Type-Based Termination for Futures

Siva Somayyajula
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Frank Pfenning
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

In sequential functional languages, sized types enable termination checking of programs with complex patterns of recursion in the presence of mixed inductive-coinductive types. In this paper, we adapt sized types and their metatheory to the concurrent setting. We extend the semi-axiomatic sequent calculus, a subsuming paradigm for futures-based functional concurrency, and its underlying operational semantics with recursion and arithmetic refinements. The latter enables a new and highly general sized type scheme we call *sized type refinements*. As a widely applicable technical device, we type recursive programs with infinitely deep typing derivations that unfold all recursive calls. Then, we observe that certain such derivations can be made infinitely wide but finitely deep. The resulting trees serve as the induction target of our strong normalization result, which we develop via a novel logical relations argument.

2012 ACM Subject Classification Theory of computation → Proof theory; Computing methodologies → Concurrent programming languages

Keywords and phrases type-based termination, sized types, futures, concurrency, infinite proofs

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.12

Related Version *Extended Version*: `https://arxiv.org/abs/2105.06024`

Funding This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092.
Siva Somayyajula: Partially supported by the Sansom Graduate Fellowship in Computer Science.

Acknowledgements We would like to thank Farzaneh Derakhshan, Klaas Pruiksma, Henry DeYoung, Ankush Das, and the anonymous reviewers for helpful discussion and suggestions regarding the contents of this paper.

1 Introduction

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 [59] 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. [39] and further developed by others [8, 10, 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 1 adorned with our novel *sized type refinements*: $\mathrm{nat}[i]$ describes unary natural numbers less than or equal to i and $\mathrm{stream}_A[i]$ describes infinite A-streams that allow the first $i+1$

elements to be observed before reaching potentially undefined or divergent behavior. $\oplus$ and & are respectively analogous to eager variant record and lazy record types in the functional setting.

▶ **Example 1** (Recursive types)**.**

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

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

The phrases $\phi \wedge \ldots$ and $\phi \Rightarrow \ldots$ are *constrained types*, so that the succ branch of $\mathrm{nat}[i]$ produces a nat at height $i-1$ *when* $i > 0$ whereas the tail branch of $\mathrm{stream}_A[i]$ can produce the remainder of the stream at depth $i-1$ *assuming* $i > 0$. Starting from $\mathrm{nat}[i]$, recursing *on*, for example, $\mathrm{nat}[i-1]$ ($i > 0$ 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 $\mathrm{stream}_A[i]$, recursing *into* $\mathrm{stream}_A[i-1]$ (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 and introduction encodes induction and coinduction, respectively. To see how sizes are utilized in the definition of recursive programs, consider the type signatures below. We will define the code of these programs in Example 5.

▶ **Example 2** (Evens and odds I)**.** Postponing the details of our typing judgment for the moment, 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. Note that indexing begins at zero.

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

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. We provide implementations for both versions in Example 5.

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

$\exists j.\, X[j]$ 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 [65]. 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 $j \leq i$.

Sized types are *compositional*: since termination checking is reduced to an instance of typechecking, we avoid the brittleness of syntactic termination checking. However, we find that *ad hoc* features for implementing size arithmetic in the prior work can be subsumed by

more general *arithmetic refinements* [26, 65], giving rise to our notion of sized type refinements that combine the "good parts" of modern sized type systems. First, the instances of constraint conjunction and implication to encode inductive and coinductive types, respectively, 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 2 are given in $\mathsf{CIC}_{\ell}^{\widehat{}}$ [54] and MiniAgda [3, 1]. Furthermore, we avoid transfinite indices in favor of permitting some unbounded quantification (following Vezzosi [62]), achieving the effect of somewhat complicated infinite sizes without leaving finite arithmetic.

Moreover, some prior work, which is based on sequential functional languages, encodes recursion via various fixed point combinators that make both mixed inductive-coinductive programming [9] and substructural typing difficult, the latter requiring the use of the ! modality [63]. Thus, like $\mathsf{F}_{\omega}^{\mathsf{cop}}$ [4], we consider a signature of parametric recursive definitions. However, we make typing derivations for recursive programs infinitely deep by unfolding recursive calls *ad infinitum* [13, 45], which is not only more elegant than finitary typing, but also simplifies our normalization argument. To prove strong normalization, we observe that *arithmetically closed* typing derivations, which have no free arithmetic variables or constraint assumptions, can be translated to infinitely wide but finitely deep trees of a different judgment. The resulting derivations are then the induction target for our proof, leaving the option of making the original typing judgment arbitrarily rich. Thus, although our proposed language is not substructural, this result extends to programs that use their data substructurally. In short, our contributions are as follows:

1. A general system of sized types based on arithmetic refinements subsuming features of prior systems, such as the mixed inductive-coinductive types of MiniAgda [3] as well as the linear size arithmetic of $\mathsf{CIC}_{\ell}^{\widehat{}}$ [54]. Moreover, we do not depend on transfinite arithmetic.
2. The first language for mixed inductive-coinductive programming that is a subsuming paradigm [46] for futures-based functional concurrency.
3. A method for proving normalization in the presence of infinitely deep typing derivations by translation to infinitely wide but finitely deep trees. The diamond property of program reduction implies that normalization is schedule-independent, encompassing call-by-need and call-by-value strategies.

We define SAX^{∞}, which extends the semi-axiomatic sequent calculus (SAX) [33] with arithmetic refinements, recursion, and infinitely deep typing derivations (Section 2). Then, we define an auxiliary type system called SAX^{ω} which has infinitely wide but finitely deep derivations to which we translate the derivations of SAX^{∞} (Section 3). Then, we show that all SAX^{ω}-typed programs are strongly normalizing by a novel logical relations argument over *configurations* of processes that capture the state of a concurrent computation (Section 4).

2 SAX^{∞}

In this section, we extend SAX [33] with recursion and arithmetic refinements in the style of Das and Pfenning [26]. SAX is a logic-based formalism and subsuming paradigm [46] for concurrent functional programming that conceives call-by-need and call-by-value strategies as particular concurrent schedules [51]. Concurrency and parallelism devices like fork/join, futures [37], and SILL-style [61] monadic concurrency can all be encoded and used side-by-side in SAX [51].

To review SAX, let us make observations about proof-theoretic *polarity*. In the sequent calculus, inference rules are either *invertible* – 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*. The key innovation of SAX is to replace the noninvertible rules with their axiomatic counterparts in a Hilbert-style system. 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.

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

Since the axiomatic rules drop the premises of their sequent calculus counterparts, cut elimination corresponds to asynchronous communication just as the standard sequent calculus models synchronous communication [15]. In particular, SAX has a *shared memory interpretation*, mirroring the memory-based semantics of *futures* [37]. A future x of type A either contains an object of type A or is not yet populated. A process reading from x either succeeds immediately or blocks if x is not yet populated. As a result, the sequent becomes the typing judgment (extended with arithmetic refinements in the style of [26]):

$$\overbrace{i, j, \ldots}^{\mathcal{V}}; \overbrace{\phi, \psi, \ldots}^{\mathcal{C}}; \overbrace{x : A, y : B, \ldots}^{\Gamma} \vdash^{\overline{e}} P :: (z : C)$$

where the *arithmetic variables* in $\mathcal{V}$ are free in the *constraints* (arithmetic formulas) in $\mathcal{C}$, the types in Γ, the *process* P, and type C; moreover, the *address variables* in Γ, which are free in P, stand for addresses of memory cells representing futures. In particular, P reads from $x, y, \ldots$ (*sources*) and writes to z (a *destination*) according to the protocols specified by $A, B, \ldots$ and C, respectively. z is written to exactly once corresponding to the population of a future [37]. Lastly, the vector (indicated by the overline) of *arithmetic expressions* $\overline{e}$ will be used to track the sizes encountered at each recursive call as mentioned in the introduction. Now, let us examine the definitions of types and processes. For our purposes, detailed syntaxes for expressions e and formulas ϕ are unnecessary.

▶ **Definition 3** (Type). *Types are defined by the following grammar, presupposing some mutually recursive type definitions of the form $X[\overline{i}] = A_X(\overline{i})$. Positive types (in the left column) and negative types (in the right column) are colored red and black, respectively. Recursive type names are colored blue since they take on the polarity of their definientia.*

$A, B :=$	$\mathbf{1}$	*unit*	$\mid$	$X[\overline{e}]$	*equirecursive type*
$\mid$	$A \otimes B$	*eager pair*	$\mid$	$A \to B$	*function*
$\mid$	$\oplus\{\ell : A_\ell\}_{\ell \in S}$	*eager variant record*	$\mid$	$\&\{\ell : A_\ell\}_{\ell \in S}$	*lazy record*
$\mid$	$\phi \wedge A$	*constraint conjunction*	$\mid$	$\phi \Rightarrow A$	*constraint implication*
$\mid$	$\exists i.\, A(i)$	*arithmetic dependent pair*	$\mid$	$\forall i.\, A(i)$	*arithmetic dep. function*

There are eight kinds of processes: two for the structural rules (identity and cut), one for each combination of type polarity (positive or negative) and rule type (left or right), one for definition calls, and one for unreachable code.

▶ **Definition 4** (Process). *Processes are defined by the following grammar. The superscripts* R *and* W *indicating reading from or writing to a cell.*

$P, Q :=$	$y^{\mathrm{W}} \leftarrow x^{\mathrm{R}}$	*copy contents of x to y*
$\mid$	$x \leftarrow P(x); Q(x)$	*allocate x, spawn P to write to x and concurrently proceed as Q, which may read from x*
$\mid$	$x^{\mathrm{W}}.V$	*write value V to x*
$\mid$	$\mathbf{case}\, x^{\mathrm{R}}\, K$	*read value stored in x and pass it to continuation K*
$\mid$	$\mathbf{case}\, x^{\mathrm{W}}\, K$	*write continuation K to x*
$\mid$	$x^{\mathrm{R}}.V$	*read continuation stored in x then pass value V to it*
$\mid$	$y \leftarrow f\ \overline{e}\ \overline{x}$	*expands to $P_f(\overline{e}, \overline{x}, y)$ from a signature of mutually recursive definitions of the form $y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y)$*
$\mid$	$\mathbf{impossible}$	*unreachable code due to inconsistent arithmetic context*

The first two kinds of processes correspond to the identity and cut rules. Values V and continuations K are specified on a per-type-and-rule basis in the following two tables. Note the address variable x distinguished by each rule.

polarity	*right rule*	*left rule*
positive	$x^{\mathrm{W}}.V$	$\mathbf{case}\, x^{\mathrm{R}}\, K$
negative	$\mathbf{case}\, x^{\mathrm{W}}\, K$	$x^{\mathrm{R}}.V$

type(s)	*value V*	*continuation K*
$\mathbf{1}$	$\langle\rangle$	$\langle\rangle \Rightarrow P$
$\otimes, \rightarrow$	$\langle y, z\rangle$	$\langle y, z\rangle \Rightarrow P(y, z)$
$\&, \oplus$	$\ell\, y$	$\{\ell\, y \Rightarrow P(y)\}_{\ell \in S}$
$\wedge, \Rightarrow$	$\langle *, y\rangle$	$\langle *, y\rangle \Rightarrow P(y)$
$\forall, \exists$	$\langle e, y\rangle$	$\langle i, y\rangle \Rightarrow P(i, y)$

To borrow terminology from linear logic, the "multiplicative" group $(\mathbf{1}, \otimes, \rightarrow)$ is concerned with writing addresses, whereas the "additive" group $(\oplus, \&)$ is concerned with writing labels and their case analysis. Constrained types read and write a placeholder $*$ indicating that a constraint is asserted or assumed. However, we will suppress instances of $*$ in the example code given, since assumptions and assertions are inferrable in the absence of consecutively alternating constraints (e.g., $\phi \wedge (\psi \Rightarrow A)$). On the other hand, the arithmetic data communicated by quantifiers are visible since inference is difficult in general [27]. Now that we are acquainted with the process syntax, let us complete Example 2.

▶ **Example 5** (Evens and odds II)**.** Recall that we are implementing the following signature and $\mathrm{stream}_A[i] = \&\{\mathrm{head} : A, \mathrm{tail} : i > 0 \Rightarrow \mathrm{stream}_A[i-1]\}$.

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

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.

$$y \leftarrow \mathrm{evens}\ i\ x = \mathbf{case}\, y^{\mathrm{W}}\{\, \mathrm{head}\ h \Rightarrow x^{\mathrm{R}}.\, \mathrm{head}\ h,\ \underbrace{\mathrm{tail}\, y_t}_{i>0\ \text{assumed}} \Rightarrow x_t \leftarrow \underbrace{x^{\mathrm{R}}.\, \mathrm{tail}\, x_t}_{2i>0\ \text{asserted}}; \underbrace{y_t \leftarrow \mathrm{odds}\, (i-1)\ x_t}_{i;i>0 \vdash i-1<i\ \text{checked}}\}$$

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

By inlining the definition of odds in evens and vice versa, both programs terminate according to our criterion from the introduction even though odds calls evens with argument i. However, we sketch an alternate termination argument for similar such definitions at the end of Section 3. On the other hand, consider the alternate signature we gave.

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

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 (boxed and highlighted yellow), which can be expanded into SAX [51].

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

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.

$$y \leftarrow \mathrm{evens}\ i\ x = \mathbf{case}\, y^{\mathrm{W}}\{\mathrm{head}\ h \Rightarrow y \leftarrow \mathrm{head}\, x,$$
$$\mathrm{tail}\, y_t \Rightarrow x_t \leftarrow \mathrm{tail}\, x; y_t \leftarrow \mathrm{odds}\ (i-1)\ x_t\}$$
$$y \leftarrow \mathrm{odds}\ i\ x = x_t \leftarrow \mathrm{tail}\, x; y \leftarrow \mathrm{evens}\ i\ x_t$$

Refer to Figure 1 for the full process typing judgment – we will comment on specific rules when necessary, but section 5 of [33] discusses the propositional rules more closely. In particular, the arithmetic typing rules make use of a *well-formedness judgment* $\mathcal{V}; \mathcal{C} \vdash e$ and *entailment* $\mathcal{V}; \mathcal{C} \vdash \phi$. Moreover, since the constraint rules are implicit, the noninvertible ones are not axiomatic. Most importantly, there are two rules for recursive calls; let us reproduce them below.

$$\frac{\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}} P :: (y : A)}{\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}}_{\infty} P :: (y : A)}\ \infty \qquad \frac{\mathcal{V}; \mathcal{C} \vdash \overline{e'} < \overline{e} \quad y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y) \quad \mathcal{V}; \mathcal{C}; \overline{x} : \overline{A} \vdash^{\overline{e'}}_{\infty} P_f(\overline{e'}, \overline{x}, y) :: (y : A)}{\mathcal{V}; \mathcal{C}; \Gamma, \overline{x} : \overline{A} \vdash^{\overline{e}} y \leftarrow f\ \overline{e'}\ \overline{x} :: (y : A)}\ \mathrm{call}$$

Our process typing judgment is itself *mixed inductive-coinductive* [22] – we introduce the auxiliary judgment $\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}}_{\infty} P :: (y : A)$ that is *coinductively* generated by the ∞ rule (indicated by the double line). Since the premise of the call rule refers to $\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}}_{\infty} P :: (y : A)$, all valid typing derivations are trees whose infinite branches have a call-∞ pair occurring 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[1] for termination.

For typechecking in finite time, restricting our type system to *circular derivations*, which can be represented as finite trees with loops, and decidable arithmetic (e.g., Presburger) is sufficient, although we do not show this formally. In short, such a restricted system can

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

$$\frac{}{\mathcal{V};\mathcal{C};\Gamma,x:A\vdash^{\overline{e}} y^{\mathrm{W}}\leftarrow x^{\mathrm{R}}::(y:A)}\ \mathrm{id}\qquad \frac{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}} P(x)::(x:A)\quad \mathcal{V};\mathcal{C};\Gamma,x:A\vdash^{\overline{e}} Q(x)::(z:C)}{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}} x\leftarrow P(x);Q(x)::(z:C)}\ \mathrm{cut}$$

$$\frac{}{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}} x^{\mathrm{W}}.\langle\rangle::(x:\mathbf{1})}\ \mathbf{1}\mathrm{R}\qquad \frac{\mathcal{V};\mathcal{C};\Gamma,x:\mathbf{1}\vdash^{\overline{e}} P::(z:C)}{\mathcal{V};\mathcal{C};\Gamma,x:\mathbf{1}\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{R}}\,(\langle\rangle\Rightarrow P)::(z:C)}\ \mathbf{1}\mathrm{L}$$

$$\frac{}{\mathcal{V};\mathcal{C};\Gamma,y:A,z:B\vdash^{\overline{e}} x^{\mathrm{W}}.\langle y,z\rangle::(x:A\otimes B)}\otimes\mathrm{R}\qquad \frac{\mathcal{V};\mathcal{C};\Gamma,x:A\otimes B,y:A,z:B\vdash^{\overline{e}} P(y,z)::(w:C)}{\mathcal{V};\mathcal{C};\Gamma,x:A\otimes B\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{R}}\,(\langle y,z\rangle\Rightarrow P(y,z))::(w:C)}\otimes\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C};\Gamma,y:A\vdash^{\overline{e}} P(y,z)::(z:B)}{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{W}}\,(\langle y,z\rangle\Rightarrow P(y,z))::(x:A\to B)}\to\mathrm{R}\qquad \frac{}{\mathcal{V};\mathcal{C};\Gamma,x:A\to B,y:A\vdash^{\overline{e}} x^{\mathrm{R}}.\langle y,z\rangle::(z:B)}\to\mathrm{L}$$

$$\frac{k\in S}{\mathcal{V};\mathcal{C};\Gamma,y:A_k\vdash^{\overline{e}} x^{\mathrm{W}}.k\,y::(x:\oplus\{\ell:A_\ell\}_{\ell\in S})}\oplus\mathrm{R}\qquad \frac{\{\mathcal{V};\mathcal{C};\Gamma,x:\oplus\{\ell:A_\ell\}_{\ell\in S},y:A_\ell\vdash^{\overline{e}} P(y)::(z:C)\}_{\ell\in S}}{\mathcal{V};\mathcal{C};\Gamma,x:\oplus\{\ell:A_\ell\}_{\ell\in S}\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{R}}\,\{\ell\,y\Rightarrow P_\ell(y)\}_{\ell\in S}::(z:C)}\oplus\mathrm{L}$$

$$\frac{\{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}} P(y)::(y:A_\ell)\}_{\ell\in S}}{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{W}}\,\{\ell\,y\Rightarrow P_\ell(y)\}_{\ell\in S}::(x:\&\{\ell:A_\ell\}_{\ell\in S})}\&\mathrm{R}\qquad \frac{k\in S}{\mathcal{V};\mathcal{C};\Gamma,x:\&\{\ell:A_\ell\}_{\ell\in S}\vdash^{\overline{e}} x^{\mathrm{R}}.k\,y::(y:A_k)}\&\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C}\vdash e}{\mathcal{V};\mathcal{C};\Gamma,y:A(e)\vdash^{\overline{e}} x^{\mathrm{W}}.\langle e,y\rangle::(x:\exists i.\,A(i))}\exists\mathrm{R}\qquad \frac{\mathcal{V},i;\mathcal{C};\Gamma,x:\exists i.\,A(i),y:A(i)\vdash^{\overline{e}} P(i,y)::(z:C)}{\mathcal{V};\mathcal{C};\Gamma,x:\exists i.\,A(i)\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{R}}\,(\langle i,y\rangle\Rightarrow P(i,y))::(z:C)}\exists\mathrm{L}$$

$$\frac{\mathcal{V},i;\mathcal{C};\Gamma\vdash^{\overline{e}} P(i,y)::(y:A(i))}{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{W}}\,(\langle i,y\rangle\Rightarrow P(i,y))::(x:\forall i.\,A(i))}\forall\mathrm{R}\qquad \frac{\mathcal{V};\mathcal{C}\vdash e}{\mathcal{V};\mathcal{C};\Gamma,x:\forall i.\,A(i)\vdash^{\overline{e}} x^{\mathrm{R}}.\langle e,y\rangle::(y:A(e))}\forall\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C}\vdash\phi}{\mathcal{V};\mathcal{C};\Gamma,y:A\vdash^{\overline{e}} x^{\mathrm{W}}.\langle *,y\rangle::(x:\phi\wedge A)}\wedge\mathrm{R}\qquad \frac{\mathcal{V};\mathcal{C},\phi;\Gamma,x:\phi\wedge A,y:A\vdash^{\overline{e}} P(y)::(z:C)}{\mathcal{V};\mathcal{C};\Gamma,x:\phi\wedge A\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{R}}\,(\langle *,y\rangle\Rightarrow P(y))::(z:C)}\wedge\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C},\phi;\Gamma\vdash^{\overline{e}} P(y)::(y:A)}{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}}\mathbf{case}\,x^{\mathrm{W}}\,(\langle *,y\rangle\Rightarrow P(y))::(x:\phi\Rightarrow A)}\Rightarrow\mathrm{R}\qquad \frac{\mathcal{V};\mathcal{C}\vdash\phi}{\mathcal{V};\mathcal{C};\Gamma,x:\phi\Rightarrow A\vdash^{\overline{e}} x^{\mathrm{R}}.\langle *,y\rangle::(y:A)}\Rightarrow\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}} P::(y:A)}{\mathcal{V};\mathcal{C};\Gamma\vdash^{\overline{e}}_{\infty} P::(y:A)}\infty\qquad \frac{\mathcal{V};\mathcal{C}\vdash\overline{e'}<\overline{e}\quad y\leftarrow f\,\overline{i}\,\overline{x}=P_f(\overline{i},\overline{x},y)\quad \mathcal{V};\mathcal{C};\overline{x}:\overline{A}\vdash^{\overline{e'}}_{\infty} P_f(\overline{e'},\overline{x},y)::(y:A)}{\mathcal{V};\mathcal{C};\Gamma,\overline{x}:\overline{A}\vdash^{\overline{e}} y\leftarrow f\,\overline{e'}\,\overline{x}::(y:A)}\ \mathrm{call}$$

Figure 1 SAX$^{\infty}$ Typing.

be put in correspondence with a finitary system that detects said loops [13, 21, 51] and arithmetic constraint obligations can be discharged mechanically [25]. In Example 22 in the appendix, we show a hypothetical instance of typechecking (note that we use "$D\in J$" to indicate a derivation D of the judgment J). Now, consider the following example that demonstrates a use case of mixed induction-coinduction in concurrency.

▶ **Example 6** (Left-fair streams)**.** Let us define the mixed inductive-coinductive type $\mathrm{lfair}_{A,B}[i,j]$ of *left-fair streams* [9]: 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.

$$\mathrm{lfair}_{A,B}[i,j]=\oplus\{\mathrm{now}:\&\{\mathrm{head}:A,\mathrm{tail}:\mathrm{lfair}^{\nu}_{A,B}[i,j]\},\mathrm{later}:B\otimes\mathrm{lfair}^{\mu}_{A,B}[i,j]\}$$
$$\mathrm{lfair}^{\nu}_{A,B}[i,j]=i>0\Rightarrow\exists j'.\ \mathrm{lfair}_{A,B}[i-1,j']$$
$$\mathrm{lfair}^{\mu}_{A,B}[i,j]=j>0\wedge\mathrm{lfair}_{A,B}[i,j-1]$$

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, $\mathrm{lfair}^{\nu}_{A,B}[i,j]$). 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, $\mathrm{lfair}^{\mu}_{A,B}[i,j]$). 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 [9] that removes all of a left-fair stream's timeout acknowledgements concurrently, returning an A-stream. For brevity, we nest patterns (boxed and highlighted yellow), which can be expanded into nested matches [51].

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

3 SAX$^{\omega}$

Even without presenting the operational semantics yet, it is unclear how to prove normalization of program reduction in the presence of infinite typing derivations. As a result, we give a purely inductive process typing called SAX$^{\omega}$ with the judgment $\Gamma \vdash^{\omega} P :: (x:A)$ (selected rules in Figure 2 with the rest in Figure 5). By dropping the arithmetic and constraint contexts, 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 (like the ω-rule of arithmetic [40]). Moreover, the premises of $\wedge\mathrm{L}^{\omega}$ and $\Rightarrow\mathrm{R}^{\omega}$ assume the closed constraint ϕ (which has no free arithmetic variables) holds at the meta level instead of adding it to a constraint context.

Most importantly, the call rule does not refer to a coinductively-defined auxiliary judgment, because in the absence of free arithmetic variables, the tracked size arguments decrease from some $\overline{n}$ to $\overline{n'}$ to etc. Since the lexicographic order on fixed-length 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 arithmetically closed SAX$^{\infty}$ derivations to SAX$^{\omega}$ derivations. In fact, any SAX$^{\infty}$ derivation can be made arithmetically closed by substituting each of its free arithmetic variables for numbers that validate (and therefore discharge) its constraints. By trading infinitely deep derivations for infinitely wide but finitely deep ones, we may complete a logical relations argument by induction over a SAX$^{\omega}$ derivation. Thus, let us examine the translation theorem.

▶ **Theorem 7** (Translation). *If* $D \in \cdot;\cdot;\Gamma \vdash^{\overline{n}} P :: (x:A)$, *then* $\Gamma \vdash^{\omega} P :: (x:A)$.

Proof. By lexicographic induction on $(\overline{n}, D)$, we cover the important cases.

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

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

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

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

$$\frac{\cdot ; \cdot \vdash \phi}{\Gamma, y : A \vdash^\omega x^{\mathrm{W}}.\langle *, y\rangle :: (x : \phi \wedge A)}\ \wedge\mathrm{R}^\omega \qquad \frac{\Gamma, x : \phi \wedge A, y : A \vdash^\omega P(y) :: (z : C) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \wedge A \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (z : C)}\ \wedge\mathrm{L}^\omega$$

$$\frac{\Gamma \vdash^\omega P(y) :: (y : A) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma \vdash^\omega \mathbf{case}\, x^{\mathrm{W}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (x : \phi \Rightarrow A)}\ \Rightarrow\mathrm{R}^\omega \qquad \frac{\cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \Rightarrow A \vdash^\omega x^{\mathrm{R}}.\langle *, y\rangle :: (y : A)}\ \Rightarrow\mathrm{L}^\omega$$

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

Figure 2 Selected SAX^ω Typing Rules.

2. Analogously, when D ends in $\wedge\mathrm{L}$ or $\Rightarrow\mathrm{R}$, its subderivation D' assumes ϕ. The premises of the corresponding SAX^ω rules $\wedge\mathrm{L}^\omega$ and $\Rightarrow\mathrm{R}^\omega$ assume $E \in \cdot ; \cdot \vdash \phi$, so we finish by induction on $(\overline{n}, E \cdot D')$ where $E \cdot D'$ cuts ϕ out of D'.
3. Finally, assume D ends in the call rule with subderivation D'. By inversion, D' ends in the ∞ 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 SAX^ω call rule. ◀

As we mentioned in the introduction, we can make the SAX^∞ judgment arbitrarily rich to support more complex patterns of recursion. As long as derivations in that system can be translated to SAX^ω, the logical relations argument over SAX^ω typing that we detail in Section 4 does not change. For example, consider the following additions.

1. *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).
2. *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 5, 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 $(\overline{n}, f, D)$.

4 Semantics and Normalization

In this section, we will give an operational semantics for *configurations* of processes. Then, we will show that all SAX^ω-typed processes are strongly normalizing. Program execution based on processes alone is impractical, because cut elimination only facilitates communication

between two processes at a time. Thus, DeYoung et al. [33] define programs in SAX as *configurations* of simultaneously executing processes and the memory cells with which they communicate. Relatedly, the metatheory of the π-calculus must be defined up-to structural congruence to achieve a similar effect [53].

▶ **Definition 8** (Configuration). *Let* $a, b, c, \ldots \in \mathsf{Addr}$ *be* cell addresses *and* $W := V \mid K$. *A* configuration C *is defined by the following grammar.*

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

C *denotes a multiset of* objects *(processes and cells), so the join and empty rules form a commutative monoid. However, we also require that an address refers to at most one object in* C. *Lastly, a configuration* F *is* final *iff it only contains (persistent) cells.*

Now, let Γ and Δ be contexts that associate cell addresses to types. The configuration typing judgment given in Figure 3, $\Gamma \vdash C :: \Delta$, means that the objects in C are well-typed with sources in Γ and destinations in Δ (note that we are allowing the process typing judgment to use addresses in place of address variables). Notice that the typing rules preserve the invariant $\Gamma \subseteq \Delta$ thanks to the persistence of memory cells.

$$\frac{\Gamma \vdash^{\omega} P :: (a : A)}{\Gamma \vdash \mathrm{proc}\, a\, P :: (\Gamma, a : A)}\ \mathrm{proc} \qquad \frac{\Gamma \vdash^{\omega} a^{\mathrm{W}}.V :: (a : A)}{\Gamma \vdash !\mathrm{cell}\, a\, V :: (\Gamma, a : A)}\ !\mathrm{cell}_V$$

$$\frac{\Gamma \vdash^{\omega} \mathbf{case}\, a^{\mathrm{W}}\, K :: (a : A)}{\Gamma \vdash !\mathrm{cell}\, a\, K :: (\Gamma, a : A)}\ !\mathrm{cell}_K \qquad \frac{}{\Gamma \vdash \cdot :: \Gamma}\ \mathrm{empty} \qquad \frac{\Gamma \vdash C :: \Gamma' \quad \Gamma' \vdash C' :: \Delta}{\Gamma \vdash C, C' :: \Delta}\ \mathrm{join}$$

Figure 3 Configuration Typing.

Configuration reduction $\to$ is given as *multiset rewriting rules* [17] in Figure 4, which replace any subset of a configuration matching the left-hand side with the right-hand side. However, ! indicates objects that persist across reductions. Principal cuts encountered in a configuration are resolved by passing a value to a continuation also given in Figure 4 as the relation $V \triangleright K = P$.

$$\begin{aligned}
!\mathrm{cell}\, a\, W, \mathrm{proc}\, b\, (b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) &\to !\mathrm{cell}\, b\, W \\
\mathrm{proc}\, c\, (x \leftarrow P(x); Q(x)) &\to \\
\mathrm{proc}\, a\, (P(a)), \mathrm{proc}\, c\, (Q(a)) &\text{ where } a \text{ is fresh} \\
!\mathrm{cell}\, a\, K, \mathrm{proc}\, c\, (a^{\mathrm{R}}.V) &\to \mathrm{proc}\, c\, (V \triangleright K) \\
!\mathrm{cell}\, a\, V, \mathrm{proc}\, c\, (\mathbf{case}\, a^{\mathrm{R}}\, K) &\to \mathrm{proc}\, c\, (V \triangleright K) \\
\mathrm{proc}\, a\, (a \leftarrow f\ \overline{n}\ \overline{b}) &\to \mathrm{proc}\, a\, (P_f(\overline{n}, \overline{b}, a)) \\
\mathrm{proc}\, a\, (a^{\mathrm{W}}.V) &\to !\mathrm{cell}\, a\, V \\
\mathrm{proc}\, a\, (\mathbf{case}\, a^{\mathrm{W}}\, K) &\to !\mathrm{cell}\, a\, K
\end{aligned}$$

$$\begin{aligned}
\langle\rangle \triangleright \langle\rangle &\Rightarrow P = P \\
\langle a, b\rangle \triangleright (\langle x, y\rangle \Rightarrow P(x, y)) &= P(a, b) \\
k\, a \triangleright \{\ell\, x \Rightarrow P_\ell(x)\}_{\ell \in S} &= P_k(a) \\
\langle n, a\rangle \triangleright (\langle i, x\rangle \Rightarrow P(i, x)) &= P(n, a) \\
\langle *, a\rangle \triangleright (\langle *, x\rangle \Rightarrow P(x)) &= P(a)
\end{aligned}$$

Figure 4 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 [37]: 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 populated. The third and fourth rules resolve principal cuts by passing a value to a continuation, whereas the fifth one resolves definition calls. Lastly, the final two rules perform the action of writing to a cell.

Now, we are ready to prove normalization. Relatedly, refer to Das and Pfenning [25] for a proof of type safety for a session type system with arithmetic refinements. In contrast to the normalization proof for base SAX [33], we explicitly construct a model of SAX in sets of terminating configurations, also known as *semantic typing* [5, 38]. This leaves open several possibilities – for example, we could reason about programs that fail to syntactically typecheck [41, 34] or analyze fixed points of semantic type constructors. Our approach mirrors that for natural deduction:

1. We define semantic types: sets of terminating configurations with the necessary properties to prove normalization (see reducibility candidates [36]).
2. We show that semantic versions of the syntactic typing rules of processes, objects, and configurations are admissible in this model.
3. This culminates in a fundamental theorem of the logical relation that translates syntactic types to semantic ones. Weak normalization for *closed* configurations (where $\cdot \vdash C :: \Delta$) is a corollary.
4. Strong normalization of arbitrary configurations (where $\Gamma \vdash C :: \Delta$) is a corollary of the fundamental theorem as well as a weak form of the diamond property [6].

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

▶ **Definition 9** (Semantic type). *A* semantic type $\mathscr{A}, \mathscr{B}, \ldots \in \mathbf{Sem}$ *is a set of pairs of addresses and final configurations, writing* $F \in [a : \mathscr{A}]$ *for* $(a; F) \in \mathscr{A}$*, such that if* $F \in [a : \mathscr{A}]$*, then:*

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

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

Conditions 1 and 2 are required to reproduce the identity rule semantically, but condition 3 is a symptom of working in a concurrent setting: we need to aggregate the semantic type ascriptions of different sub-configurations. In the next definition, we quickly define each semantic type in **boldface** based on its syntactic counterpart.

▶ **Definition 10** (Semantic types).

1. $F \in [a : \mathbb{1}] \triangleq F = F', !\mathsf{cell}\, a\, \langle\rangle$.
2. $F \in [c : \mathscr{A} \otimes \mathscr{B}] \triangleq F = F', !\mathsf{cell}\, c\, \langle a, b\rangle$ *where* $F' \in [a : \mathscr{A}]$ *and* $F' \in [b : \mathscr{B}]$*.*
3. $F \in [c : \mathscr{A} \to \mathscr{B}] \triangleq !\mathsf{cell}\, c\, K \in F$ *for some* K *and* $F, F', \mathrm{proc}\, b\, (c^{\mathrm{R}}.\langle a, b\rangle) \in [\![b : \mathscr{B}]\!]$ *for all* a, b, F' *such that* $F, F' \in [a : \mathscr{A}]$*.*
4. $F \in [b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S}] \triangleq F = F', !\mathsf{cell}\, b\, (k\, a)$ *and* $F' \in [a : \mathscr{A}_k]$ *for some* $k \in S$*.*
5. $F \in [b : \mathbin{\&}\{\ell : \mathscr{A}_\ell\}_{\ell \in S}] \triangleq !\mathsf{cell}\, b\, K \in F$ *for some* K *and* $F, \mathrm{proc}\, a\, (b^{\mathrm{R}}.k\, a) \in [\![a : \mathscr{A}_k]\!]$ *for all* $k \in S, a \in \mathsf{Addr}$*.*

Assume $\mathscr{F} : \mathbb{N} \to \mathbf{Sem}$ *and* ϕ *is a closed constraint.*

1. $F \in [b : \exists \mathscr{F}] \triangleq F = F', !\mathsf{cell}\, b\, \langle n, a\rangle$ *and* $F' \in [a : \mathscr{F}(n)]$*.*
2. $F \in [b : \forall \mathscr{F}] \triangleq F, \mathrm{proc}\, a\, (b^{\mathrm{R}}.\langle n, a\rangle) \in [\![a : \mathscr{F}(n)]\!]$ *for all* $a \in \mathsf{Addr}, n \in \mathbb{N}$*.*
3. $F \in [b : \phi \wedge \mathscr{A}] \triangleq$ *where* $F = F', !\mathsf{cell}\, b\, \langle *, a\rangle$*,* $\cdot; \cdot \vdash \phi$*, and* $F' \in [a : \mathscr{A}]$*.*
4. $F \in [b : \phi \Rightarrow \mathscr{A}] \triangleq$ *if* $\cdot; \cdot \vdash \phi$*, then* $F, \mathrm{proc}\, a\, (b^{\mathrm{R}}.\langle *, a\rangle) \in [\![a : \mathscr{A}]\!]$ *for all* $a \in \mathsf{Addr}$*.*

Positive semantic types are defined by *intension* – the contents of a particular cell – whereas negative semantic types are defined by *extension* – how interacting with a configuration produces the desired result. 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 [36, 4]. Now, to state the semantic typing rules, we need to define the *semantic typing judgment*.

▶ **Definition 11** (Semantic typing judgment). *Let* $\mathbf{\Gamma}$ *and* $\mathbf{\Delta}$ *be contexts associating cell addresses to semantic types.*

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

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

▶ **Lemma 12** (Backward closure). *If* $C \to^* C'$ *and* $\mathbf{\Gamma} \vDash C' :: \mathbf{\Delta}$*, then* $\mathbf{\Gamma} \vDash C :: \mathbf{\Delta}$.

We are finally ready to prove a representative sample of semantic typing rules, all of which are in Figure 6 (the dashed lines indicate that they are admissible rules). Afterwards, we can tackle objects and configurations. As we promised, conditions 1 and 2 are used for the admissibility of the identity rule.

▶ **Lemma 13** (id). $\mathbf{\Gamma}, a : \mathscr{A} \vDash \operatorname{proc} b\,(b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) :: (b : \mathscr{A})$

Proof. Assuming $F \in [\mathbf{\Gamma}, a : \mathscr{A}]$, we want to show $F, \operatorname{proc} b\,(b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) \in [\![b : \mathscr{A}]\!]$. By condition 1, $!\mathrm{cell}\, a\, W \in F$. By condition 2, $F, !\mathrm{cell}\, b\, W \in [b : \mathscr{A}]$. Since $F, \operatorname{proc} b\,(b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) \to F, !\mathrm{cell}\, b\, W$, we are done by Lemma 12. ◀

The reader may have noticed that each semantic type's definition encodes its own noninvertible rule, which makes the admissibility of rules like $\otimes$R immediate. Invertible rules require more effort; consider $\otimes$L below.

▶ **Lemma 14** ($\otimes$L). *If* $\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B} \vdash \operatorname{proc} d\,(P(a, b)) :: (d : \mathscr{C})$*, then* $\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B} \vdash \operatorname{proc} d\,(\mathbf{case}\, c^{\mathrm{R}}\,(\langle x, y\rangle \Rightarrow P(x, y))) :: (d : \mathscr{C})$.

Proof. Assuming $F \in [\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}]$, we want to show that $F, \operatorname{proc} d\,(\mathbf{case}\, c^{\mathrm{R}}\,(\langle x, y\rangle \Rightarrow P(x, y))) \in [\![d : \mathscr{C}]\!]$. Since $F \in [c : \mathscr{A} \otimes \mathscr{B}]$, we have $F = F', !\mathrm{cell}\, c\, \langle a, b\rangle$ where $F' \in [a : \mathscr{A}]$ and $F' \in [b : \mathscr{B}]$. As a result, both $F \in [a : \mathscr{A}]$ and $F \in [b : \mathscr{B}]$ by condition 3. In sum, $F \in [\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B}]$, so by the premise, $F, \operatorname{proc} d\,(P(a, b)) \in [\![d : \mathscr{C}]\!]$. Since $F, \operatorname{proc} d\,(\mathbf{case}\, c^{\mathrm{R}}\,(\langle x, y\rangle \Rightarrow P(x, y))) \to F, \operatorname{proc} d\,(P(a, b))$, we are done by Lemma 12. ◀

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 prove semantic object typing rules, we need a *logical relation* that interprets syntactic types as semantic ones.

▶ **Definition 15** (Logical relation). *We define a logical relation* $(\!|A|\!)_n$ *by lexicographic induction on* (n, A) *that sends arithmetically closed types 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.*

$$(\!|\mathbf{1}|\!)_n \triangleq \mathbb{1} \qquad (\!|A \otimes B|\!)_n \triangleq (\!|A|\!)_n \otimes (\!|B|\!)_n \qquad (\!|A \to B|\!)_n \triangleq (\!|A|\!)_n \to (\!|B|\!)_n$$
$$(\!|\oplus\{\ell : A_\ell\}_{\ell\in S}|\!)_n \triangleq \oplus\{\ell : (\!|A_\ell|\!)_n\}_{\ell\in S} \qquad (\!|\&\{\ell : A_\ell\}_{\ell\in S}|\!)_n \triangleq \mathcal{&}\{\ell : (\!|A_\ell|\!)_n\}_{\ell\in S}$$
$$(\!|X[\overline{m}]|\!)_0 \triangleq \emptyset \qquad (\!|\forall i.\, A(i)|\!)_n \triangleq \forall(\lambda m.\, (\!|A(m)|\!)_n) \qquad (\!|\exists i.\, A(i)|\!)_n \triangleq \exists(\lambda m.\, (\!|A(m)|\!)_n)$$
$$(\!|X[\overline{m}]|\!)_{n+1} \triangleq (\!|A_X[\overline{m}]|\!)_n \quad (\!|\phi \wedge A|\!)_n \triangleq \phi \wedge (\!|A|\!)_n \qquad (\!|\phi \Rightarrow A|\!)_n \triangleq \phi \Rightarrow (\!|A|\!)_n$$

The index n *is merely a technical device for defining the logical relation – it is* not *a step index. Now, let* $F \in (\!|A|\!) \triangleq F \in (\!|A|\!)_n$ *for some* n. $(\!|\cdot|\!)$ *is then extended to contexts* Γ *and* Δ *in the obvious way.*

▶ **Lemma 16** (Semantic object typing).
1. *If* $D \in \Gamma \vdash^\omega a^{\mathrm{W}}.V :: (a : A)$, *then* $(\!|\Gamma|\!) \vDash !\mathrm{cell}\, a\, V :: (a : (\!|A|\!))$.
2. *If* $D \in \Gamma \vdash^\omega \mathbf{case}\, a^{\mathrm{W}}\, K :: (a : A)$, *then* $(\!|\Gamma|\!) \vDash !\mathrm{cell}\, a\, K :: (a : (\!|A|\!))$.
3. *If* $D \in \Gamma \vdash^\omega P :: (a : A)$, *then* $(\!|\Gamma|\!) \vDash \mathrm{proc}\, a\, P :: (a : (\!|A|\!))$.

Proof. Part 1 follows by case analysis on D applying the relevant semantic typing rules, like $\otimes$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 $\to\mathrm{R}^\omega$). In part 3, if P reads a cell (like $\to\mathrm{L}^\omega$ or $\otimes\mathrm{L}^\omega$), then we invoke the relevant semantic typing rule. If P writes a continuation K, then $\mathrm{proc}\, a\, (\mathbf{case}\, a^W\, K) \to !\mathrm{cell}\, a\, K$, so we invoke part 2 on D and conclude by Lemma 12. Writing a value follows symmetrically, invoking part 1. ◀

The reader may have already noticed that there is a disconnect: the conclusions of our semantic object typing rules have a single succedent, e.g., $(\!|\Gamma|\!) \vDash \mathrm{proc}\, a\, P :: (a : (\!|A|\!))$, but its syntactic counterpart factors sources through: $\Gamma \vdash \mathrm{proc}\, a\, P :: (\Gamma, a : A)$. The following lemma recovers this information as a consequence of memory cell persistence.

▶ **Lemma 17** (Recall). *If* $\mathbf{\Gamma} \vDash C :: \mathbf{\Delta}$, *then* $\mathbf{\Gamma} \vDash C :: \mathbf{\Gamma}, \mathbf{\Delta}$.

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

▶ **Lemma 18** (Semantic configuration typing).
1. *Empty:* $\mathbf{\Gamma} \vDash \cdot :: \mathbf{\Gamma}$
2. *Join: If* $\mathbf{\Gamma} \vDash C :: \mathbf{\Gamma}'$ *and* $\mathbf{\Gamma}' \vDash C' :: \mathbf{\Delta}$, *then* $\mathbf{\Gamma} \vDash C, C' :: \mathbf{\Delta}$.

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

▶ **Theorem 19** (Fundamental theorem). *If* $D \in \Gamma \vdash C :: \Delta$, *then* $(\!|\Gamma|\!) \vDash C :: (\!|\Delta|\!)$.

Proof. By induction on D, the empty and join cases are discharged by Lemma 18. The object typing cases are covered by Lemma 16 then Lemma 17. ◀

Strong normalization follows from weak normalization as well as a weak form of the diamond property (as follows) [6]. As we mentioned in the introduction, the latter implies that normalization is independent of how processes are scheduled.

▶ **Theorem 20** (Diamond property). *Let $C_1 \sim C_2$ iff C_1 is equal to C_2 up-to renaming of addresses. Assume $\Gamma \vdash C :: \Delta$, $C \to C_1$, and $C \to C_2$ where $C_1 \not\sim C_2$. Then, $C_1 \to C_1'$ and $C_2 \to C_2'$ such that $C_1' \sim C_2'$.*

Proof. The proof follows that of theorem 10 (the diamond property) in [33], as the only relevant addition is the unfolding of recursive definitions. ◀

▶ **Theorem 21** (Strong normalization). *If $\Gamma \vdash C :: \Delta$, then there are no infinite reduction sequences beginning with C.*

5 Related Work

Our system is closely related to the sequential functional language of Lepigre and Raffalli [45], 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 in the introduction. 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 our system, especially the ability to handle programs not annotated with sizes.

Sized types. Sized types are a type-oriented formulation of size-change termination [44] for rewrite systems [60, 12]. Sized (co)inductive types [8, 10, 2, 4] gave way to sized mixed inductive-coinductive types [3, 4]. In parallel, linear size arithmetic for sized inductive types [19, 64, 11] was generalized to support coinductive types as well [54]. 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 in the introduction, we use unbounded quantification [62] in lieu of transfinite sizes to represent (co)data of arbitrary height and depth. However, the state of the art [3, 4, 18] supports polymorphic, higher-kinded, and dependent types, which we aim to incorporate in future work.

Size inference. Our system keeps constraints implicit but arithmetic data explicit at the process level in agreement with observations made about constraint and arithmetic term reconstruction in a session-typed calculus [27]. On the other hand, systems like $\mathsf{CIC}_{\ell}^{\widehat{}}$ [54] and $\mathsf{CIC}\widehat{*}$ [18] 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 at the moment.

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 [30, 31, 7]. 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 [14, 58], 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 [52]. Severi et al. [56] give a mixed functional and concurrent programming language where corecursive definitions are typed with Nakano's later modality [48]. Since Vezzosi [62] gives an embedding of the later modality and its dual into sized types, we believe that a similar arrangement can be achieved in our setting. In any case, we support recursion schemes more complex than structural (co)recursion [47].

π-calculi. Certain type systems for π-calculi [42, 49, 35] guarantee the eventual success of communication only if or regardless of whether processes diverge [23]. Considering a configuration C such that $\Gamma \vdash C :: (\Gamma, a : X[n])$ where $X[i]$ is a positive coinductive type, we conjecture that $|C|$, which has all constraint and arithmetic data erased, is similarly "productive" even if it may *not* terminate. Intuitively, C writes a number of cells as a function of n then terminates, so $|C|$ represents 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 [29, 28, 20]. 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 $\mathscr{P}$ [55] 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 [67] on graph types [66] is also sufficient. Our system is able to guarantee termination despite utilizing non-linear types, but it remains open how type refinements compare to graph types.

6 Conclusion and Future Work

We have presented a highly general concurrent language that conceives mixed inductive-coinductive programming as a mode of use of arithmetic refinements. Moreover, we prove normalization via a novel logical relations argument in the presence of infinitely deep typing derivations that is mediated through infinitely wide but finitely deep (inductive) typing. There are three main points of interest for future work.

1. *Richer types*: to mix linear [50], affine linear, non-linear, etc. references to memory as well as persistent and ephemeral memory, we conjecture that moving to a type system based on adjoint logic [52] is appropriate. In that case, sizes could be related to the grades of the adjoint modalities [57]. Furthermore, we are interested in generalizing to substructural polymorphic, higher-kinded [24], and dependent types [16, 43].
2. *Implementation*: we are interested in developing a convenient surface language (perhaps a functional one [51]) for SAX and implementing our type system, following Rast [25], an implementation of resource-aware session types that includes arithmetic refinements. Perhaps various validity conditions of infinite proofs can be implemented as implicit uses of sized type refinements.
3. *Message passing*: we would like to transport our results to the asynchronous message passing interpretation of SAX [52], avoiding a technically difficult detour through asynchronous typed π-calculi [32].

References

1 Andreas Abel. Productive infinite objects via copatterns and sized types in agda.

2 Andreas Abel. Semi-continuous Sized Types and Termination. *Logical Methods in Computer Science*, Volume 4, Issue 2, April 2008.

3 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In Dale Miller and Zoltán Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012*, volume 77 of *EPTCS*, pages 1–11, 2012. doi:10.4204/EPTCS.77.1.

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

5 Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, page 247, USA, 2001. IEEE Computer Society.

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

7 David Baelde, Amina Doumane, and Alexis Saurin. Infinitary Proof Theory: the Multiplicative Additive Case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

8 Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comput. Sci.*, 14(1):97–141, 2004. doi:10.1017/S0960129503004122.

9 Henning Basold. *Mixed inductive-coinductive reasoning: types, programs and logic*. PhD thesis, Radboud University Nijmegen, April 2018.

10 Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications*, pages 24–39, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

11 Frédéric Blanqui and Colin Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 105–119, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

12 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*, Coimbra, Portugal, September 2009. Full version.

13 James Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 78–92, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

14 James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, pages 101–112. ACM, 2008.

15 Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

16 Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, page 264, USA, 1996. IEEE Computer Society.

17 Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, 2009. Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

18 Jonathan Chan and William J. Bowman. Practical sized typing for coq. *CoRR*, abs/1912.05601, 2019. arXiv:1912.05601.

19 Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2):261–300, 2001.

20 Ioana Cristescu and Daniel Hirschkoff. Termination in a π-calculus with subtyping. *Mathematical Structures in Computer Science*, 26(8):1395–1432, 2016.

21 Francesco Dagnino. Foundations of regular coinduction. *Logical Methods in Computer Science*, Volume 17, Issue 4, October 2021. `doi:10.46298/lmcs-17(4:2)2021`.

22 Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction, 2009.

23 Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *Journal of Logical and Algebraic Methods in Programming*, 124:100717, 2022.

24 Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Subtyping on Nested Polymorphic Session Types, 2021. `arXiv:2103.15193`.

25 Ankush Das and Frank Pfenning. Rast: A Language for Resource-Aware Session Types, 2020. `arXiv:2012.13129`.

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

27 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, New York, NY, USA, 2020. Association for Computing Machinery.

28 Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in impure concurrent languages. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 328–342, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

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

30 Farzaneh Derakhshan and Frank Pfenning. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *CoRR*, abs/1908.01909, August 2019.

31 Farzaneh Derakhshan and Frank Pfenning. Circular Proofs in First-Order Linear Logic with Least and Greatest Fixed Points. *CoRR*, abs/2001.05132, January 2020.

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

33 Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-Axiomatic Sequent Calculus. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

34 Derek Dreyer, Amin Timany, Robbert Krebbers, Lars Birkedal, and Ralf Jung. What Type Soundness Theorem Do You Really Want to Prove?, October 2019.

35 Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, pages 63–77, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

36 Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989.

37 Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

38 Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In *Proceedings of the 10th ACM SIGPLAN Interna-*

tional Conference on Certified Programs and Proofs, CPP 2021, pages 178–198, New York, NY, USA, 2021. Association for Computing Machinery.

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

40 Aleksandar Ignjatovic. Hilbert's program and the omega-rule, June 2018.

41 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

42 Naoki Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, pages 233–247, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

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

44 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *SIGPLAN Not.*, 36(3):81–92, January 2001.

45 Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for Curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1), February 2019.

46 Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers, USA, 2004.

47 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. *SIGPLAN Not.*, 51(9):434–447, September 2016.

48 Hiroshi Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

49 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, New York, NY, USA, 2014. Association for Computing Machinery.

50 Frank Pfenning. Types and Programming Languages, 2020.

51 Klaas Pruiksma and Frank Pfenning. Back to Futures. *CoRR*, abs/2002.04607, February 2020.

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

53 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

54 Jorge Luis Sacchini. Linear Sized Types in the Calculus of Constructions. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 169–185, Cham, 2014. Springer International Publishing.

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

56 Paula Severi, Luca Padovani, Emilio Tuosto, and Mariangiola Dezani-Ciancaglini. On Sessions and Infinite Data. In Alberto Lluch Lafuente and José Proença, editors, *Coordination Models and Languages*, pages 245–261, Cham, 2016. Springer International Publishing.

57 Siva Somayyajula. Towards Unifying (Co)induction and Structural Control. In *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*, Rome (virtual), Italy, June 2021.

58 Gadi Tellez and James Brotherston. Automatically verifying temporal properties of pointer programs with cyclic proof. *Journal of Automated Reasoning*, 64(3):555–578, 2020. doi:10.1007/s10817-019-09532-0.

59 The Coq Development Team. The Coq Proof Assistant, January 2021.

60 René Thiemann and Jürgen Giesl. Size-change termination for term rewriting. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, pages 264–278, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

61 Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 350–369, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

62 Andrea Vezzosi. Total (Co)Programming with Guarded Recursion. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, pages 77–78, Tallinn, Estonia, 2015. Institute of Cybernetics at Tallinn University of Technology.

63 Philip Wadler. Propositions as sessions. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.

64 Hongwei Xi. Dependent types for program termination verification. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 231–242, 2001.

65 Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.

66 Nobuko Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 371–386, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

67 Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the π-calculus. *Information and Computation*, 191(2):145–202, 2004.

A Appendix

▶ **Example 22** (Typechecking)**.** The process definition below, whose type signature is $i; \cdot; x : \mathrm{nat}[i] \vdash^i y \leftarrow \mathrm{eat}\ \ i\ x :: (y : \mathbf{1})$, traverses a unary natural number by induction to produce a unit. Recall $\mathrm{nat}[i] = \oplus\{\mathrm{zero} : \mathbf{1}, \mathrm{succ} : i > 0 \wedge \mathrm{nat}[i-1]\}$.

$$y \leftarrow \mathrm{eat}\ \ i\ x = \mathbf{case}\, x^{\mathrm{R}}\, \{\, \mathrm{zero}\ z \Rightarrow y^{\mathrm{W}} \leftarrow z^{\mathrm{R}}, \mathrm{succ}\ z \Rightarrow y \leftarrow \mathrm{eat}\ \ (i-1)\ z\}$$

Now, let us construct a typing derivation of its body below.

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

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

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

$$\frac{}{\Gamma \vdash^{\omega} x^{\mathrm{W}}.\langle\rangle :: (x : \mathbf{1})}\ \mathbf{1}\mathrm{R}^{\omega} \qquad \frac{\Gamma, x : \mathbf{1} \vdash^{\omega} P :: (z : C)}{\Gamma, x : \mathbf{1} \vdash^{\omega} \mathbf{case}\, x^{\mathrm{R}}\, (\langle\rangle \Rightarrow P) :: (z : C)}\ \mathbf{1}\mathrm{L}^{\omega}$$

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

$$\frac{\Gamma, y : A \vdash^{\omega} P(y, z) :: (z : B)}{\Gamma \vdash^{\omega} \mathbf{case}\, x^{\mathrm{W}}\, (\langle y, z\rangle \Rightarrow P(y, z)) :: (x : A \to B)}\ \to\mathrm{R}^{\omega} \qquad \frac{}{\Gamma, x : A \to B, y : A \vdash^{\omega} x^{\mathrm{R}}.\langle y, z\rangle :: (z : B)}\ \to\mathrm{L}^{\omega}$$

$$\frac{k \in S}{\Gamma, y : A_k \vdash^{\omega} x^{\mathrm{W}}.k\, y :: (x : \oplus\{\ell : A_\ell\}_{\ell \in S})}\ \oplus\mathrm{R}^{\omega} \qquad \frac{\{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S}, y : A_\ell \vdash^{\omega} P(y) :: (z : C)\}_{\ell \in S}}{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S} \vdash^{\omega} \mathbf{case}\, x^{\mathrm{R}}\, \{\ell\, y \Rightarrow P_\ell(y)\}_{\ell \in S} :: (z : C)}\ \oplus\mathrm{L}^{\omega}$$

$$\frac{\{\Gamma \vdash^{\omega} P(y) :: (y : A_\ell)\}_{\ell \in S}}{\Gamma \vdash^{\omega} \mathbf{case}\, x^{\mathrm{W}}\, \{\ell\, y \Rightarrow P_\ell(y)\}_{\ell \in S} :: (x : \&\{\ell : A_\ell\}_{\ell \in S})}\ \&\mathrm{R}^{\omega} \qquad \frac{k \in S}{\Gamma, x : \&\{\ell : A_\ell\}_{\ell \in S} \vdash^{\omega} x^{\mathrm{R}}.k\, y :: (y : A_k)}\ \&\mathrm{L}^{\omega}$$

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

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

$$\frac{\cdot ; \cdot \vdash \phi}{\Gamma, y : A \vdash^{\omega} x^{\mathrm{W}}.\langle *, y\rangle :: (x : \phi \wedge A)}\ \wedge\mathrm{R}^{\omega} \qquad \frac{\Gamma, x : \phi \wedge A, y : A \vdash^{\omega} P(y) :: (z : C) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \wedge A \vdash^{\omega} \mathbf{case}\, x^{\mathrm{R}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (z : C)}\ \wedge\mathrm{L}^{\omega}$$

$$\frac{\Gamma \vdash^{\omega} P(y) :: (y : A) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma \vdash^{\omega} \mathbf{case}\, x^{\mathrm{W}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (x : \phi \Rightarrow A)}\ \Rightarrow\mathrm{R}^{\omega} \qquad \frac{\cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \Rightarrow A \vdash^{\omega} x^{\mathrm{R}}.\langle *, y\rangle :: (y : A)}\ \Rightarrow\mathrm{L}^{\omega}$$

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

Figure 5 SAX$^{\omega}$ Typing Rules.

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) :: (b : \mathscr{A})}\ \mathrm{id} \qquad \frac{\mathbf{\Gamma} \vDash \mathrm{proc}\, a\, P :: (a : \mathscr{A}) \quad \mathbf{\Gamma}, a : A \vDash \mathrm{proc}\, c\, (Q(a)) :: (c : \mathscr{C})}{\mathbf{\Gamma} \vDash \mathrm{proc}\, c\, (x \leftarrow P; Q(x)) :: (c : \mathscr{C})}\ \mathrm{cut}$$

$$\frac{}{\mathbf{\Gamma} \vDash !\mathrm{cell}\, a\, \langle\rangle :: (a : \mathbb{1})}\ \mathbb{1}\mathrm{R} \qquad \frac{\mathbf{\Gamma}, a : \mathbb{1} \vDash \mathrm{proc}\, b\, P :: (b : \mathscr{C})}{\mathbf{\Gamma}, a : \mathbb{1} \vDash \mathrm{proc}\, b\, (\mathbf{case}\, a^{\mathrm{R}}\, (\langle\rangle \Rightarrow P)) :: (b : \mathscr{C})}\ \mathbb{1}\mathrm{L}$$

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{A}, b : \mathscr{B} \vDash !\mathrm{cell}\, c\, \langle a, b\rangle :: (c : \mathscr{A} \otimes \mathscr{B})}\ \otimes\mathrm{R} \qquad \frac{\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B} \vdash \mathrm{proc}\, d\, (P(a, b)) :: (d : \mathscr{D})}{\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B} \vdash \mathrm{proc}\, d\, (\mathbf{case}\, c^{\mathrm{R}}\, (\langle x, y\rangle \Rightarrow P(x, y))) :: (d : \mathscr{D})}\ \otimes\mathrm{L}$$

$$\frac{\mathbf{\Gamma}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (P(a, b)) :: (b : \mathscr{B})}{\mathbf{\Gamma} \vDash !\mathrm{cell}\, c\, (\langle x, y\rangle \Rightarrow P(x, y)) :: (c : \mathscr{A} \to \mathscr{B})}\ \to\mathrm{R} \qquad \frac{}{\mathbf{\Gamma}, c : \mathscr{A} \to \mathscr{B}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (c^{\mathrm{R}}.\langle a, b\rangle) :: (b : \mathscr{B})}\ \to\mathrm{L}$$

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{A}_k \vDash !\mathrm{cell}\, b\, (k\, a) :: (b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S})}\ \oplus\mathrm{R} \qquad \frac{\{\mathbf{\Gamma}, b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S}, a : \mathscr{A}_k \vDash \mathrm{proc}\, c\, (P_k(a)) :: (c : \mathscr{C})\}_{k \in S}}{\mathbf{\Gamma}, b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S} \vDash \mathrm{proc}\, c\, (\mathbf{case}\, b^{\mathrm{R}}\, \{\ell\, x \Rightarrow P_\ell(x)\}_{\ell \in S}) :: (c : \mathscr{C})}\ \oplus\mathrm{L}$$

$$\frac{\{\mathbf{\Gamma} \vDash \mathrm{proc}\, a\, (P_\ell(a)) :: (a : \mathscr{A}_\ell)\}_{\ell \in S}}{\mathbf{\Gamma} \vDash !\mathrm{cell}\, b\, \{\ell\, x \Rightarrow P_\ell(x)\}_{\ell \in S} :: (b : \&\{\ell : \mathscr{A}_\ell\}_{\ell \in S})}\ \&\mathrm{R} \qquad \frac{}{\mathbf{\Gamma}, b : \&\{\ell : \mathscr{A}_\ell\}_{\ell \in S} \vDash \mathrm{proc}\, b\, (c^{\mathrm{R}}.k\, a) :: (a : \mathscr{A})}\ \&\mathrm{L}$$

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{F}(n) \vDash !\mathrm{cell}\, b\, \langle n, a\rangle :: (b : \exists \mathscr{F})}\ \exists\mathrm{R} \qquad \frac{\{\mathbf{\Gamma}, b : \exists \mathscr{F}, a : \mathscr{F}(n) \vDash \mathrm{proc}\, c\, (P(n, a)) :: (c : \mathscr{C})\}_{n \in \mathbb{N}}}{\mathbf{\Gamma}, b : \exists \mathscr{F} \vDash \mathrm{proc}\, c\, (\mathbf{case}\, b^{\mathrm{R}}\, (\langle i, x\rangle \Rightarrow P(i, x))) :: (c : \mathscr{C})}\ \exists\mathrm{L}$$

$$\frac{\{\mathbf{\Gamma} \vDash \mathrm{proc}\, a\, (P(n, a)) :: (a : \mathscr{F}(n))\}_{n \in \mathbb{N}}}{\mathbf{\Gamma} \vDash !\mathrm{cell}\, b\, (\langle i, x\rangle \Rightarrow P(i, x)) :: (b : \forall \mathscr{F})}\ \forall\mathrm{R} \qquad \frac{}{\mathbf{\Gamma}, b : \forall \mathscr{F} \vDash \mathrm{proc}\, a\, (b^{\mathrm{R}}.\langle n, a\rangle) :: (a : \mathscr{F}(n))}\ \forall\mathrm{L}$$

$$\frac{\cdot; \cdot \vdash \phi}{\mathbf{\Gamma}, b : A \vDash !\mathrm{cell}\, a\, \langle *, b\rangle :: (a : \phi \wedge \mathscr{A})}\ \wedge\mathrm{R} \qquad \frac{\mathbf{\Gamma}, a : \phi \wedge \mathscr{A}, b : \mathscr{A} \vdash \mathrm{proc}\, a\, (P(b)) :: (c : \mathscr{C}) \quad \text{if } \cdot; \cdot \vdash \phi}{\mathbf{\Gamma}, a : \phi \wedge \mathscr{A} \vDash \mathrm{proc}\, a\, (\mathbf{case}\, a^{\mathrm{R}}\, (\langle *, y\rangle \Rightarrow P(y))) :: (c : \mathscr{C})}\ \wedge\mathrm{L}$$

$$\frac{\mathbf{\Gamma} \vDash \mathrm{proc}\, b\, (P(b)) :: (b : \mathscr{A}) \text{ if } \cdot; \cdot \vdash \phi}{\mathbf{\Gamma} \vDash !\mathrm{cell}\, a\, (\langle *, y\rangle \Rightarrow P(y)) :: (a : \phi \Rightarrow \mathscr{A})}\ \Rightarrow\mathrm{R} \qquad \frac{\cdot; \cdot \vdash \phi}{\mathbf{\Gamma}, a : \phi \Rightarrow \mathscr{A} \vDash \mathrm{proc}\, b\, (a^{\mathrm{R}}.\langle *, b\rangle) :: (b : \mathscr{A})}\ \Rightarrow\mathrm{L}$$

(no rule for **impossible**)

$$\frac{y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y) \quad \overline{b} : \overline{\mathscr{A}} \vDash \mathrm{proc}\, a\, (P_f(\overline{n}, \overline{b}, a)) :: (a : \mathscr{A})}{\mathbf{\Gamma}, \overline{b} : \overline{\mathscr{A}} \vDash \mathrm{proc}\, a\, (a \leftarrow f\ \overline{n}\ \overline{b}) :: (a : \mathscr{A})}\ \mathrm{call}$$

Figure 6 Semantic Object Typing Rules.