto \underbrace{y \leftarrow \mathsf{proj}\;(i,j-1)\;x'}_{i,j;j>0\vdash(i,j-1)<(i,j)\text{ checked}}\;)
\end{aligned}
$$

4.2 Termination

To prove termination of RSAX with respect to the futures-based dynamics reviewed in the previous section, we first survey the three proof-theoretic representations of recursion:

- Infinitely deep as well as circular proofs, which we have utilized, but typically require automata-based [42, 38], rewriting-based [8], or "semantic"/Henkin completeness-based cut elimination strategies [19] in contrast to logical relation arguments.
- (Co)induction rule schemata corresponding to fixed-point combinators, which we avoided for reasons mentioned at the beginning of the section. Moreover, they are actually less expressive than circular proofs in general [12].
- Infinitely wide but finitely deep proofs—for example, Schütte [79] proved cut elimination for Peano arithmetic by translation to a system that eliminates the induction rule and replaces the noninvertible quantifier rules with ω-rules that have a premise per each natural number (in lieu of introducing eigenvariables). The corresponding concept in the theory of programming languages—infinitely wide programs—was brought to the λ-calculus by Tait [86], Howard [49], and further developed by Martin-Löf [61].

The last point is crucial: if we can translate our infinite typing derivations to a system of infinitely wide but finitely deep proofs with analogous ω-rules, then we can retain our original inductive termination argument. Thus, we give a purely inductive process typing called ω process typing with the judgment $\Gamma \vdash^{\omega} P :: (x : A)$ (rules in figure 4). Constraining Γ to be free of arithmetic variables or constraint hypotheses, the rules $\exists\mathrm{L}^{\omega}$ and $\forall\mathrm{R}^{\omega}$ have one premise per natural number n instead of introducing a new arithmetic variable. Moreover, the premises of $\wedge\mathrm{L}^{\omega}$ and $\supset\mathrm{R}^{\omega}$ assume the closed constraint ϕ holds at the meta level instead of adding it to the context.

Most importantly, the call rule is not coinductive because, in the absence of free arithmetic variables, the arithmetic arguments decrease from some $\overline{n}$ to $\overline{n'}$ to etc. Since our chosen lexicographic order on natural number vectors is well-founded, this sequence necessarily terminates. To rephrase: the exact number of recursive calls is known. While this system is impractical for type checking, we can translate any derivation of the original process typing judgment to one of ω typing using the theorem below. Technically, we are restricting our attention to constructive instances of the ω-rules where each premise is proved uniformly using a computable procedure [103].

Theorem 4 (Translation)**.** Let Γ be free of arithmetic variables or constraint hypotheses. If $D \in \Gamma \vdash^{\overline{n}} P :: (x : A)$, then $\Gamma \vdash^{\omega} P :: (x : A)$.

Proof. By lexicographic induction on $(\overline{n}, D)$, we cover the important cases. We show the proof as a transformation $\rightsquigarrow$ on derivations with IH as the induction hypothesis.

- When D ends in the identity or an axiomatic rule, we are done by the corresponding ω rule.

$$D = \frac{}{\Gamma, x : A \vdash^{\overline{n}} x \to y :: (y : A)}\ \text{id} \rightsquigarrow \frac{}{\Gamma, x : A \vdash^{\omega} x \to y :: (y : A)}\ \text{id}^{\omega}$$

- When D ends in an invertible propositional rule with subderivation D', we proceed by induction on $(\overline{n}, D')$.

$$D = \frac{D' \in \Gamma, y : A \vdash P(y,z) :: (z : B)}{\Gamma \vdash x\ \mathbf{match}\ (\langle y,z\rangle \mapsto P(y,z)) :: (x : A \to B)}\ {\to}\text{R} \rightsquigarrow$$

$$\frac{IH(\overline{n}, D') \in \Gamma, y : A \vdash^{\omega} P(y,z) :: (z : B)}{\Gamma \vdash^{\omega} x\ \mathbf{match}\ (\langle y,z\rangle \mapsto P(y,z)) :: (x : A \to B)}\ {\to}\text{R}^{\omega}$$

- When D ends in $\exists$L or $\forall$R, its subderivation D' introduces a fresh arithmetic variable i. The m^{th} premise of the corresponding ω rules $\exists\text{L}^{\omega}$ and $\forall\text{R}^{\omega}$ are fulfilled by induction on $(\overline{n}, [m/i]D')$.

$$D = \frac{D' \in \Gamma, i \vdash^{\overline{n}} P(i,y) :: (y : A(i))}{\Gamma \vdash^{\overline{n}} x\ \mathbf{match}\ (\langle i,y\rangle \mapsto P(i,y)) :: (x : \forall i.\, A(i))}\ \forall\text{R} \rightsquigarrow$$

$$\frac{IH(\overline{n}, [m/i]D') \in \Gamma \vdash^{\omega} P(m,y) :: (y : A(m)) \text{ for all } m \in \mathbb{N}}{\Gamma \vdash^{\omega} x\ \mathbf{match}\ (\langle i,y\rangle \mapsto P(i,y)) :: (x : \forall i.\, A(i))}\ \forall\text{R}^{\omega}$$

- Analogously, when D ends in $\wedge$L or $\supset$R, its subderivation D' assumes $y \div \phi$. The premises of the corresponding ω rules $\wedge\text{L}^{\omega}$ and $\supset\text{R}^{\omega}$ assume $E \in \cdot \vdash \phi$, so we finish by induction on $(\overline{n}, E \cdot D')$ where $E \cdot D'$ cuts ϕ out of D'.

$$\frac{}{\Gamma, x : A \vdash^{\omega} x \to y :: (y : A)}\ \mathrm{id}^{\omega} \qquad \frac{\Gamma \vdash^{\omega} P(x) :: (x : A) \quad \Gamma, x : A \vdash^{\omega} Q(x) :: (z : C)}{\Gamma \vdash^{\omega} x \leftarrow P(x); Q(x) :: (z : C)}\ \mathrm{cut}^{\omega}$$

$$\frac{}{\Gamma \vdash^{\omega} x \circ \langle\rangle :: (x : \mathbf{1})}\ \mathbf{1}\mathrm{R}^{\omega} \qquad \frac{\Gamma, x : \mathbf{1} \vdash^{\omega} P :: (z : C)}{\Gamma, x : \mathbf{1} \vdash^{\omega} x\ \mathbf{match}\ (\langle\rangle \mapsto P) :: (z : C)}\ \mathbf{1}\mathrm{L}^{\omega}$$

$$\frac{}{\Gamma, y : A, z : B \vdash^{\omega} x \circ \langle y, z\rangle :: (x : A \otimes B)}\ \otimes\mathrm{R}^{\omega} \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \vdash^{\omega} P(y,z) :: (w : C)}{\Gamma, x : A \otimes B \vdash^{\omega} x\ \mathbf{match}\ (\langle y, z\rangle \mapsto P(y,z)) :: (w : C)}\ \otimes\mathrm{L}^{\omega}$$

$$\frac{\Gamma, y : A \vdash^{\omega} P(y,z) :: (z : B)}{\Gamma \vdash^{\omega} x\ \mathbf{match}\ (\langle y, z\rangle \mapsto P(y,z)) :: (x : A \to B)}\ {\to}\mathrm{R}^{\omega} \qquad \frac{}{\Gamma, x : A \to B, y : A \vdash^{\omega} x \circ \langle y, z\rangle :: (z : B)}\ {\to}\mathrm{L}^{\omega}$$

$$\frac{k \in S}{\Gamma, y : A_k \vdash^{\omega} x \circ k \cdot y :: (x : \oplus\{\ell : A_\ell\}_{\ell \in S})}\ \oplus\mathrm{R}^{\omega} \qquad \frac{\{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S}, y : A_k \vdash^{\omega} P_k(y) :: (z : C)\}_{k \in S}}{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S} \vdash^{\omega} x\ \mathbf{match}\ \{\ell \cdot y \mapsto P_\ell(y)\}_{\ell \in S} :: (z : C)}\ \oplus\mathrm{L}^{\omega}$$

$$\frac{\{\Gamma \vdash^{\omega} P(y) :: (y : A_k)\}_{k \in S}}{\Gamma \vdash^{\omega} x\ \mathbf{match}\ \{\ell \cdot y \mapsto P_\ell(y)\}_{\ell \in S} :: (x : \&\{\ell : A_\ell\}_{\ell \in S})}\ \&\mathrm{R}^{\omega} \qquad \frac{k \in S}{\Gamma, x : \&\{\ell : A_\ell\}_{\ell \in S} \vdash^{\omega} x \circ k \cdot y :: (y : A_k)}\ \&\mathrm{L}^{\omega}$$

$$\frac{}{\Gamma, y : A(n) \vdash^{\omega} x \circ \langle n, y\rangle :: (x : \exists i.\, A(n))}\ \exists\mathrm{R}^{\omega} \qquad \frac{\Gamma, x : \exists i.\, A(i), y : A(n) \vdash^{\omega} P(n,y) :: (z : C) \text{ for all } n \in \mathbb{N}}{\Gamma, x : \exists i.\, A(i) \vdash^{\omega} x\ \mathbf{match}\ (\langle i, y\rangle \mapsto P(i,y)) :: (z : C)}\ \exists\mathrm{L}^{\omega}$$

$$\frac{\Gamma \vdash^{\omega} P(n,y) :: (y : A(n)) \text{ for all } n \in \mathbb{N}}{\Gamma \vdash^{\omega} x\ \mathbf{match}\ (\langle i, y\rangle \mapsto P(i,y)) :: (x : \forall i.\, A(i))}\ \forall\mathrm{R}^{\omega} \qquad \frac{n \in \mathbb{N}}{\Gamma, x : \forall i.\, A(i) \vdash^{\omega} x \circ \langle n, y\rangle :: (y : A(n))}\ \forall\mathrm{L}^{\omega}$$

$$\frac{\cdot \vdash \phi}{\Gamma, y : A \vdash^{\omega} x \circ [y] :: (x : \phi \wedge A)}\ \wedge\mathrm{R}^{\omega} \qquad \frac{\Gamma, x : \phi \wedge A, y : A \vdash^{\omega} P(y) :: (z : C) \text{ if } \cdot \vdash \phi}{\Gamma, x : \phi \wedge A \vdash^{\omega} x\ \mathbf{match}\ ([y] \mapsto P(y)) :: (z : C)}\ \wedge\mathrm{L}^{\omega}$$

$$\frac{\Gamma \vdash^{\omega} P(y) :: (y : A) \text{ if } \cdot \vdash \phi}{\Gamma \vdash^{\omega} x\ \mathbf{match}\ ([y] \mapsto P(y)) :: (x : \phi \supset A)}\ {\supset}\mathrm{R}^{\omega} \qquad \frac{\cdot \vdash \phi}{\Gamma, x : \phi \supset A \vdash^{\omega} x \circ [y] :: (y : A)}\ {\supset}\mathrm{L}^{\omega}$$

$$\text{(no rule for } \mathbf{impossible}\text{)} \qquad \frac{y \leftarrow f\ \bar{i}\ \overline{x} = P_f(\bar{i}, \overline{x}, y) \quad \overline{x} : \overline{A} \vdash^{\omega} P_f(\overline{n}, \overline{x}, y) :: (y : A)}{\Gamma, \overline{x} : \overline{A} \vdash^{\omega} y \leftarrow f\ \overline{n}\ \overline{x} :: (y : A)}\ \mathrm{call}^{\omega}$$

Figure 4: ω Process Typing Rules

$$D = \dfrac{D' \in \Gamma, \phi \vdash^{\overline{n}} P(y) :: (y : A)}{\Gamma \vdash^{\overline{n}} x\, \mathbf{match}\, ([y] \mapsto P(y)) :: (x : \phi \supset A)} \supset\mathrm{R} \rightsquigarrow$$

$$\dfrac{IH(\overline{n}, E \cdot D') \in \Gamma \vdash^{\omega} P(y) :: (y : A) \text{ if } E \in \cdot \vdash \phi}{\Gamma \vdash^{\omega} x\, \mathbf{match}\, ([y] \mapsto P(y) :: (x : \phi \supset A)} \supset\mathrm{R}^{\omega}$$

- Finally, assume D ends in the call rule with subderivation D'. Although D' may be larger than D, we have some new arithmetic arguments $\overline{n'} < \overline{n}$. Thus, we are done by induction on $(\overline{n'}, D')$ then the ω call rule.

$$D = \dfrac{\cdot \vdash \overline{n'} < \overline{n} \quad D' \in \infty(\overline{x} : \overline{A} \vdash_{\infty}^{\overline{n'}} P_f(\overline{n'}, \overline{x}, y) :: (y : A))}{\Gamma, \overline{x} : \overline{A} \vdash^{\overline{n}} y \leftarrow f\, \overline{n'}\, \overline{x} :: (y : A)} \text{call} \rightsquigarrow$$

$$\dfrac{IH(\overline{n'}, D') \in \overline{x} : \overline{A} \vdash^{\omega} P_f(\overline{n}, \overline{x}, y) :: (y : A)}{\Gamma, \overline{x} : \overline{A} \vdash^{\omega} y \leftarrow f\, \overline{n}\, \overline{x} :: (y : A)} \text{call}^{\omega}$$

□

As a bonus, we can make the original process typing judgment arbitrarily rich to support more complex patterns of recursion. As long as derivations in that system can be translated to the ω system, the logical relations argument over ω typing that we detail shortly does not change. For example, consider the following additions.

- Multiple blocks: To support multiple blocks of definitions, we may simply impose the requirement that mutual recursion may not occur *across* blocks. In other words, the call graph *across* blocks is directed acyclic, imposing a well-founded order on definition names: $g < f$ iff f calls g. As a result, translation of the definition f may proceed by lexicographic induction on $(f, \overline{n}, D)$. For example, let f call g. If g is defined in a different block than f, then the arithmetic arguments it applies ($\overline{n}$) may increase. Otherwise, $\overline{n}$ must decrease, since g is "equal" to f (in this order).

- Mutual recursion with priorities: Definitions in a block can be ordered by *priority*: if $g < f$, then f can call g with arguments of the same size. In example 4, odds calls evens with arguments of the same size but evens calls odds with arguments of lesser size. As a result, evens < odds. If $<$ is well-founded (like in this example), then translation of f may proceed by lexicographic induction on $(\overline{n}, f, D)$.

We are finally equipped with the tools to extend our previous termination argument. First, we re-define configuration typing in terms of ω process typing in figure 5.

Then, we need to define the semantic versions of our new type formers.

Definition 10 (Semantic types II)**.** Assume $\mathscr{F}$ is a $\mathbb{N}$-indexed semantic type and that ϕ is closed.

- $\mathcal{F} \in [b : \exists \mathscr{F}] \triangleq \mathcal{F} = \mathcal{F}', !\mathrm{cell}\, b\, \langle n, a \rangle$ and $\mathcal{F}' \in [a : \mathscr{F}(n)]$.

$$\frac{\Gamma \vdash^{\omega} P :: (z : C)}{\Sigma \vdash \mathrm{proc}\, a\, ([\Sigma, c : C/\Gamma, z : C]P) :: (\Sigma, c : C)}\ \mathrm{proc}$$

$$\frac{\Gamma \vdash^{\omega} z \circ V :: (z : C)}{\Sigma \vdash \mathrm{!cell}\, c\, ([\Sigma, c : C/\Gamma, z : C]V) :: (\Sigma, c : C)}\ \mathrm{!cell}_V \qquad \frac{\Gamma \vdash^{\omega} z\, \mathbf{match}\, K :: (z : C)}{\Sigma \vdash \mathrm{!cell}\, a\, ([\Sigma, c : C/\Gamma, z : C]K) :: (\Sigma, c : C)}\ \mathrm{!cell}_K$$

$$\frac{}{\Sigma \vdash \cdot :: \Sigma}\ \mathrm{empty} \qquad \frac{\Sigma \vdash \mathcal{C} :: \Sigma' \quad \Sigma' \vdash \mathcal{C}' :: \Delta}{\Sigma \vdash \mathcal{C}, \mathcal{C}' :: \Delta}\ \mathrm{join}$$

Figure 5: ω Configuration Typing

- $\mathcal{F} \in [b : \forall \mathscr{F}] \triangleq \mathcal{F}, \mathrm{proc}\, a\, (b^{\mathrm{R}}.\langle n, a \rangle) \in [\![a : \mathscr{F}(n)]\!]$ for all $n \in \mathbb{N}$ where a is fresh.
- $\mathcal{F} \in [b : \phi \wedge \mathscr{A}] \triangleq$ where $\mathcal{F} = \mathcal{F}', \mathrm{!cell}\, b\, [a]$, $\cdot \vdash \phi$, and $\mathcal{F}' \in [a : \mathscr{A}]$.
- $\mathcal{F} \in [b : \phi \supset \mathscr{A}] \triangleq$ if $\cdot \vdash \phi$, then $\mathcal{F}, \mathrm{proc}\, a\, (b^{\mathrm{R}}.[a]) \in [\![a : \mathscr{A}]\!]$ where a is fresh.

In figure 6 are the corresponding semantic object typing rule lemmas.

$$\frac{}{\boldsymbol{\Sigma}, a : \mathscr{F}(n) \vDash \mathrm{!cell}\, b\, \langle n, a \rangle :: \boldsymbol{\Sigma}, a : \mathscr{F}(n), b : \exists \mathscr{F}}\ \exists\mathrm{R} \qquad \frac{\{\boldsymbol{\Sigma}, b : \exists \mathscr{F}, a : \mathscr{F}(n) \vDash \mathrm{proc}\, c\, (P(n, a)) :: \boldsymbol{\Delta}, c : \mathscr{C}\}_{n \in \mathbb{N}}}{\boldsymbol{\Sigma}, b : \exists \mathscr{F} \vDash \mathrm{proc}\, c\, (b\, \mathbf{match}\, (\langle i, x \rangle \mapsto P(i, x))) :: \boldsymbol{\Delta}, c : \mathscr{C}}\ \exists\mathrm{L}$$

$$\frac{\{\boldsymbol{\Sigma} \vDash \mathrm{proc}\, a\, (P(n, a)) :: \boldsymbol{\Delta}, a : \mathscr{F}(n)\}_{n \in \mathbb{N}}}{\boldsymbol{\Sigma} \vDash \mathrm{!cell}\, b\, (\langle i, x \rangle \mapsto P(i, x)) :: \boldsymbol{\Delta}, b : \forall \mathscr{F}}\ \forall\mathrm{R} \qquad \frac{}{\boldsymbol{\Sigma}, b : \forall \mathscr{F} \vDash \mathrm{proc}\, a\, (b \circ \langle n, a \rangle) :: \boldsymbol{\Sigma}, b : \forall \mathscr{F}, a : \mathscr{F}(n)}\ \forall\mathrm{L}$$

$$\frac{\cdot \vdash \phi}{\boldsymbol{\Sigma}, a : A \vDash \mathrm{!cell}\, b\, [a] :: \boldsymbol{\Sigma}, a : A, b : \phi \wedge \mathscr{A}}\ \wedge\mathrm{R} \qquad \frac{\boldsymbol{\Sigma}, b : \phi \wedge \mathscr{A}, a : \mathscr{A} \vDash \mathrm{proc}\, c\, (P(a)) :: \boldsymbol{\Delta}, c : \mathscr{C} \text{ if } \cdot \vdash \phi}{\boldsymbol{\Sigma}, b : \phi \wedge \mathscr{A} \vDash \mathrm{proc}\, c\, (b\, \mathbf{match}\, ([y] \mapsto P(y))) :: \boldsymbol{\Delta}, c : \mathscr{C}}\ \wedge\mathrm{L}$$

$$\frac{\boldsymbol{\Sigma} \vDash \mathrm{proc}\, a\, (P(a)) :: \boldsymbol{\Delta}, a : \mathscr{A} \text{ if } \cdot \vdash \phi}{\boldsymbol{\Sigma} \vDash \mathrm{!cell}\, b\, ([y] \mapsto P(y)) :: \boldsymbol{\Delta}, b : \phi \supset \mathscr{A}}\ \supset\mathrm{R} \qquad \frac{\cdot \vdash \phi}{\boldsymbol{\Sigma}, b : \phi \supset \mathscr{A} \vDash \mathrm{proc}\, a\, (b \circ [a]) :: \boldsymbol{\Sigma}, b : \phi \supset \mathscr{A}, a : \mathscr{A}}\ \supset\mathrm{L}$$

$$\text{(no rule for } \textbf{impossible}\text{)} \qquad \frac{y \leftarrow f\, \bar{i}\, \bar{x} = P_f(\bar{i}, \bar{x}, y) \quad \bar{b} : \overline{\mathscr{A}} \vDash \mathrm{proc}\, a\, (P_f(\bar{n}, \bar{b}, a)) :: \boldsymbol{\Delta}, a : \mathscr{A}}{\boldsymbol{\Sigma}, \bar{b} : \overline{\mathscr{A}} \vDash \mathrm{proc}\, a\, (a \leftarrow f\, \bar{n}\, \bar{b}) :: \boldsymbol{\Delta}, a : \mathscr{A}}\ \mathrm{call}$$

Figure 6: Semantic Object Typing Rule Lemmas II

To redevelop the fundamental theorem, we need to redefine the semantic interpretation of types to take into account the new type formers as well as recursive types.

Definition 11 (Semantic interpretation II)**.** We define $(\![A]\!)_n$ by lexicographic induction on (n, A) that types free of arithmetic variables to semantic ones. At a recursive type, n

is stepped down to allow A to potentially grow larger. Note that λ marks a meta-level anonymous function and that ϕ is closed.

$$\begin{array}{lll}
\llparenthesis \mathbf{1} \rrparenthesis_n \triangleq \mathbb{1} & \llparenthesis A \otimes B \rrparenthesis_n \triangleq \llparenthesis A \rrparenthesis_n \boldsymbol{\otimes} \llparenthesis B \rrparenthesis_n & \llparenthesis A \multimap B \rrparenthesis_n \triangleq \llparenthesis A \rrparenthesis_n \boldsymbol{\multimap} \llparenthesis B \rrparenthesis_n \\
 & \llparenthesis \oplus\{\ell : A_\ell\}_{\ell \in S} \rrparenthesis_n \triangleq \boldsymbol{\oplus}\{\ell : \llparenthesis A_\ell \rrparenthesis_n\}_{\ell \in S} & \llparenthesis \& \{\ell : A_\ell\}_{\ell \in S} \rrparenthesis_n \triangleq \boldsymbol{\&}\{\ell : \llparenthesis A_\ell \rrparenthesis_n\}_{\ell \in S} \\
\llparenthesis X[\overline{m}] \rrparenthesis_0 \triangleq \emptyset & \llparenthesis \forall i.\, A(i) \rrparenthesis_n \triangleq \boldsymbol{\forall}(\lambda m.\, \llparenthesis A(m) \rrparenthesis_n) & \llparenthesis \exists i.\, A(i) \rrparenthesis_n \triangleq \boldsymbol{\exists}(\lambda m.\, \llparenthesis A(m) \rrparenthesis_n) \\
\llparenthesis X[\overline{m}] \rrparenthesis_{n+1} \triangleq \llparenthesis A_X[\overline{m}] \rrparenthesis_n & \llparenthesis \phi \wedge A \rrparenthesis_n \triangleq \phi \boldsymbol{\wedge} \llparenthesis A \rrparenthesis_n & \llparenthesis \phi \supset A \rrparenthesis_n \triangleq \phi \boldsymbol{\supset} \llparenthesis A \rrparenthesis_n
\end{array}$$

The index n is merely a technical device for defining the semantic interpretation—it is *not* a step index. Now, let $\mathcal{F} \in \llparenthesis A \rrparenthesis \triangleq \mathcal{F} \in \llparenthesis A \rrparenthesis_n$ for some n. $\llparenthesis \cdot \rrparenthesis$ is then extended to contexts Γ and Δ in the obvious way.

As before, the semantic interpretation induces a map from syntactic typing derivations to semantic ones which is developed exactly as before.

Lemma 6 (Semantic object typing II)**.**

1. If $\Gamma \vdash^\omega z \circ V :: (z : C)$, then $\llparenthesis \Sigma \rrparenthesis \vDash \,!\mathrm{cell}\, a\, ([\Sigma, c : C/\Gamma, z : C]V) :: \llparenthesis \Sigma \rrparenthesis, c : \llparenthesis C \rrparenthesis$.
2. If $\Gamma \vdash^\omega z$ **match** $K :: (z : C)$, then $\llparenthesis \Sigma \rrparenthesis \vDash \,!\mathrm{cell}\, a\, ([\Sigma, c : C/\Gamma, z : C]K) :: \llparenthesis \Sigma \rrparenthesis, c : \llparenthesis C \rrparenthesis$.
3. If $\Gamma \vdash^\omega P :: (z : C)$, then $\llparenthesis \Sigma \rrparenthesis \vDash \mathrm{proc}\, a\, ([\Sigma, c : C/\Gamma, z : C]P) :: \llparenthesis \Sigma \rrparenthesis, c : \llparenthesis C \rrparenthesis$.

The rest of the termination argument follows as-is, except that the diamond property must also consider the unfolding of recursive definitions.

4.3 Related Work

RSAX is closely related to the sequential functional language of Lepigre et al. [57], which utilizes circular typing derivations for a sized type system with mixed inductive-coinductive types, also avoiding continuity checking. In particular, their well-foundedness criterion on circular proofs seems to correspond to our checking that sizes decrease between recursive calls. However, they encode recursion using a fixed point combinator and use transfinite size arithmetic, both of which we avoid as we explained earlier. Moreover, our metatheory, which handles *infinite* typing derivations (via mixed induction-coinduction at the meta level), seems to be both simpler and more general since it does not have to explicitly rule out non-circular derivations. Nevertheless, we are interested in how their innovations in polymorphism and Curry-style subtyping can be integrated into RSAX, especially the ability to handle programs not annotated with sizes.

Sized types. Sized types are a type-oriented formulation of size-change termination [53] for rewrite systems [90, 15]. Sized (co)inductive types [10, 13, 2, 4] gave way to sized mixed inductive-coinductive types [3, 4]. In parallel, linear size arithmetic for sized inductive types [24, 101, 14] was generalized to support coinductive types as well [75]. We present, to our knowledge, the first sized type system for a concurrent programming language as well as the first system to combine both features from above. As we mentioned earlier, we use unbounded quantification [97] in lieu of transfinite sizes to represent (co)data of arbitrary height and depth. However, the state of the art [3, 4, 23] supports polymorphic and higher-kinded types, which is part of the future (not proposed) work.

Size inference. RSAX keeps constraints implicit but arithmetic data explicit at the process level in agreement with observations made about constraint resp. full reconstruction in a session-typed calculus [33]. On the other hand, systems like $\mathsf{CIC}^{\widehat{}}_{\ell}$ [75] and $\mathsf{CIC}^{\widehat{*}}$ [23] have comprehensive 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 arithmetic refinements, we do not consider size inference.

Infinite and circular proofs. Validity conditions of infinite proofs have been developed to keep cut elimination productive, which correspond to criteria like the guardedness check [37, 38, 8]. 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 [18, 87], although it is unclear how they could generalize to concurrent programs (in the setting of concurrent separation logic) as well as codata.

Session types. Session types are inextricably linked with SAX, as it also has an asynchronous message passing interpretation [71]. Severi et al. [80] give a mixed functional and concurrent programming language where corecursive definitions are typed with Nakano's later modality [64]. Since Vezzosi [97] gives an embedding of the later modality and its dual into sized types, we believe that a similar arrangement can be achieved in RSAX. In any case, RSAX supports recursion schemes more complex than structural (co)recursion [59]. Relatedly, Derakhshan et al. [36] give a information flow control session type system that generalizes to one that is sound even in the presence of non-termination.

π-calculi. Certain type systems for π-calculi [52, 66, 44] guarantee the eventual success of communication only if or regardless of whether processes diverge [28]. Considering a configuration $\mathcal{C}$ such that $\Gamma \vdash \mathcal{C} :: (\Gamma, a : X[n])$ where $X[i]$ is a positive coinductive type, we conjecture that $|\mathcal{C}|$, which has all constraint and arithmetic data erased, is similarly "productive" even if it may *not* terminate. Intuitively, $\mathcal{C}$ writes a number of cells as a function of n then terminates, so $|\mathcal{C}|$ represents $\mathcal{C}$ in the limit since $X[i]$ is positive coinductive. However, this behavior is more desirable in a message passing setting rather than in our shared memory setting.

On the other hand, there are type systems that themselves guarantee termination—some assign numeric levels to each channel name and restrict communication such that a measure induced by said levels decreases consistently [35, 34, 25]. While message passing is a different setting than ours, we are interested in the relationship between sizes and levels, if any. Other such type systems constrain the type and/or term structure; the language $\mathscr{P}$ [76] requires grammatical restrictions on both types and terms, the latter of which we are trying to avoid. On the other hand, the combination of linearity and a certain acyclicity condition [105] on graph types [104] is also sufficient. RSAX is able to guarantee termination despite utilizing non-linear types, but it remains open how type refinements compare to graph types.

5 Proposed Work: Dependent RSAX (DRSAX)

In this section, we tentatively generalize RSAX to Dependent RSAX (DRSAX) in an attempt to close the gap between processes and proofs. RSAX did so for arithmetic data via quantification over arithmetic inputs and outputs as well as constraints to relate them together. Likewise, DRSAX will need the ability to quantify over process inputs and outputs—addresses—and constraints to relate them together. We do this in two steps:

1. Analogous to arithmetic quantifiers, to RSAX we modify the function and eager product types to be dependent, i.e., to quantify over process inputs and outputs. Furthermore, recursive type definitions may bind address variables.

$$\begin{aligned} A^-, B^- &:= \ldots \mid (x : A) \to B(x) \\ A^+, B^+ &:= \ldots \mid (x : A) \otimes B(x) \\ A, B, C &:= \ldots \mid X[\overline{e} \mid \overline{x}] \end{aligned}$$

1. We extend the refinement theory with quantification over and equality of sorted terms, bringing us to many-sorted first-order logic where we conflate address variables and (D)(R)SAX values with term variables and terms, respectively. However, addresses are only meaningful with respect to the processes that write to them. As a result, we add the constraint $x \leftarrow P(x); \psi(x)$ corresponding to modal necessity in dynamic logic, expanding to the weakest precondition of the assignment of x by P with respect to ϕ. That is, $x \leftarrow P(x); \psi(x) \equiv \forall x. \phi(x) \supset \psi(x)$ for some ϕ computed, *en passant*, during type checking. Thus, our process typing judgment becomes $x : A, \ldots, y : B \vdash^{\overline{e}} P :: (z : C) \mid \phi(x, \ldots, y, z)$.

The combination of these features admits relations between process inputs and outputs. As a case study, we define and use observational equality [5, 11], a heterogenous[6] equality constraint $\phi, \psi := \dots \mid (x : A) = (y : B)$ that, unlike Martin-Lof's identity type and some other type-theoretic treatments of equality, admits extensionality principles for negative types. This is achieved by *fiat*—by defining the constraint by recursion on A and B. We show representative cases below posed as axioms in the refinement theory.

For example, two functions are equal when their applications are equal at equal arguments. Likewise, two lazy records are equal when they are equal at all projections. For brevity below, we omit sort annotations for quantifiers.

- $(f : (x : A) \to B(x)) = (g : (y : A') \to B'(y)) \Leftrightarrow$
  $\forall x. \forall y. (x : A) = (y : A') \Rightarrow$
  $(z : B(x)) \leftarrow f \circ \langle x, z \rangle; (w : B(y)) \leftarrow g \circ \langle y, w \rangle; (z : B(x)) = (w : B(y))$

- $(r : \&\{\ell : A_\ell\}_{\ell \in S}) = (s : \&\{\ell : B_\ell\}_{\ell \in S}) \Leftrightarrow$
  $\bigwedge_{\ell \in S} (x : A_\ell) \leftarrow r \circ \ell \cdot x; (y : B_\ell) \leftarrow s \circ \ell \cdot y; (x : A_\ell) = (y : B_\ell)$

In general, two objects of negative types are equal when they coincide at all eliminations. On the other hand, two objects of positive type are equal when they have the same structure. As a result, equal constituents imply equal eager pairs and sums.

- $(\langle x, z \rangle : (x : A) \otimes B(x)) = (\langle y, w \rangle : (y : A') \otimes B'(y)) \Leftrightarrow$
  $(x : A) = (y : A') \wedge (z : B(x)) = (w : B'(y))$

- $(k \cdot x : \oplus\{\ell : A_\ell\}_{\ell \in S}) = (k \cdot y : \oplus\{\ell : B_\ell\}_{\ell \in S}) \Leftrightarrow$
  $(x : A_\ell) = (y : B_\ell)$ if $k \in S$

- $(k \cdot x : \oplus\{\ell : A_\ell\}_{\ell \in S}) = (k' y : \oplus\{\ell : B_\ell\}_{\ell \in S}) \Leftrightarrow \bot$ if $k, k' \in S$ and $k \neq k'$

Like Altenkirch et al. [5], we want to embed the judgmental equality into the observational equality ("reflexivity") and have an indiscernibility of identicals reasoning principle ("substitution"). We pose these as axiom schemas in the refinement theory, as follows. Note that we write $x = y$ for $(x : A) = (y : A)$ when A is unambiguous.

- Reflexivity: $x \equiv y : A \Rightarrow (x : A) = (y : A)$

- Substitution: $x = y \Rightarrow \phi(x) \Leftrightarrow \phi(y)$

Note: unlike [5], we leave a type-level equality constraint and coercions along said equalities to future work. Now that we have settled the high-level design of the type theory, the next few subsections review core typing rules and examples.

[6]Heterogeneity makes stating equalities in the presence of dependency easier, i.e., when A and B are equal but not judgmentally.

5.1 Typing

In this subsection, we give a representative sample of process typing rules. In line with [96], we refer to the component of the weakest precondition computed during type checking as the process's "reflection."

Identity and Cut. The identity rule states that the reflection of the identity process between y and x is $y \equiv x$—we do not annotate the type of a judgmental equality when it is obvious from context.

$$\frac{}{\Gamma, x : A \vdash x \to y :: (y : A) \mid y \equiv x}\ \text{id}$$

The cut rule is more subtle: in $x \leftarrow P(x); Q(x)$, $Q(x)$, reflected by $\psi(x)$, may use the information $\phi(x)$ reflected by $P(x)$, resulting in $\exists x.\, \psi(x)$. Nested quantifiers in reflected constraints poses a hurdle for decidable typechecking (relative to decidability of the refinement theory), which we discuss in the further proposed work.

$$\frac{\Gamma \vdash P(x) :: (x : A) \mid \phi(x) \quad \Gamma, x : A, \phi(x) \vdash Q(x) :: (z : C) \mid \psi(x)}{\Gamma \vdash x \leftarrow P(x); Q(x) :: (z : C) \mid \exists x.\, \psi(x)}\ \text{cut}$$

Positive Conjunction and Implication. Similar to the identity rule, positive conjunction's right rule reflects the identity of z as a pair of address variables x and y.

$$\frac{}{\Gamma, x : A, y : B(x) \vdash z \circ \langle x, y \rangle :: (z : (x : A) \otimes B(x)) \mid z \equiv \langle x, y \rangle}\ \otimes\text{R}$$

As the sequent calculus counterpart of a dependent elimination principle, $\otimes$L reveals the identity of the eliminated address variable as a pair by adding a constraint antecedent to the context. The remaining left rules for positive types do the same.

$$\frac{\Gamma, z : (x : A) \otimes B(x), x : A, y : B(x), z \equiv \langle x, y \rangle \vdash P(x, y) :: (w : C) \mid \phi(x, y)}{\Gamma, z : (x : A) \otimes B(x) \vdash z\ \textbf{match}\ (\langle x, y \rangle \mapsto P(x, y)) :: (w : C) \mid \forall x, y.\, z \equiv \langle x, y \rangle \Rightarrow \phi(x, y)}\ \otimes\text{L}$$

The right rule for implication is dual to the left rule for positive conjunction: the reflection of a function captures the behavior of its body given an arbitrary function call.

$$\frac{\Gamma, x : A \vdash P(x, y) :: (y : B(x)) \mid \phi(x, y)}{\Gamma \vdash z\ \textbf{match}\ (\langle x, y \rangle \mapsto P(x, y)) :: (z : (x : A) \to B(x)) \mid \forall x, y.\, y \equiv z(x) \Rightarrow \phi(x, y)}\ \to\text{R}$$

Likewise, the left rule for implication indicates that y contains the result of a function call.

$$\frac{}{\Gamma, z : (x : A) \to B(x), x : A \vdash z \circ \langle x, y \rangle :: (y : B(x)) \mid y \equiv z(x)}\ \to\text{L}$$

Recursion Recursive definitions are represented as functions in the refinement theory that are constrained by their bodies. Reflecting a recursive definition into a refinement theory is risky, because one cannot typically declare that it is a least, greatest, or nested fixed point [9]. In our case, however, the recursion scheme instance used by a DRSAX definition is determined by its sized type refinement.

$$\frac{\Gamma \vdash \overline{e'} < \overline{e} \quad y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y) \quad \infty(\overline{x} : \overline{A} \vdash^{\overline{e'}} P_f(\overline{e'}, \overline{x}, y) :: (y : A) \mid \phi(\overline{e'}, \overline{x}, y))}{\Gamma, \overline{x} : \overline{A} \vdash^{\overline{e}} y \leftarrow f\ \overline{e}'\ \overline{x} :: (y : A) \mid y \equiv f(\overline{e}', \overline{x})}\ \text{call}$$

To connect definitions to their bodies, we add the following axiom schemes to the refinement theory.

- Process definition: $y \equiv f(\overline{i}, \overline{x}) \Leftrightarrow \phi(\overline{i}, \overline{x}, y)$ (where ϕ is from above)

5.2 Examples

We can now prove within DRSAX properties about the programs we have defined—we will go through examples of induction and coinduction individually as well as mixed induction-coinduction.

Example 7 ($x + 0 = x$)**.** We will start with a classic proof by induction: for a natural number x, $x + 0 = x$. First, we need to define addition on the natural numbers. To that end, it is useful to further refine the type of natural numbers to indicate exact height: $\mathsf{nat}[i] = \oplus\{\mathsf{zero} : i = 0 \wedge \mathbf{1}, \mathsf{succ} : i > 0 \wedge \mathsf{nat}[i-1]\}$.

$$\begin{array}{l}
i, j, x : \mathsf{nat}[i], y : \mathsf{nat}[j] \vdash^{(i,j)} z \leftarrow \mathsf{add}\ (i,j)\ (x,y) :: (z : \mathsf{nat}[i+j]) \\
z \leftarrow \mathsf{add}\ (i,j)\ (x,y) = \\
\quad x\ \mathbf{match}\ \{\mathsf{zero}\ x' \mapsto y \rightarrow z, \\
\qquad \underbrace{\mathsf{succ}\ x'}_{i>0 \text{ assumed}} \mapsto z' \leftarrow \underbrace{\mathsf{add}(i-1,j)\ (x',y)}_{i,j,i>0 \vdash (i-1,j)<(i,j) \text{ checked}}\ ; z \circ \mathsf{succ}\ z'\}
\end{array}$$

Now, we introduce some notation: $\{\phi\} \triangleq \phi \wedge \mathbf{1}$ and $\boxed{x} \triangleq x \circ [\langle\rangle]$ (nesting values once again). In the proof below, notice our dual use of type refinements: the goal is a constrained type *and* sized types are used to ensure soundness of the induction.

$$\begin{array}{l}
i, x : \mathsf{nat}[i] \vdash^{i} p \leftarrow \mathsf{runit}\ i\ x :: (p : \{z \leftarrow z \circ \mathsf{zero}\langle\rangle ; y \leftarrow \mathsf{add}\ (i,0)\ (x,z); y = x\}) \\
p \leftarrow \mathsf{runit}\ i\ x = x\ \mathbf{match}\ \{\mathsf{zero}\ x' \mapsto \underbrace{\boxed{p}}_{y \equiv z \Rightarrow y = x \text{ asserted since } x = z}, \mathsf{succ}\ x' \mapsto \underbrace{p' \leftarrow \mathsf{runit}\ (i-1)\ x'}_{y' = x' \text{ assumed}}; \quad \underbrace{\boxed{p}}_{y' = x' \Leftrightarrow \mathsf{succ}\ y' = \mathsf{succ}\ x' \Leftrightarrow y = x \text{ asserted}}
\end{array}$$

Example 8 (Bisimulation)**.** As we noted before, observational equality shines in its treatment of negative types. In this example, we will show that two ways of constructing an infinite streams of ones coincide. In fact, the extensionality principle endowed to recursive record types allows us to argue for equality by constructing a <u>bisimulation</u>. Let us define the two streams: one is direct, the other maps the successor function over a stream of zeroes. First, we let $\mathrm{nat} = \exists i.\ \mathrm{nat}[i]$ since the sizes of the stream elements are irrelevant to termination in this case. The following process definition is direct, recalling $\mathrm{stream}_A[i] = \&\{\mathrm{head} : A, \mathrm{tail} : i > 0 \supset \mathrm{stream}_A[i-1]\}$.

$$i \vdash^i y \leftarrow \mathrm{ones}\ i :: (y : \mathrm{stream}_{\mathrm{nat}}[i])$$
$$y \leftarrow \mathrm{ones}\ i = y\ \mathbf{match}\ \{\mathrm{head}\, h \mapsto h \circ \langle 1, \mathrm{succ}\,(\mathrm{zero}\langle\rangle)\rangle;\ \underbrace{\mathrm{tail}\, t}_{i>0\text{ assumed}} \mapsto t \leftarrow \underbrace{\mathrm{ones}\,(i-1)}_{i, i>0 \vdash i-1<i \text{ checked}}\ \}$$

On the other hand, the following defines map, a stream of zeroes, and the stream of ones resulting from mapping the successor function over the stream of zeroes.

$$i, f : A \to B, x : \mathrm{stream}_A[i] \vdash^i y \leftarrow \mathrm{map}\ i\ (f, x) :: (y : \mathrm{stream}_A[i])$$
$$y \leftarrow \mathrm{map}\ i\ (f, x) = y\ \mathbf{match}\ \{\mathrm{head}\, h \mapsto h' \leftarrow x \circ \mathrm{head}\, h'; f \circ \langle h', h\rangle,$$
$$\mathrm{tail}\, t' \leftarrow x \circ \mathrm{head}\, t'; t \leftarrow \mathrm{map}\,(i-1)\ (f, t')\}$$
$$i \vdash^i y \leftarrow \mathrm{zeroes}\ i :: (y : \mathrm{stream}_{\mathrm{nat}}[i])$$
$$y \leftarrow \mathrm{zeroes}\ i = y\ \mathbf{match}\ \{\mathrm{head}\, h \mapsto h \circ \langle 0, \mathrm{zero}\langle\rangle\rangle; \mathrm{tail}\, t \mapsto t \leftarrow \mathrm{zeroes}\,(i-1)\}$$
$$i \vdash^i y \leftarrow \mathrm{ones}'\ i :: (y : \mathrm{stream}_{\mathrm{nat}}[i])$$
$$y \leftarrow \mathrm{ones}'\ i = f \leftarrow f\ \mathbf{match}\ (\langle\langle i, x\rangle, z\rangle \mapsto z \circ \langle i+1, \mathrm{succ}\, x\rangle); z \leftarrow \mathrm{zeroes}\, i; y \leftarrow \mathrm{map}\ i\ (f, z)$$

Finally, we can prove that both streams are equal. First, we introduce a type synonym for the proof goal:

$$\mathrm{bisim\_t}[i] = \{o \leftarrow \mathrm{ones}\ i; o' \leftarrow \mathrm{ones}'\ i; o = o'\}$$

To develop our proof strategy, we expand $o = o'$ below.

$$o = o' \Leftrightarrow (h \leftarrow o \circ \mathrm{head}\, h; h' \leftarrow o' \circ \mathrm{head}\, h'; h = h') \wedge (i > 0 \Rightarrow t \leftarrow o \circ \mathrm{tail}\, t; t' \leftarrow o' \circ \mathrm{tail}\, t'; t = t')$$

It's immediate that the heads are (judgmentally, therefore observationally) equal, so it suffices to recurse on the tails, constructing a bisimulation. Furthermore, the argument i decreases, corresponding to soundness of coinduction.

$$i \vdash^i p \leftarrow \mathrm{bisim}\ i :: (p : \mathrm{bisim\_t}[i-1])$$
$$p \leftarrow \mathrm{bisim}\ i = (p' : i > 0 \supset \mathrm{bisim\_t}[i-1]) \leftarrow \mathrm{bisim}\,(i-1)\underbrace{\boxed{; p}}_{\text{note } t \equiv \mathrm{ones}\,(i-1) \text{ and } t' \equiv \mathrm{ones}'\,(i-1)}$$

Above, we have used a trick introduced by Leino and Moskal [56] in the context of coinduction in Dafny: we cut in the recursive call *assuming* $i > 0$ by annotating the cut formula (type).

Example 9 (Idempotence of projection)**.** In our final example, we will prove an idempotence property of a mixed inductive-coinductive program: projection of left-fair streams. This is technically quite challenging, as we must reason about equality using mixed induction-coinduction. Recall that projection has the signature $i, j, x : \mathsf{lfair}_{A,B}[i, j] \vdash^{(i,j)} y \leftarrow \mathsf{proj}\ (i, j)\ x :: (y : \mathsf{stream}_A[i])$, where:

$$\begin{aligned}
\mathsf{lfair}_{A,B}[i,j] &= \oplus\{\mathsf{now} : \&\{\mathsf{head} : A, \mathsf{tail} : \mathsf{lfair}^{\nu}_{A,B}[i,j]\}, \mathsf{later} : B \otimes \mathsf{lfair}^{\mu}_{A,B}[i,j]\} \\
\mathsf{lfair}^{\nu}_{A,B}[i,j] &= i > 0 \supset \exists j'.\ \mathsf{lfair}_{A,B}[i-1, j'] \\
\mathsf{lfair}^{\mu}_{A,B}[i,j] &= j > 0 \land \mathsf{lfair}_{A,B}[i, j-1]
\end{aligned}$$

To apply projection twice, we need a process definition (injection) that reflects streams back into left-fair streams.

$$\begin{aligned}
&i, x : \mathsf{stream}_A[i] \vdash^{i} y \leftarrow \mathsf{inj}\ i\ x :: (y : \mathsf{lfair}_{A,B}[i, 0]) \\
&y \leftarrow \mathsf{inj}\ i\ x = \\
&\quad y' \leftarrow y'\ \mathbf{match}\ \{\mathsf{head}\, h \mapsto x \circ \mathsf{head}\, h, \\
&\qquad\qquad \underbrace{\mathsf{tail}\, t}_{i>0 \text{ assumed}} \mapsto s \leftarrow x \circ \mathsf{tail}\, s; t' \leftarrow \underbrace{\mathsf{inj}\ (i-1)\ s}_{i, i>0 \vdash i-1<i \text{ checked}}; t \circ \langle j, t' \rangle\}; \\
&\quad y \circ \mathsf{now}\, y'
\end{aligned}$$

We set up the roundtrip of projection-injection-projection as our goal.

$$\mathsf{idem\_t}[i, j \mid x] = \{y \leftarrow \mathsf{proj}\ (i, j)\ x; x' \leftarrow \mathsf{inj}\ i\ y; y' \leftarrow \mathsf{proj}\ (i, 0)\ x'; y' = y\}$$

Recall that $y = y'$ expands to the following.

$$y = y' \Leftrightarrow (h \leftarrow y \circ \mathsf{head}\, h; h' \leftarrow y' \circ \mathsf{head}\, h'; h = h') \land (i > 0 \Rightarrow t \leftarrow y \circ \mathsf{tail}\, t; t' \leftarrow t' \circ \mathsf{tail}\, t'; t = t')$$

With some aggressive variable substitution, let us take a look at our proof obligations. Below, the notation $.\ell$ indicates projection of a record in the refinement term theory.

- $\mathsf{proj}(i, 0, \mathsf{inj}(i, \mathsf{proj}(i, j, x))).\mathsf{head} = \mathsf{proj}(i, j, x).\mathsf{head}$
- $i > 0 \Rightarrow \mathsf{proj}(i, 0, \mathsf{inj}(i, \mathsf{proj}(i, j, x))).\mathsf{tail} = \mathsf{proj}(i, j, x).\mathsf{tail}$

The first is trivial, the second requires case analysis on x.

- $i > 0 \Rightarrow \mathrm{proj}(i, 0, \mathrm{inj}(i, \mathrm{proj}(i, j, \mathrm{now}(x')))).\mathrm{tail} = \mathrm{proj}(i, j, \mathrm{now}(x')).\mathrm{tail}$
- $j > 0 \Rightarrow \mathrm{proj}(i, 0, \mathrm{inj}(i, \mathrm{proj}(i, j, \mathrm{pad}(\langle b, x' \rangle)))).\mathrm{tail} = \mathrm{proj}(i, j, \mathrm{pad}(\langle b, x' \rangle)).\mathrm{tail}$

In the first sub-case, the assumption $i > 0$ gives us access to the tail of the left-fair stream. In the second sub-case, the assumption $j > 0$ supersedes that of i. Both sub-cases are then established by recursion, as follows.

$$
\begin{aligned}
&i, j, x : \mathrm{lfair}_{A,B}[i, j] \vdash^{(i,j)} p \leftarrow \mathrm{idem}\,(i, j)\ x :: (p : \mathrm{idem}\_t[i, j \mid x]) \\
&p \leftarrow \mathrm{idem}\,(i, j)\ x = \\
&\qquad x\ \mathbf{match}\ \{\mathrm{now}\, s \mapsto s' \leftarrow s \circ \mathrm{tail}\, s'; s'\ \mathbf{match}\ (\langle j', x' \rangle \mapsto p' \leftarrow \mathrm{idem}\,(i-1, j')\ x'; \boxed{p}), \\
&\qquad\qquad \mathrm{pad}\langle b, x' \rangle \mapsto p' \leftarrow \mathrm{idem}\,(i, j-1)\ x'; \boxed{p}\}
\end{aligned}
$$

5.3 Related Work

As we mentioned in the introduction, candidate concurrent type theories attempt to retrofit dependent types onto substructural session types, dubbed dependent session types. In the opposite direction, there has been work on embedding session-typed process calculi into dependent type theories, which we call embedded session types. Both solutions have similar limitations, as we will see below.

Dependent session types. Toninho et al. [93] initiated the line of work on dependent session types by presenting a session-typed process calculus in 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 channels with which processes communicate in the linear layer. In their retrospective paper ten years later [92], they note that most subsequent developments ([94, 32, 89], etc.) have similar issues precisely because non-linear dependence on linear objects must be restricted.

As somewhat of an exception to the rule, Toninho et al. [95] allow proof terms to depend on processes by way of a contextual monad following Toninho et al. [91]. While this relaxes the aforementioned restriction, their embedding of the functional (proof term) layer into the process layer indicates that each layer duplicates the other. As we mentioned in the introduction, DRSAX need not make this distinction.

Embedded session types. Dually, 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 [16, 100, 63, 78, 60]. Embedded implementation is certainly not opposed by DRSAX nor the line of work above, but moving the burden of proof to the meta level means that one must reason manually about the typing (especially if the object language is substructural and the metalanguage is structural) and operational semantics of programs. In general, reasoning internal to a type theory is more succinct since proofs take place modulo typing, judgmental equality, etc.

Dependent call-by-push-value. The polarized nature of DRSAX suggests a comparison to systems based on dependent call-by-push-value [67]. In particular, Niu et al.'s cost-aware logical framework [65] enforces a (generalized) *phase distinction* [83] between the intensional/cost and extensional/behavioral aspects of a program. Cost reasoning, which is the core of parallelism, is the subject of future work (in the context of SAX configurations). 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-coinduction.

Deductive verification. Dafny [54], F* [84], Why3 [41], and languages with liquid types [74, 96] allow formal verification by reflecting program dynamics into a refinement theory. Although we do not treat other effects, our aim is a native treatment of concurrency unlike in [55, 84, 77, 51]. Moreover, DRSAX is the only system to treat mixed induction-coinduction.

6 Further Proposed Work

In this proposal, we have worked towards a concurrent type theory and its metatheory. In this section, we summarize the pending research questions and future work not covered by this proposal. The primary goals are as follows:

- *Finalize the design of DRSAX.* Time permitting, we aim to explore the issues surrounding relative decidability of typechecking that we raised in the previous section.
- *Metatheory.* We will prove the termination of DRSAX programs by transporting our argument for RSAX to the dependent setting (translation to ω typing). Furthermore, as we indicated in the third section, we will semantically characterize (co)inductive types following [56].

Now, let us summarize the fallback options in case one or both options do not pan out.

- The final design of DRSAX may involve cutting back on the complexity of the refinement layer, especially observational equality. For example, we could restrict the theory of equality to purely positive types. The resulting system would be similar to a dependent extension of refined session types [46]. At worst, we may fallback completely to RSAX, which still retains index dependency (over arithmetic). In this case, we could investigate different refinement theories and compare their expressive power.

- If proving termination is insurmountable, we could fall back to a weaker metatheory: for example, type soundness.

We now identify four avenues of future work that will not be covered in the proposed work.

- *Implementation*: we are interested in developing a convenient surface language for (D)RSAX, following Rast [30], an implementation of resource-aware session types that includes arithmetic refinements.
- *Richer types*: as we mentioned in the second section, our long-term goal is to interpret adjoint SAX [71] within our framework, covering substructural dependent types. Furthermore, we are interested in generalizing to polymorphic and higher-kinded types [29].
- *Message passing*: we would like to transport our results to the asynchronous message passing interpretation of SAX [71], avoiding technically difficulties with typed asynchronous π-calculi [39].
- *Effects*: Rocha et al. [73] suggest that shared state and non-determinism can be incorporated by moving to differential linear logic; however, it is unclear how to transport these results to a structural and intuitionistic setting.

References

[1] Andreas Abel. "Productive Infinite Objects via Copatterns and Sized Types in Agda". Lorentz Center. Jan. 2014.

[2] Andreas Abel. "Semi-continuous Sized Types and Termination". In: *Logical Methods in Computer Science* Volume 4, Issue 2 (Apr. 2008).

[3] Andreas Abel. "Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types". In: *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012*. Ed. by Dale Miller and Zoltán Ésik. Vol. 77. EPTCS. 2012, pp. 1–11.

[4] Andreas Abel and Brigitte Pientka. "Well-founded recursion with copatterns and sized types". In: *Journal of Functional Programming* 26 (2016), e2.

[5] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational Equality, Now!" In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. PLPV '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 57–68. ISBN: 9781595936776. DOI: 10.1145/1292597.1292608. URL: https://doi.org/10.1145/1292597.1292608.

[6] JEAN-MARC ANDREOLI. “Logic Programming with Focusing Proofs in Linear Logic”. In: *Journal of Logic and Computation* 2.3 (June 1992), pp. 297–347. ISSN: 0955-792X. DOI: 10.1093/logcom/2.3.297. eprint: https://academic.oup.com/logcom/article-pdf/2/3/297/6137548/2-3-297.pdf. URL: https://doi.org/10.1093/logcom/2.3.297.

[7] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[8] David Baelde, Amina Doumane, and Alexis Saurin. “Infinitary Proof Theory: the Multiplicative Additive Case”. In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 42:1–42:17. ISBN: 978-3-95977-022-4.

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[10] Gilles Barthe, Maria João Frade, Eduardo Giménez, Luıs Pinto, and Tarmo Uustalu. “Type-based termination of recursive definitions”. In: *Math. Struct. Comput. Sci.* 14.1 (2004), pp. 97–141.

[11] Henning. Basold, J. J. M. M. Rutten, and H. Geuvers. “Mixed inductive-coinductive reasoning : types, programs and logic”. In: (2018).

[12] Stefano Berardi and Makoto Tatsuta. “Classical System of Martin-Löf’s Inductive Definitions Is Not Equivalent to Cyclic Proof System”. In: *Foundations of Software Science and Computation Structures*. Ed. by Javier Esparza and Andrzej S. Murawski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 301–317. ISBN: 978-3-662-54458-7.

[13] Frédéric Blanqui. “A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems”. In: *Rewriting Techniques and Applications*. Ed. by Vincent van Oostrom. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 24–39. ISBN: 978-3-540-25979-4.

[14] Frédéric Blanqui and Colin Riba. “Combining Typing and Size Constraints for Checking the Termination of Higher-Order Conditional Rewrite Systems”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Miki Hermann and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 105–119. ISBN: 978-3-540-48282-6.

[15] Frédéric Blanqui and Cody Roux. “On the relation between sized-types based termination and semantic labelling”. In: *18th EACSL Annual Conference on Computer Science Logic - CSL 09*. Full version. Coimbra, Portugal, Sept. 2009.

[16] Edwin Brady and Kevin Hammond. “Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols”. In: *Fundam. Inf.* 102.2 (Apr. 2010), pp. 145–176. ISSN: 0169-2968.

[17] James Brotherston. “Cyclic Proofs for First-Order Logic with Inductive Definitions”. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Ed. by Bernhard Beckert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 78–92. ISBN: 978-3-540-31822-4.

[18] James Brotherston, Richard Bornat, and Cristiano Calcagno. “Cyclic Proofs of Program Termination in Separation Logic”. In: *Proceedings of POPL-35*. ACM, 2008, pp. 101–112.

[19] James Brotherston and Alex Simpson. “Complete Sequent Calculi for Induction and Infinite Descent”. In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 2007, pp. 51–62. DOI: 10.1109/LICS.2007.16.

[20] Luis Caires, Frank Pfenning, and Bernardo Toninho. “Linear logic propositions as session types”. In: *Mathematical Structures in Computer Science* 26.3 (2016), pp. 367–423. DOI: 10.1017/S0960129514000218.

[21] Luıs Caires and Frank Pfenning. “Session Types as Intuitionistic Linear Propositions”. In: *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*. Paris, France: Springer LNCS 6269, Aug. 2010, pp. 222–236.

[22] Iliano Cervesato and Andre Scedrov. “Relating state-based and process-based concurrency through linear logic”. In: *Information and Computation* 207.10 (2009). Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006), pp. 1044–1077. ISSN: 0890-5401.

[23] Jonathan Chan and William J. Bowman. “Practical Sized Typing for Coq”. In: *CoRR* abs/1912.05601 (2019). arXiv: 1912.05601.

[24] Wei-Ngan Chin and Siau-Cheng Khoo. “Calculating Sized Types”. In: *Higher-Order and Symbolic Computation* 14.2 (2001), pp. 261–300.

[25] Ioana Cristescu and Daniel Hirschkoff. “Termination in a π-calculus with subtyping”. In: *Mathematical Structures in Computer Science* 26.8 (2016), pp. 1395–1432.

[26] Francesco Dagnino. “Foundations of regular coinduction”. In: *Logical Methods in Computer Science* Volume 17, Issue 4 (Oct. 2021). DOI: 10.46298/lmcs-17(4:2)2021. URL: https://lmcs.episciences.org/8539.

[27] Nils Anders Danielsson and Thorsten Altenkirch. *Mixing Induction and Coinduction*. 2009.

[28] Ornela Dardha and Jorge A. Pérez. “Comparing type systems for deadlock freedom”. In: *Journal of Logical and Algebraic Methods in Programming* 124 (2022), p. 100717. ISSN: 2352-2208.

[29] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. *Subtyping on Nested Polymorphic Session Types*. 2021. arXiv: 2103.15193 [cs.PL].

[30] Ankush Das and Frank Pfenning. *Rast: A Language for Resource-Aware Session Types*. 2020. arXiv: 2012.13129 [cs.PL].

[31] Ankush Das and Frank Pfenning. "Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description)". In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Ed. by Zena M. Ariola. Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 33:1–33:17. ISBN: 978-3-95977-155-9.

[32] Ankush Das and Frank Pfenning. "Session Types with Arithmetic Refinements". In: *31st International Conference on Concurrency Theory (CONCUR 2020)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 13:1–13:18. ISBN: 978-3-95977-160-3.

[33] 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. Bologna, Italy: Association for Computing Machinery, 2020. ISBN: 9781450388214.

[34] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. "Termination in Impure Concurrent Languages". In: *CONCUR 2010 - Concurrency Theory*. Ed. by Paul Gastin and François Laroussinie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 328–342. ISBN: 978-3-642-15375-4.

[35] Yuxin Deng and Davide Sangiorgi. "Ensuring termination by typability". In: *Information and Computation* 204.7 (2006), pp. 1045–1082. ISSN: 0890-5401.

[36] Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. "Session Logical Relations for Noninterference". In: *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781665448956. URL: https://doi.org/10.1109/LICS52264.2021.9470654.

[37] Farzaneh Derakhshan and Frank Pfenning. "Circular Proofs as Session-Typed Processes: A Local Validity Condition". In: *CoRR* abs/1908.01909 (Aug. 2019).

[38] Farzaneh Derakhshan and Frank Pfenning. "Circular Proofs in First-Order Linear Logic with Least and Greatest Fixed Points". In: *CoRR* abs/2001.05132 (Jan. 2020).

[39] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. "Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication". In: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*. Ed. by Patrick Cégielski and Arnaud Durand. Vol. 16. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 228–242. ISBN: 978-3-939897-42-2.

[40] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. “Semi-Axiomatic Sequent Calculus”. In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Ed. by Zena M. Ariola. Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 29:1–29:22. ISBN: 978-3-95977-155-9.

[41] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3 — Where Programs Meet Provers”. In: *Proceedings of the 22nd European Symposium on Programming*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, Mar. 2013, pp. 125–128.

[42] Jérôme Fortier and Luigi Santocanale. “Cuts for circular proofs: semantics and cut-elimination”. In: *Computer Science Logic 2013 (CSL 2013)*. Ed. by Simona Ronchi Della Rocca. Vol. 23. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 248–262. ISBN: 978-3-939897-60-6. DOI: 10.4230/LIPIcs.CSL.2013.248. URL: http://drops.dagstuhl.de/opus/volltexte/2013/4201.

[43] Neil Ghani, Peter G. Hancock, and Dirk Pattinson. “Representations of Stream Processors Using Nested Fixed Points”. In: *Log. Methods Comput. Sci.* 5.3 (2009).

[44] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. “Deadlock Analysis of Unbounded Process Networks”. In: *CONCUR 2014 – Concurrency Theory*. Ed. by Paolo Baldan and Daniele Gorla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 63–77. ISBN: 978-3-662-44584-6.

[45] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. USA: Cambridge University Press, 1989. ISBN: 0521371813.

[46] Dennis Griffith and Elsa L. Gunter. “LiquidPi: Inferrable Dependent Session Types”. In: *NASA Formal Methods*. Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 185–197. ISBN: 978-3-642-38088-4.

[47] Robert H. Halstead. “MULTILISP: A Language for Concurrent Symbolic Computation”. In: *ACM Trans. Program. Lang. Syst.* 7.4 (Oct. 1985), pp. 501–538. ISSN: 0164-0925.

[48] Robert Harper. *Practical Foundations for Programming Languages*. 2nd ed. Cambridge University Press, 2016. DOI: 10.1017/CBO9781316576892.

[49] William Alvin Howard. “The Formulae-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Hindley B., Seldin J. Roger, and P. Jonathan. 1980.

[50] 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. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 410–423. ISBN: 0897917693.

[51] 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. Beijing, China: Association for Computing Machinery, 2012, pp. 45–54. ISBN: 9781450312059. DOI: 10.1145/2254064.2254071. URL: https://doi.org/10.1145/2254064.2254071.

[52] Naoki Kobayashi. "A New Type System for Deadlock-Free Processes". In: *CONCUR 2006 – Concurrency Theory*. Ed. by Christel Baier and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 233–247. ISBN: 978-3-540-37377-3.

[53] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. "The Size-Change Principle for Program Termination". In: *SIGPLAN Not.* 36.3 (Jan. 2001), pp. 81–92. ISSN: 0362-1340.

[54] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[55] K. Rustan M. Leino. "Modeling Concurrency in Dafny". In: *Engineering Trustworthy Software Systems*. Ed. by Jonathan P. Bowen, Zhiming Liu, and Zili Zhang. Cham: Springer International Publishing, 2018, pp. 115–142. ISBN: 978-3-030-02928-9.

[56] K. Rustan M. Leino and Michał Moskal. "Co-induction Simply". In: *FM 2014: Formal Methods*. Ed. by Cliff Jones, Pekka Pihlajasaari, and Jun Sun. Cham: Springer International Publishing, 2014, pp. 382–398. ISBN: 978-3-319-06410-9.

[57] Rodolphe Lepigre and Christophe Raffalli. "Practical Subtyping for Curry-Style Languages". In: *ACM Trans. Program. Lang. Syst.* 41.1 (Feb. 2019). ISSN: 0164-0925.

[58] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*. USA: Kluwer Academic Publishers, 2004.

[59] Sam Lindley and J. Garrett Morris. "Talking Bananas: Structural Recursion for Session Types". In: *SIGPLAN Not.* 51.9 (Sept. 2016), pp. 434–447. ISSN: 0362-1340.

[60] Daniel Marshall and Dominic Orchard. "Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types". In: *Electronic Proceedings in Theoretical Computer Science* 356 (Mar. 2022), pp. 1–11. DOI: 10.4204/eptcs.356.1. URL: https://doi.org/10.4204%2Feptcs.356.1.

[61] Per Martin-Löf. "Infinite terms and a system of natural deduction". In: *Compositio Mathematica* 24.1 (1972), pp. 93–103.

[62] Dylan McDermott and Alan Mycroft. “Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order”. In: *Programming Languages and Systems*. Ed. by Luís Caires. Cham: Springer International Publishing, 2019, pp. 235–262. ISBN: 978-3-030-17184-1.

[63] Jan de Muijnck-Hughes, Edwin C. Brady, and Wim Vanderbauwhede. “Value-Dependent Session Design in a Dependently Typed Language”. In: *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*. Ed. by Francisco Martins and Dominic Orchard. Vol. 291. EPTCS. 2019, pp. 47–59. DOI: 10.4204/EPTCS.291.5. URL: https://doi.org/10.4204/EPTCS.291.5.

[64] Hiroshi Nakano. “A modality for recursion”. In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. 2000, pp. 255–266.

[65] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. “A Cost-Aware Logical Framework”. In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). DOI: 10.1145/3498670. arXiv: 2107.04663 [cs.PL].

[66] Luca Padovani. “Deadlock and Lock Freedom in the Linear π-Calculus”. In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. CSL-LICS ’14. Vienna, Austria: Association for Computing Machinery, 2014. ISBN: 9781450328869.

[67] Pierre-Marie Pédrot and Nicolas Tabareau. “The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371126. URL: https://doi.org/10.1145/3371126.

[68] Frank Pfenning and Robert J. Simmons. “Substructural Operational Semantics as Ordered Logic Programming”. In: *2009 24th Annual IEEE Symposium on Logic In Computer Science*. 2009, pp. 101–110. DOI: 10.1109/LICS.2009.8.

[69] Gordon Plotkin. *Lambda-definability and logical relations*. 1973.

[70] V. R. Pratt. “Semantical Considerations On Floyd-Hoare Logic”. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1976, pp. 109–121. DOI: 10.1109/SFCS.1976.27. URL: https://doi.ieeecomputersociety.org/10.1109/SFCS.1976.27.

[71] Klaas Pruiksma and Frank Pfenning. “A message-passing interpretation of adjoint logic”. In: *Journal of Logical and Algebraic Methods in Programming* 120 (2021), p. 100637. ISSN: 2352-2208.

[72] Klaas Pruiksma and Frank Pfenning. “Back to Futures”. In: *Journal of Functional Programming* 32 (2022), e6. DOI: 10.1017/S0956796822000016.

[73] Pedro Rocha and Luıs Caires. “Propositions-as-Types and Shared State”. In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021). DOI: 10.1145/3473584. URL: https://doi.org/10.1145/3473584.

[74] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. “Liquid Types”. In: *SIGPLAN Not.* 43.6 (June 2008), pp. 159–169. ISSN: 0362-1340. DOI: 10.1145/1379022.1375602. URL: https://doi.org/10.1145/1379022.1375602.

[75] Jorge Luis Sacchini. “Linear Sized Types in the Calculus of Constructions”. In: *Functional and Logic Programming*. Ed. by Michael Codish and Eijiro Sumii. Cham: Springer International Publishing, 2014, pp. 169–185. ISBN: 978-3-319-07151-0.

[76] Davide Sangiorgi. “Termination of processes”. In: *Mathematical Structures in Computer Science* 16.1 (2006), pp. 1–39.

[77] César Santos, Francisco Martins, and Vasco Thudichum Vasconcelos. “Deductive Verification of Parallel Programs Using Why3”. In: *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015*. Ed. by Sophia Knight, Ivan Lanese, Alberto Lluch-Lafuente, and Hugo Torres Vieira. Vol. 189. EPTCS. 2015, pp. 128–142. DOI: 10.4204/EPTCS.189.11. URL: https://doi.org/10.4204/EPTCS.189.11.

[78] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. *Effpi: A Toolkit for Verified Message-Passing Programs in Dotty*. URL: https://doi.org/10.1145/3325968.

[79] Kurt Schutte. *Proof theory (translation from German by J. N. Crossley)*. Springer-Verlag Berlin, 1977. ISBN: 0387079114.

[80] Paula Severi, Luca Padovani, Emilio Tuosto, and Mariangiola Dezani-Ciancaglini. “On Sessions and Infinite Data”. In: *Coordination Models and Languages*. Ed. by Alberto Lluch Lafuente and José Proença. Cham: Springer International Publishing, 2016, pp. 245–261. ISBN: 978-3-319-39519-7.

[81] Siva Somayyajula and Frank Pfenning. “Type-Based Termination for Futures”. In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. 2022.

[82] Richard Statman. “Logical relations and the typed -calculus”. In: *Information and Control* 65.2 (1985), pp. 85–97. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(85)80001-2. URL: https://www.sciencedirect.com/science/article/pii/S0019995885800012.

[83] Jonathan Sterling and Robert Harper. “Logical Relations as Types: Proof-Relevant Parametricity for Program Modules”. In: *Journal of the ACM* 68.6 (Oct. 2021). ISSN: 0004-5411. DOI: 10.1145/3474834. arXiv: 2010.08599 [cs.PL].

[84] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martınez. “SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs”. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3409003. URL: https://doi.org/10.1145/3409003.

[85] W. W. Tait. “Intensional interpretations of functionals of finite type I”. In: *Journal of Symbolic Logic* 32.2 (1967), pp. 198–212. DOI: 10.2307/2271658.

[86] W.W. Tait. “Infinitely Long Terms of Transfinite Type”. In: *Formal Systems and Recursive Functions*. Ed. by J.N. Crossley and M.A.E. Dummett. Vol. 40. Studies in Logic and the Foundations of Mathematics. Elsevier, 1965, pp. 176–185. DOI: https://doi.org/10.1016/S0049-237X(08)71689-6. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08716896.

[87] Gadi Tellez and James Brotherston. “Automatically Verifying Temporal Properties of Pointer Programs with Cyclic Proof”. In: *Journal of Automated Reasoning* 64.3 (2020), pp. 555–578. DOI: 10.1007/s10817-019-09532-0. URL: https://doi.org/10.1007/s10817-019-09532-0.

[88] The Coq Development Team. *The Coq Proof Assistant*. Version 8.13. Jan. 2021.

[89] Peter Thiemann and Vasco T. Vasconcelos. “Label-Dependent Session Types”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371135. URL: https://doi.org/10.1145/3371135.

[90] René Thiemann and Jürgen Giesl. “Size-Change Termination for Term Rewriting”. In: *Rewriting Techniques and Applications*. Ed. by Robert Nieuwenhuis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 264–278. ISBN: 978-3-540-44881-5.

[91] Bernardo Toninho, Luis Caires, and Frank Pfenning. “Higher-Order Processes, Functions, and Sessions: A Monadic Integration”. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 350–369. ISBN: 978-3-642-37036-6.

[92] Bernardo Toninho, Luıs Caires, and Frank Pfenning. “A Decade of Dependent Session Types”. In: *23rd International Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450386890. URL: https://doi.org/10.1145/3479394.3479398.

[93] Bernardo Toninho, Luıs 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. Odense, Denmark: Association for Computing Machinery, 2011, pp. 161–172. ISBN: 9781450307765. DOI: 10.1145/2003476.2003499. URL: https://doi.org/10.1145/2003476.2003499.

[94] Bernardo Toninho and Nobuko Yoshida. “Certifying data in multiparty session types”. In: *Journal of Logical and Algebraic Methods in Programming* 90 (2017), pp. 61–83. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2016.11.005. URL: https://www.sciencedirect.com/science/article/pii/S2352220816300864.

[95] Bernardo Toninho and Nobuko Yoshida. “Depending on Session-Typed Processes”. In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Com-

puter Science. Springer, 2018, pp. 128–145. DOI: 10.1007/978-3-319-89366-2_7. URL: https://doi.org/10.1007/978-3-319-89366-2%5C_7.

[96] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. "Refinement Reflection: Complete Verification with SMT". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158141.

[97] Andrea Vezzosi. "Total (Co)Programming with Guarded Recursion". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by Tarmo Uustalu. Tallinn, Estonia: Institute of Cybernetics at Tallinn University of Technology, 2015, pp. 77–78. ISBN: 978-9949-430-87-1.

[98] Philip Wadler. "Propositions as sessions". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Peter Thiemann and Robby Bruce Findler. ACM, 2012, pp. 273–286.

[99] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. "A Concurrent Logical Framework: The Propositional Fragment". In: *Types for Proofs and Programs*. Ed. by Stefano Berardi, Mario Coppo, and Ferruccio Damiani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 355–377. ISBN: 978-3-540-24849-1.

[100] Hanwen Wu and Hongwei Xi. "Dependent Session Types". In: *CoRR* abs/1704.07004 (2017). arXiv: 1704.07004. URL: http://arxiv.org/abs/1704.07004.

[101] Hongwei Xi. "Dependent types for program termination verification". In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 2001, pp. 231–242.

[102] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming". In: *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*. Ed. by A. Aiken. ACM Press, Jan. 1999, pp. 214–227.

[103] Serge Yoccoz. "Constructive Aspects of the Omega-Rule: Application to Proof Systems in Computer Science and Algorithmic Logic". In: *Proceedings on Mathematical Foundations of Computer Science 1989*. MFCS '89. Berlin, Heidelberg: Springer-Verlag, 1989, pp. 553–565. ISBN: 3540514864.

[104] Nobuko Yoshida. "Graph types for monadic mobile processes". In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by V. Chandru and V. Vinay. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 371–386. ISBN: 978-3-540-49631-1.

[105] Nobuko Yoshida, Martin Berger, and Kohei Honda. "Strong normalisation in the π-calculus". In: *Information and Computation* 191.2 (2004), pp. 145–202. ISSN: 0890-5401.