Submitted to MFPS 2023

Dependent Type Refinements for Futures*

Siva Somayyajula[1] Frank Pfenning[2]

Computer Science Department
Carnegie Mellon University
Pittsburgh, USA

Abstract

Type refinements combine the compositionality of typechecking with the expressivity of program logics, offering a synergistic approach to program verification. In this paper we apply dependent type refinements to SAX, a futures-based process calculus that arises from the Curry-Howard interpretation of the intuitionistic semi-axiomatic sequent calculus and includes unrestricted recursion both at the level of types and processes. With our type refinement system, we can reason about the partial correctness of SAX programs, complementing prior work on sized type refinements that supports reasoning about termination. Our design regime synthesizes the infinitary proof theory of SAX with that of bidirectional typing and Hoare logic, deriving some standard reasoning principles for data and (co)recursion while enabling information hiding for codata. We prove syntactic type soundness, which entails a notion of partial correctness that respects codata encapsulation. We illustrate our language through a few simple examples.

Keywords: futures, type refinements, partial correctness, sequent calculus

1 Introduction

Type refinements internalize assertions into the type structure of functional programs, combining the compositionality of typechecking with the expressivity of program logics. While having comparable verification capabilities to "traditional" dependent type theories [71], *dependent type refinements* can also have their assertion logics extended for domain-specific verification [73]. This line of work has also revealed close connections between the proof theory of Hoare logic and that of static type refinement discipline—some correspondences include path-sensitive elimination rules to the conditional rule, substitution to composition, and subsumption to consequence.

Recent work on *dependent session types* [63] have begun to transport related results to process calculi, yet are limited by the need to carefully interface the linearity inherent in session types with type dependency. Solutions tending towards traditional type theory marry a separate dependently-typed proof language with a session-typed process calculus [67], whereas those in the space of type refinements are typically limited to an index language incapable of expressing the complete nuances of process dynamics [65]. This raises the question: is it possible to stake out a middle ground and adapt expressive dependent type refinements to a process calculus? In this article, we develop such type refinements within SAX, a futures-based process calculus that arises from the Curry-Howard interpretation of the intuitionistic

*
[1] Email: ssomayya@cs.cmu.edu
[2] Email: fp@cs.cmu.edu

semi-axiomatic sequent calculus [22]. Thus, we are not bound by the constraints of linearity. However, two major questions generate the design space:

(i) What is a suitable program logic and to what extent should/can it account for process dynamics, including higher-order (co)data?

(ii) How do types internalize assertions and how should the type system be presented?

A core desideratum typically forces certain answers to these questions—for example, to ensure decidable typechecking, *liquid types* [54] are designed around the following answers:

(i) Assertions are quantifier-free, with quantifiers encoded as dependent types [71, Section 3]. Functions and their extensional equality are encoded by first-order approximation [71, Sections 4 and 5] and typeclasses [68], respectively, in contrast to higher-order program logics [53].

(ii) Refined types are distinguished type constructors and typing is *bidirectional* [51,23].

Our guiding principle is to follow the proof theory of the semi-axiomatic sequent calculus, answering these questions as follows:

(i) We develop a first-order theory of SAX values, function applications, and lazy record projections. In particular, we *avoid* directly reifying processes into the assertion logic.

(ii) We use bidirectional typing from a *verificationist* [43] point of view not to effect algorithmic typechecking, but to determine the shape of types and their associated typing rules. In particular, we synthesize the semi-axiomatic sequent calculus with Hoare logic, viewing the former as an intermediate point between (bidirectional) natural deduction and the (unidirectional) sequent calculus, with the right/introduction rules guiding the definition of types. While we recover the expected properties for positive (data) types, curiously, refinements nested under a negative (codata) type may hide information from assertions attached to said type, providing a facility for codata *encapsulation*.

Considering recursion both at the level of types and processes, our type system establishes partial correctness, complementing *sized type refinements* used to guarantee termination in SAX [59]. Following *op. cit.*, we view (sub)typing derivations for equirecursive types [36] and recursive programs, respectively, as *infinite proofs* [9,20] generated by *mixed inductive-coinductive* inference systems [3,14]. Standard *assume-guarantee reasoning* for recursive programs that arises from typing derivation circularity, when combined with subsumption, uniformly admits reasoning with induction, coinduction, and mixed induction and coinduction within the language.

In summary, our primary contribution is Dependent Refined SAX (DRSAX): a dependent type refinement system for SAX (Section 2) with a corresponding type soundness result entailing *observable* partial correctness (Section 3). Essentially, we show that assertions about objects that are not encapsulated hold directly. Our secondary contribution is the design regime listed above, which leads us to a novel derivation of codata encapsulation and a uniform consideration of induction, coinduction, and mixed induction and coinduction within DRSAX.

2 DRSAX: Semi-Axiomatic Type Theory Meets Hoare Logic

In this section, we develop DRSAX by first commenting on the judgmental structure of the semi-axiomatic sequent extended for our purposes. After quickly reviewing some ancillary definitions, we examine the relevant typing rules with examples.

2.1 Judgmental Structure

In short, the semi-axiomatic sequent calculus replaces the typical right and left rules for positive and negative types (i.e., the *non-invertible* ones), respectively, with axioms. The corresponding typing judgment for processes takes on the following form:

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

The *process* P may perform blocking reads from *source addresses* $x, \dots, y$ of *futures* and must perform a non-blocking write exactly once to the future addressed by the *destination* z according to *types* $A, \dots, B$ and C, respectively, corresponding to asynchronous communication with futures [26]. Data addressed are values and process continuations of *positive* and *negative* type, respectively, recalling the binary term and type distinction of call-by-push-value [41]. To make the jump to Hoare logic, we attach *preconditions* to the antecedents and a *postcondition* to the succedent with *refined types*: the judgment becomes $x : \mathbf{A}$ where $\mathbf{A}$ has the form $A \mid \lambda x.\, \phi(x)$. $\phi, \psi, \chi, \dots$ are *assertions* from the external logical theory described in the next sub-section. As shorthand, we write $x : A \mid \phi(x)$ to conflate the antecedent or succedent variable with that bound in ϕ or $\phi(\cdot)$ when the x need not be mentioned.

$$x : \mathbf{A}, \dots, y : \mathbf{B}(x, \dots) \vdash P(x, \dots, y, z) \div (z : \mathbf{C}(x, \dots, y))$$

We encounter questions immediately—for example, considering disjunction as a labelled sum type $\oplus\{\ell : A_\ell\}_{\ell \in S}$, which of the following is a more appropriate right axiom? Note that any non-refined type A is canonically the refined type $A \mid \top$.

$$\oplus\mathrm{R},\ k \in S \quad \Gamma, x : A_k \vdash \dots \div (y : \oplus\{\ell : A_\ell\}_{\ell \in S} \mid y \equiv k \cdot x) \qquad \text{vs.} \qquad \oplus\mathrm{R},\ k \in S \quad \Gamma, x : A_k \mid \phi(k \cdot x) \vdash \dots \div (y : \oplus\{\ell : A_\ell\}_{\ell \in S} \mid \phi(y))$$

The framework for bidirectional typing in [24,23] tells us which: the principal judgment's data in a positive introduction rule are always *inputs*. Thus, the second rule is clearly canonical, as $\oplus\{\ell : A_\ell\}_{\ell \in S}$ and $\phi(y)$ are inputs unlike in the first rule. Peculiarly, A_k and $\phi(k \cdot x)$ are *outputs*, indicating that type information flows from right to left, like in *backwards bidirectional typing* [74,23]. In particular, this rule simulates bottom-up flow in natural deduction:

$$\frac{\oplus\mathrm{I},\ k \in S \qquad \Gamma \vdash \dots : A_k}{\Gamma \vdash \dots : \oplus\{\ell : A_\ell\}_{\ell \in S}}$$

Dually, left axioms for negative types flow from left to right, corresponding to the top-to-bottom flow for negative eliminations in natural deduction. Consider the following example of negative conjunction as a lazy record type (omitting refinements in the axiom for the moment).

$$\frac{\&\mathrm{E},\ k \in S \qquad \Gamma \vdash \dots : \&\{\ell : A_\ell\}_{\ell \in S}}{\Gamma \vdash \dots : A_k} \quad \rightsquigarrow \quad \&\mathrm{L},\ k \in S \quad \Gamma, y : \&\{\ell : A_\ell\}_{\ell \in S} \vdash \dots \div (x : A_k)$$

Taking a step back and thinking of a (non-semi-axiomatic) sequent calculus as an algorithmic type system, the resulting hypothetical judgment would appear as follows:

$$x \Rightarrow A, \dots, y \Rightarrow B \vdash P(x, \dots, y, z) \div (z \Leftarrow C)$$

where the arrows $\Rightarrow$ and $\Leftarrow$ distinguish between antecedent and succedent judgments, respectively, all of which are inputs. Thus, we would have *unidirectional* process typing from the bottom up. As we have seen however, the semi-axiomatic sequent inherits some bidirectionality from natural deduction, in which both antecedents and the succedent may be outputs. Thus, we must allow $\Leftarrow$ and $\Rightarrow$ judgments to respectively appear in Γ and the succedent as outputs. We call the resulting hypothetical judgment(s) *process typing*, defined in Figure 1 along with ancillary judgments (wellformedness judgments are standard and omitted).

2.2 Syntax

We briefly comment on the syntax for addresses, types, and processes in Figure 2.

- We distinguish between *address variables* introduced in the previous subsection and *runtime addresses* which only appear in Section 3.

judgments	$J := (x \Leftarrow A \mid \phi(x)) \mid (x \Rightarrow B \mid \phi(x))$
contexts	$\Gamma := \cdot \mid \Gamma, J$
assertion sequent	$\Gamma \vdash \phi$
subtyping	$\Gamma \vdash A \leq B$
process typing	$\Gamma \vdash P \div J$

Fig. 1. Judgments

$A :=$	$A^+ \mid A^-$		
$\mid$	X	recursive type ($X = A$)	
$\mathbf{A} :=$	$(A \mid \lambda x.\, \phi(x))$	refined type	
$A^+ :=$	$\mathbf{1}$	(positive) unit	
$\mid$	$(x : \mathbf{A}) \otimes B(x)$	dependent eager pairs	
$\mid$	$\oplus\{\ell : A_\ell\}_{\ell \in S}$	eager sums	
$A^- :=$	$(x : \mathbf{A}) \to \mathbf{B}(x)$	dependent functions	
$\mid$	$\&\{\ell : \mathbf{A}_\ell\}_{\ell \in S}$	lazy records	

(a) Types

$s, t :=$	$x, y, z, \ldots$	address variables	
$\mid$	$a, b, c, \ldots$	runtime addresses	
$V :=$	$\langle\rangle$	($\mathbf{1}$R)	
$\mid$	$\langle s, t\rangle$	($\otimes$R, $\to$L)	
$\mid$	$\ell \cdot t$	($\oplus$R, &L)	
$K :=$	$\langle\rangle \Rightarrow P$		($\mathbf{1}$L)
$\mid$	$\langle x, y\rangle \Rightarrow P(x, y)$		($\otimes$L, $\to$R)
$\mid$	$\{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell \in S}$		($\oplus$L, &R)

(b) Values and Continuations

$P, Q :=$	$t \leftarrow s$	copy contents of s to t	(identity)
$\mid$	$x \leftarrow P(x); Q(x)$	spawn P writing to x, proceed concurrently as Q	(cut)
$\mid$	$t.V$	write V to t, or pass V to continuation in t	(positive right/negative left rule)
$\mid$	$\mathbf{case}\, t\, K$	pass value in t to K or, write K to t	(negative right/positive left rule)
$\mid$	$(x : \mathbf{A})\ \mathbf{in}\ P$	refined type annotation	(annoL/R)
$\mid$	$f(\overline{s}, t)$	definition call	$(f(\overline{x : \mathbf{A}}, y : \mathbf{C}(\overline{x})) = P(\overline{x}, y))$

(c) Processes

Fig. 2. Syntax

- In addition to labelled sum and lazy record types, we have dependent eager pair, unit, and function types that seem asymmetrically presented—we elaborate on this later.
- With the exception of type annotation and definition calls, which belong to an ambient signature of typed mutually recursive definitions, each process corresponds to a judgmental or logical rule in the semi-axiomatic sequent calculus. Note that $\overline{x : \mathbf{A}}$ is a *telescope* [18], i.e., a context where each subsequent binding may refer to previously bound variables.

Now, the following sub-sections elaborate on the language, starting with the assertion logic, judgmental rules, logical rules, and ending with recursive definitions.

2.3 Assertion Logic

Assertions, given by the grammar below, are drawn from the (classical) first-order theory of equality with uninterpreted functions. The ellipsis indicates the potential for extension to effect richer verification—for example, including the theory of arithmetic enables termination checking [59].

$$\begin{aligned} \phi, \psi &:= \bot \mid \top \mid M \equiv N \mid \mathrm{is}_k(M) \mid \phi \wedge \psi \mid \phi \supset \psi \mid \forall x.\, \phi(x) \mid \dots \\ M, N &:= s \mid \underbrace{\langle\rangle}_{\mathbf{1}\mathrm{R}} \mid \underbrace{\langle M, N\rangle}_{\otimes\mathrm{R}} \mid \underbrace{M`N}_{\to\mathrm{L}} \mid \underbrace{k \cdot M}_{\oplus\mathrm{R}} \mid \underbrace{M.k}_{\&\mathrm{L}} \end{aligned}$$

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

- *Positive right axioms*: $\langle\rangle$ is a unit value, $\langle M, N\rangle$ is a pair of values M and N, and $k \cdot M$ is a k-tagged value M. These function symbols are additionally subject to the first-order theory of *non-cyclic* data structures [50]. Note that even with recursive types, values cannot be cyclic, because a non-allocating process cannot write to and read from the same address. For convenience, we assume the availability of the assertion $\mathrm{is}_k(M)$ that is true when M is a k-tagged value.
- *Negative left axioms*: $M`N$ represents an application of the SAX function denoted by M to argument N and $M.k$ is the k^{th} projection of the record denoted by M. Note our use of the phrase "the [continuation] denoted by M"—we do *not* directly encode function bodies into the assertion logic, as that could reveal information hidden by negative type refinements discussed in Sections 2.6 and 2.7. Instead, a continuation addressed by s is abstractly described by assertions about $s`N$ or $M.k$—similar to *copattern matching* [1].

2.4 Phase Change: Subsumption and Type Annotation

Analogous to natural deduction, changes of phase between inputs and outputs are mediated by *subsumption* ($\leq$R/L) and *type annotation* (AnnoR/L). The judgmental distinction between the left- and right-hand sides of the sequent require two rules each [39].

$$\frac{\Gamma \vdash P \div (y \Rightarrow \mathbf{A}) \qquad \Gamma \vdash \mathbf{A} \leq \mathbf{B}}{\Gamma \vdash P \div (y \Leftarrow \mathbf{B})}\ \leq\mathrm{R} \qquad \frac{\Gamma, x \Leftarrow \mathbf{B} \vdash P \div (z \Leftarrow \mathbf{C}) \qquad \Gamma \vdash \mathbf{A} \leq \mathbf{B}}{\Gamma, x \Rightarrow \mathbf{A} \vdash P \div (z \Leftarrow \mathbf{C})}\ \leq\mathrm{L}$$

$$\frac{\Gamma \vdash P \div (y \Leftarrow \mathbf{A})}{\Gamma \vdash (y : \mathbf{A})\ \mathbf{in}\ P \div (y \Rightarrow \mathbf{A})}\ \mathrm{annoR} \qquad \frac{\Gamma, x \Rightarrow \mathbf{A} \vdash P \div (z \Leftarrow \mathbf{C})}{\Gamma, x \Leftarrow \mathbf{A} \vdash (x : \mathbf{A})\ \mathbf{in}\ P \div (z \Leftarrow \mathbf{C})}\ \mathrm{annoL}$$

Subsumption combines covariant succedent and contravariant antecedent subtyping with *postcondition strengthening* and *precondition weakening*, analogous to the consequence rule in Hoare logic [31]. The subtyping rules in Figure 3 generalize polarized subtyping (which includes *width* and *depth* subtyping for sums and records [36]) to internalize type dependency [2]. In particular:

- Introducing the auxiliary judgment $\Gamma \vdash \mathbf{A} \leq \mathbf{B}$, $\leq$Pred corresponds to the standard *predicate subtyping* [55] rule at the core of type refinement systems [31].
- The ∞ sign surrounding the premises of $\leq$RecR/L indicates a coinductive occurrence of the subtyping judgment (with all other ones being inductive) [13,14]; *ops. cit.* themselves build on coinductive axiomatizations of subtyping [8]. That is, a subtyping derivation is a (potentially) infinitely deep tree where every infinite branch passes through an instance of this rule infinitely many times, representing the unfolding of a recursive type. See Example 6 in [36] for one such derivation.

$$\frac{\Gamma \vdash A \leq B \qquad \Gamma, x \Rightarrow A \mid \phi(x) \vdash \psi(x)}{\Gamma \vdash (A \mid \phi(\cdot)) \leq (B \mid \psi(\cdot))}\ \leq\textsc{Pred} \qquad \frac{X = A \qquad \infty(\Gamma \vdash A \leq B)}{\Gamma \vdash X \leq B} \qquad \frac{X = A \qquad \infty(\Gamma \vdash B \leq A)}{\Gamma \vdash B \leq X}$$

$$\frac{\Gamma \vdash \mathbf{A} \leq \mathbf{A}' \qquad \Gamma, x \Rightarrow \mathbf{A} \vdash \mathbf{B}(x) \leq \mathbf{B}'(x)}{\Gamma \vdash (x : \mathbf{A}') \to \mathbf{B}(x) \leq (x : \mathbf{A}) \to \mathbf{B}'(x)}\ \leq\!\to \qquad \frac{}{\Gamma \vdash \mathbf{1} \leq \mathbf{1}}\ \leq\!\mathbf{1} \qquad \frac{\Gamma \vdash \mathbf{A} \leq \mathbf{A}' \qquad \Gamma, x \Rightarrow \mathbf{A} \vdash C(x) \leq D(x)}{\Gamma \vdash (x : \mathbf{A}) \otimes C(x) \leq (x : \mathbf{A}') \otimes D(x)}\ \leq\!\otimes$$

$$\frac{S \subseteq T \qquad \{\Gamma \vdash A_\ell \leq B_\ell\}_{\ell \in S}}{\Gamma \vdash \oplus\{\ell : A_\ell\}_{\ell \in S} \leq \oplus\{\ell : B_k\}_{k \in T}}\ \leq\!\oplus \qquad \frac{T \subseteq S \qquad \{\Gamma \vdash \mathbf{A}_k \leq \mathbf{B}_k\}_{k \in T}}{\Gamma \vdash \&\{\ell : \mathbf{A}_\ell\}_{\ell \in S} \leq \&\{\ell : \mathbf{B}_k\}_{k \in T}}\ \leq\!\&$$

Fig. 3. Subtyping

To finish, we verify that the subtyping relation is indeed reflexive and transitive via mixed induction and coinduction.

Remark 2.1 (Mixed Induction and Coinduction) *Proofs by "mixed induction and coinduction" over the structure of (sub)typing derivations involve a lexicographic guarded coinduction to prove judgments marked ∞ prioritized over a structural induction on (smaller) subderivations (i.e., when guardedness does not change). Refer to [13] for further examples.*

Lemma 2.2 (Reflexivity and Transitivity of Subtyping)

- *Reflexivity:* $\Gamma \vdash A \leq A$
- *Transitivity: if* $\Gamma \vdash A \leq B$ *and* $\Gamma \vdash B \leq C$*, then* $\Gamma \vdash A \leq C$

Proof. The first part is by a lexicographic combination of guarded coinduction to prove any instances of the ∞-marked subtyping judgment prioritized over structural induction on A, and the second is a straightforward mixed induction and coinduction on the first derivation and then inversion on the second. □

2.5 Cut, Snips, and Identity

The process behind the cut rule forms the core of computation with futures in (DR)SAX: $x \leftarrow P(x); Q(x)$ spawns P to perform a non-blocking write to a newly allocated future addressed by x while concurrently proceeding as Q, which may perform a blocking read from x. We give two forms of the cut rule depending on which premise outputs the cut "formula" $\mathbf{A}$. Thinking of $\mathbf{A}$ as a *midcondition*, cuts are analogous to the composition rule in Hoare logic.

$$\frac{\Gamma \vdash P(x) \div (x \Leftarrow \mathbf{A}) \qquad \Gamma, x \Leftarrow \mathbf{A} \vdash Q(x) \div (z \Leftarrow \mathbf{C})}{\Gamma \vdash x \leftarrow P(x); Q(x) \div (z \Leftarrow \mathbf{C})}\ \textsc{snip}^+ \qquad \frac{\Gamma \vdash P(x) \div (x \Rightarrow \mathbf{A}) \qquad \Gamma, x \Rightarrow \mathbf{A} \vdash Q(x) \div (z \Leftarrow \mathbf{C})}{\Gamma \vdash x \leftarrow P(x); Q(x) \div (z \Leftarrow \mathbf{C})}\ \textsc{snip}^-$$

In the absence of annotations, these rules actually correspond to *snips* in SAX: *analytic cuts* [58] where $\mathbf{A}$ is a subformula of an axiom's principal formula (hence the rule names). Likewise, the identity rule $y \leftarrow x$, which copies the contents of x to y, comes in two forms depending on where the principal "formula" $\mathbf{A}$ is outputted:

$$\Gamma, x \Leftarrow \mathbf{A} \vdash y \leftarrow x \div (y \Leftarrow \mathbf{A})\ \textsc{id}^+ \qquad \Gamma, x \Rightarrow \mathbf{A} \vdash y \leftarrow x \div (y \Rightarrow \mathbf{A})\ \textsc{id}^-$$

2.6 Labelled Sums and Lazy Records

Let us pick up where we left off with labelled sums and lazy records in Section 2.1. Recalling our verificationist point of view and with our new judgmental structure explicitly indicating the flow of type information, the right/introduction rule for $\oplus\{\ell : A_\ell\}_{\ell\in S}$ corresponds to the following right axiom.

$$\frac{\oplus\mathrm{R/I},\ k \in S \qquad \Gamma \vdash x \Leftarrow A_k}{\Gamma \vdash y \Leftarrow \oplus\{\ell : A_\ell\}_{\ell\in S}} \quad \rightsquigarrow \quad \boxed{\begin{array}{l}\oplus\mathrm{R},\ k \in S \\ \Gamma, x \Leftarrow A_k \mid \phi(k \cdot x) \vdash y.k \cdot x \div (y \Leftarrow \oplus\{\ell : A_\ell\}_{\ell\in S} \mid \phi(y))\end{array}}$$

The process $y.k \cdot x$ writes the tagged value $k \cdot x$ to the future addressed by y. As a result, this rule can be viewed as an instance of the assignment rule in Hoare logic. Indeed, the postcondition flows from right to left, becoming a precondition by a suitable substitution. Thus, the type itself need not embed any type refinements. Now, the corresponding left rule is inherited from the sequent calculus:

$$\frac{\oplus\mathrm{L} \qquad \{\Gamma, x \Rightarrow A_k, y \Rightarrow \oplus\{\ell : A_\ell\}_{\ell\in S} \vdash (z \Leftarrow C)\}_{k\in S}}{\Gamma, y \Rightarrow \oplus\{\ell : A_\ell\}_{\ell\in S} \vdash (z \Leftarrow C)} \quad \rightsquigarrow$$

$$\boxed{\frac{\oplus\mathrm{L} \qquad \{\Gamma, x \Rightarrow A_k \mid \phi(k \cdot x), y \Rightarrow \oplus\{\ell : A_\ell\}_{\ell\in S} \mid \phi(y) \wedge y \equiv k \cdot x \vdash P_k(x) \div (z \Leftarrow \mathbf{C})\}_{k\in S}}{\Gamma, y \Rightarrow \oplus\{\ell : A_\ell\}_{\ell\in S} \mid \phi(y) \vdash \mathbf{case}\, y\, \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell\in S} \div (z \Leftarrow \mathbf{C})}}$$

The process $\mathbf{case}\, y\, \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell\in S}$ proceeds by cases of the tagged address stored in y. Adding refinements requires some care: the antecedent x inherits $\phi(k \cdot x)$ to be in *harmony* [61] with the right rule above. Thus, y is additionally subject to $y \equiv k \cdot x$ to indicate its relationship to x when typing each case branch $P_k(x)$. This strong form of path sensitivity is necessary to complete the following example.

Example 2.3 (Negation) *Letting* $\mathrm{bool} = \oplus\{\mathrm{true} : \mathbf{1}, \mathrm{false} : \mathbf{1}\}$*, we can define Boolean negation of* x*, storing the result in* y*, as* $P \triangleq \mathbf{case}\, x\, \{\mathrm{true} \cdot x' \Rightarrow y.\mathrm{false} \cdot x', \mathrm{false} \cdot x' \Rightarrow y.\mathrm{true} \cdot x'\}$*. Then, the judgment* $x \Rightarrow \mathrm{bool} \vdash P \div (y \Leftarrow \mathrm{bool} \mid \phi(x, y))$ *is derivable where* $\phi(x, y) \triangleq (\mathrm{is}_{\mathrm{true}}(x) \supset \mathrm{is}_{\mathrm{false}}(y)) \wedge (\mathrm{is}_{\mathrm{false}}(x) \supset \mathrm{is}_{\mathrm{true}}(y))$*. In the first branch, for example, the critical point is when* $x' \Rightarrow \mathrm{bool}$ *meets* $x' \Leftarrow \mathrm{bool} \mid \phi(x, \mathrm{false} \cdot x')$ *via subsumption—the assumption that* $x \Rightarrow \mathrm{bool} \mid x \equiv \mathrm{true} \cdot x'$ *is essential.*

To develop the lazy record type $\&\{\ell : \mathbf{A}_\ell\}_{\ell\in S}$, we again refine its ordinary right/introduction rule:

$$\frac{\&\mathrm{R/I} \qquad \{\Gamma \vdash x \Leftarrow A_\ell\}_{\ell\in S}}{\Gamma \vdash y \Leftarrow \&\{\ell : A_\ell\}_{\ell\in S}} \quad \rightsquigarrow \quad \boxed{\frac{\&\mathrm{R} \qquad \{\Gamma \vdash P_\ell(x) \div (x \Leftarrow A_\ell \mid \phi_\ell(x))\}_{\ell\in S} \qquad \Gamma, y \Rightarrow \ldots \mid \bigwedge_{\ell\in S} \phi_\ell(y.\ell) \vdash \psi(y)}{\Gamma \vdash \mathbf{case}\, y\, \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell\in S} \div (y \Leftarrow \&\{\ell : A_\ell \mid \phi_\ell(\cdot)\}_{\ell\in S} \mid \psi(y))}}$$

In this case, the process $\mathbf{case}\, y\, \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell\in S}$ writes a *destination-passing* lazy record to y, where its ℓ^{th} projection writes to the x provided. Destination-passing style is key to our asynchronous operational semantics, as a client of y should be able to refer to x even if it has not yet been populated by $P_\ell(x)$. Now, since the succedents of the premises are inputs in the original rule, they must each be handed a new postcondition; hence each A_ℓ becomes $A_\ell \mid \phi_\ell(\cdot)$. The twist is that $\psi(y)$ is verified directly by assuming that there is some y subject to $\phi_\ell(y.\ell)$ for each ℓ (the ellipsis repeats the record type). As we mentioned in Section 2.3, this respects encapsulation of the record by not reifying its contents into the assertion logic. In particular, intensional properties about the record *cannot* be verified if ϕ_ℓ hides them, i.e., does not mention them. Let us work through a small example to demonstrate.

Example 2.4 (Record Encapsulation) *Let* $P \triangleq \mathbf{case}\, y\, \{\mathrm{fst} \cdot x \Rightarrow z \leftarrow z.\langle\rangle; x.\mathrm{true} \cdot z\}$*. Then the judgment* $\cdot \vdash (y \Leftarrow \&\{\mathrm{fst} : \mathrm{bool} \mid \mathrm{is}_{\mathrm{true}}\} \mid \mathrm{is}_{\mathrm{true}}(y.\mathrm{fst}))$ *is derivable, but* $\cdot \vdash (y \Leftarrow \&\{\mathrm{fst} : \mathrm{bool}\} \mid \mathrm{is}_{\mathrm{true}}(y.\mathrm{fst}))$ *is not.*

Now, we produce the corresponding refined left axiom below, following our preliminary development in Section 2.1. We type the process $y.k \cdot x$, which allows the k^{th} projection of y to populate x.

$$\frac{\&\text{E},\ k \in S}{\Gamma \vdash y \Rightarrow \&\{\ell : A_\ell\}_{\ell \in S}} \quad \rightsquigarrow \quad \boxed{\frac{\&\text{L},\ k \in S}{\Gamma, y \Rightarrow \&\{\ell : \mathbf{A}_\ell\}_{\ell \in S} \mid \phi(y) \vdash y.k \cdot x \div (x \Rightarrow \mathbf{A}_k)}}$$

While it is tempting to let x also be subject to $x \equiv y.k$ as an analogously strong form of path sensitivity, it would not be type-sound, because that relationship is not made explicit in the typing of the right rule (only in the verification of $\psi(y)$). As a result, we produce the following non-example.

Example 2.5 (Failure of Swap) *We define the following process P that swaps the components of a record p and stores the result in q: $P \triangleq \mathbf{case}\, q\, \{\text{fst} \cdot x \Rightarrow p.\text{snd} \cdot x, \text{snd} \cdot y \Rightarrow p.\text{fst} \cdot y\}$. Then, the following judgment is* not *derivable:*

$$p \Rightarrow \&\{\text{fst} : A, \text{snd} : B\} \vdash P \div (q \Leftarrow \{\text{fst} : B \mid \lambda x.\, x \equiv p.\text{fst}, \text{snd} : A \mid \lambda y.\, y \equiv p.\text{snd}\})$$

Given that we are already working in the presence of non-termination, a larger set of effects occurring at negative type [41] may invalidate this kind of equality anyways.

2.7 Dependent Types

We now turn our attention to dependent eager pair $(x : \mathbf{A}) \otimes B(x)$ and function $(x : \mathbf{A}) \to \mathbf{B}(x)$ types. Following our development of the labelled sum type, we convert the right/introduction rule to a right axiom. We once again observe that flowing type information bottom up corresponds to a right-to-left flow.

$$\frac{\otimes\text{R/I}}{\Gamma \vdash x \Leftarrow A \qquad \Gamma \vdash y \Leftarrow B(x)}{\Gamma \vdash z \Leftarrow (x : A) \otimes B(x)} \quad \rightsquigarrow \quad \boxed{\frac{\otimes\text{R}}{\Gamma, x \Leftarrow \mathbf{A}, y \Leftarrow B(x) \mid \psi(\langle x, y\rangle) \vdash z.\langle x, y\rangle \div (z \Leftarrow (x : \mathbf{A}) \otimes B(x) \mid \psi(z))}}$$

The process $z.\langle x, y\rangle$ writes the pair of x and y to z. Sensing that this rule too resembles an instance of the assignment rule in Hoare logic, we can now explain the asymmetry between both conjuncts: while ϕ flows to the pair's first component x, its second component y inherits ψ by substitution of $\langle x, y\rangle$ for z. Like sums, the left rule $\otimes$L follows from harmony with its right axiom. Refer to Figure 4 for this rule as well as those for the unit type. Path sensitivity enables the following example, which was unavailable for lazy records. Note that we use the usual shorthand $A \otimes B$ for non-dependent pairs.

Example 2.6 (Swap) *Let $P \triangleq \mathbf{case}\, z\, \{\langle x, y\rangle \Rightarrow w.\langle y, x\rangle\}$ be a process that swaps a (non-dependent) pair addressed by z and writes it to w. Then, the following judgment is derivable:*

$$z \Rightarrow A \otimes B \vdash P \div (w \Leftarrow B \otimes A \mid \forall x, y.\, z \equiv \langle x, y\rangle \supset w \equiv \langle y, x\rangle)$$

Following our approach for lazy records, the refined right rule for the dependent function type is as follows.

$$\boxed{\frac{\to\text{R}}{\Gamma, x \Rightarrow A \mid \phi(x) \vdash P(x, y) \div (y \Leftarrow B(x) \mid \psi(x, y)) \qquad \Gamma, z \Rightarrow \ldots \mid \forall x.\, \phi(x) \supset \psi(x, z\text{`}x) \vdash \chi(z)}{\Gamma \vdash \mathbf{case}\, z\, (\langle x, y\rangle \Rightarrow P(x, y)) \div (z \Leftarrow (x : A \mid \phi(x)) \to (B(x) \mid \psi(x, \cdot)) \mid \chi(z))}}$$

Like that for lazy records, the process $\mathbf{case}\, z\, (\langle x, y\rangle \Rightarrow P(x, y))$ writes a destination-passing function to z whose body is $P(x, y)$ where x refers to the argument source and y the result destination. Since x and y take assertions as inputs in the premise, function types include a precondition ϕ on x and a postcondition ψ on y. As with lazy records, the postcondition χ on z is verified directly from ϕ and ψ, leaving the function body encapsulated (again, the ellipsis repeats the function type). Let us look at an example to interrogate abstraction boundaries.

Example 2.7 (Left Unit of Addition) *Let* $\text{nat} = \oplus\{\text{zero} : \mathbf{1}, \text{succ} : \text{nat}\}$. *In Example 2.9, we define* $\text{add}(x : \text{nat}, y : \text{nat}, z : \text{nat} \mid x + y \equiv z)$ *by induction on* x*, assuming that the uninterpreted function* $(+)$ *is subject to the appropriate axioms. Now, we package this definition into a process* P *writing to* w *with a pair of arguments* p *and result* z*:*

$$P \triangleq \mathbf{case}\, w\, (\langle p, z\rangle \Rightarrow \mathbf{case}\, p\, (\langle x, y\rangle \Rightarrow \text{add}(x, y, z)))$$

Then, the following judgment is derivable, which asserts that the left unit of addition is zero by applications to w*. Note that proving zero as the right unit of addition would require a separate induction on* x*.*

$$\cdot \vdash P \div (w \Leftarrow (p : \text{nat} \otimes \text{nat}) \to (\text{nat} \mid \lambda z.\, \forall x, y.\, p \equiv \langle x, y\rangle \supset z \equiv x + y) \mid \forall x, y.\, \text{is}_{\text{zero}}(x) \supset w\,\text{`}\langle x, y\rangle \equiv y)$$

However, the following judgment is not *derivable, because the action of addition is hidden by the function's output refined type.*

$$\cdot \vdash P \div (w \Leftarrow (p : \text{nat} \otimes \text{nat}) \to (\text{nat} \mid \lambda z.\, \forall x, y.\, p \equiv \langle x, y\rangle \supset \text{is}_{\text{zero}}(x) \supset y \equiv z) \mid \forall x, y.\, w\,\text{`}\langle x, y\rangle \equiv x + y)$$

Note that these functions are uncurried to avoid having to refine each intermediate function type with the necessary information to prove the final postcondition.

Finally, it remains to convert the dependent elimination rule to a left axiom. The former outputs both A and B from top down, forcing the second premise to take A as an input. Likewise in the latter, $\mathbf{A}$ flows to the same side (left) of the sequent, but $\mathbf{B}$ flows to the right.

$$\frac{\Gamma \vdash z \Rightarrow (x : A) \to B(x) \qquad \Gamma \vdash x \Leftarrow A}{\Gamma \vdash y \Rightarrow B(x)}\ {}^{\to\text{E}} \rightsquigarrow \boxed{\frac{}{\Gamma, z \Rightarrow (x : \mathbf{A}) \to \mathbf{B}(x) \mid \phi(z), x \Leftarrow \mathbf{A} \vdash z.\langle x, y\rangle \div (y \Rightarrow \mathbf{B}(x))}\ {}^{\to\text{L}}}$$

In this case, the process $z.\langle x, y\rangle$ passes the argument x and result destination z to the function addressed by z. Like lazy records, this left rule also omits strong path sensitivity, recalling our discussion of type soundness and a further integration of effects.

2.8 Recursion: Assume-Guarantee Reasoning and Recursion Invariants

Following Lakhani et al. [36], a definition call outputs its ascribed type signature when the definition body checks against the same signature. To type recursive definitions, we take a mixed inductive-coinductive view of the typing judgment as we did with subtyping with the rule below.

$$\boxed{\frac{f(\overline{x : \mathbf{A}}, y : \mathbf{C}(\overline{x})) = P(\overline{x}, y) \qquad \infty(\overline{x \Rightarrow \mathbf{A}} \vdash P(\overline{x}, y) \div (y \Leftarrow \mathbf{C}(\overline{x})))}{\overline{x \Leftarrow \mathbf{A}} \vdash f(\overline{x}, y) \div (y \Rightarrow \mathbf{C}(\overline{x}))}\ \textsc{call}}$$

Thus, when a recursive definition is checked against its type signature, recursive calls coinductively produce the typing derivation computed so far, corresponding to *assume-guarantee reasoning* that is standard for both program logics and typing recursion [9,20,31]. In particular, it seems to be the syntactic reflection of coinductively-defined partial correctness (on which we elaborate in the next section); this connection has been explored by Bell and Chlipala [6]. We reproduce Example 22 in [59] below to show the exact mechanics of this process.

Example 2.8 (Typing Derivation Circularity) *Recalling* $\text{nat} = \oplus\{\text{zero} : \mathbf{1}, \text{succ} : \text{nat}\}$*, the following process definition performs a trivial induction on a natural number and returns unit:*

$$\text{eat}(x : \text{nat}, y : \mathbf{1}) = \mathbf{case}\, x\, \{\text{zero} \cdot x' \Rightarrow y \leftarrow x', \text{succ} \cdot x' \Rightarrow \text{eat}(x', y)\}$$

The typing derivation for its body is as follows (process terms are omitted for space) where † denotes a circular edge and W stands for antecedent weakening.

$$\oplus\mathrm{L}\ \dfrac{\leq\mathrm{L}\ \dfrac{\mathrm{ID}^{+}\ \dfrac{}{\ldots, x' \Leftarrow \mathbf{1} \vdash y \Leftarrow \mathbf{1}}}{x' \Rightarrow \mathbf{1}, x \Rightarrow \mathrm{nat} \mid x \equiv \mathrm{zero} \cdot x' \vdash y \Leftarrow \mathbf{1}} \qquad \leq\mathrm{R},\leq\mathrm{L},\mathrm{W}\ \dfrac{\mathrm{CALL}\ \dfrac{\infty(x' \Rightarrow \mathrm{nat} \vdash y \Leftarrow \mathbf{1}) \quad \dagger}{x' \Leftarrow \mathrm{nat} \vdash y \Rightarrow \mathbf{1}}}{x' \Rightarrow \mathrm{nat}, x \Rightarrow \mathrm{nat} \mid x \equiv \mathrm{succ} \cdot x' \vdash y \Leftarrow \mathbf{1}}}{x \Rightarrow \mathrm{nat} \vdash y \Leftarrow \mathbf{1} \quad \dagger}$$

Notice above that *because* definition calls output, there is a mandatory change of phase between checking the body and the call, which may strengthen the postcondition and weaken the preconditions. This is analogous to checking whether the *loop invariant* implies the postcondition in the loop rule in Hoare logic. As demonstrated in the examples below, our formulation treats induction, coinduction, and mixed induction and coinduction uniformly.

Example 2.9 (Addition) *Recall from Example 2.7 that we will define addition where* $\mathrm{nat} = \oplus\{\mathrm{zero} : \mathbf{1}, \mathrm{succ} : \mathrm{nat}\}$ *and* $(+)$ *is subject to the axioms* $\forall x, y.\, \mathrm{zero} \cdot x + y \equiv y$ *and* $\forall x, y.\, \mathrm{succ} \cdot x + y \equiv \mathrm{succ} \cdot (x + y)$.

$$\begin{aligned}\mathrm{add}(x : \mathrm{nat}, y : \mathrm{nat}, z : \mathrm{nat} \mid z \equiv x + y) = \\ \mathbf{case}\, x\, \{\mathrm{zero} \cdot x' \Rightarrow z \leftarrow y, \mathrm{succ} \cdot x' \Rightarrow z' \leftarrow \mathrm{add}(x', y, z');\, z.\mathrm{succ} \cdot z'\}\end{aligned}$$

Of significance is checking the snip in the succ *branch:* $\mathrm{add}(x', y, z')$ *flows* $z' \Rightarrow \mathrm{nat} \mid z' = x' + y$ *(the induction hypothesis from typing circularity) to the right but* $z.\mathrm{succ} \cdot z'$ *flows* $z' \Leftarrow \mathrm{nat} \mid \mathrm{succ} \cdot z' \equiv x + y$ *to the left (the induction step). Path sensitivity gives us* $x \Rightarrow \mathrm{nat} \mid x \equiv \mathrm{succ} \cdot x'$*, resolving the tension by subsumption.*

Example 2.10 (Nonzero Lazy Streams) *In type refinement systems, (co)inductive invariants are typically folded into refinements [44] in lieu of being defined as separate (co)predicates [45,38]. For example, if* $\mathrm{str} = \&\{\mathrm{head} : \mathrm{nat}, \mathrm{tail} : \mathrm{str}\}$ *classifies natural number streams, then* $\mathrm{sstr} = \&\{\mathrm{head} : \mathrm{nat} \mid \mathrm{is}_{\mathrm{succ}}, \mathrm{tail} : \mathrm{sstr}\}$ *classifies those that are pointwise nonzero. We can check the following definition, which shows that a certain increasing stream starting from a nonzero number is pointwise nonzero.*

$$\mathrm{up}(x : \mathrm{nat} \mid \mathrm{is}_{\mathrm{succ}}, y : \mathrm{sstr}) = \mathbf{case}\, y\, \{\mathrm{head} \cdot h \Rightarrow h \leftarrow x, \mathrm{tail} \cdot t \Rightarrow x' \leftarrow x'.\mathrm{succ} \cdot x; \mathrm{up}(x', t)\}$$

Considering the body of up *as the coinduction step, the coinduction hypothesis (i.e., that the tail is pointwise nonzero) is implicitly part of* sstr *as outputted by the recursive call to populate* t*.*

Example 2.11 (Left-Fair Streams) *We can extend the technique from the previous example to operate on mixed inductive-coinductive data structures. For example, consider the type below of* left-fair streams *[3] where, assuming termination, consecutive elements of type* $\mathbf{A}$ *are interspersed with finitely many timeout (*later*) labels.*

$$\mathrm{lfair} = \oplus\{\mathrm{now} : \&\{\mathrm{head} : \mathbf{A}, \mathrm{tail} : \mathrm{lfair}\}, \mathrm{later} : \mathrm{lfair}\}$$

We will define a projection *operation that is guaranteed to clear these labels, producing the underlying stream. For the sake of this example, we define (later-less) streams as the following recursively refined record:*

$$\mathrm{str} = \&\{\mathrm{fst} : \oplus\{\mathrm{now} : \&\{\mathrm{head} : \mathbf{A}, \mathrm{tail} : \mathrm{str}\}, \mathrm{later} : \mathrm{str}\} \mid \mathrm{is}_{\mathrm{now}}\}$$

In the definition below, the desired invariant is implicitly checked by coinduction to construct the stream prioritized over induction to vacate the later *labels.*

$$\begin{aligned}\mathrm{proj}(x : \mathrm{lfair}, y : \mathrm{str}) = \mathbf{case}\, y\, \{\mathrm{fst} \cdot l \Rightarrow \\ \mathbf{case}\, x\, \{\mathrm{now} \cdot s \Rightarrow s' \leftarrow \mathbf{case}\, s'\, \{ \quad & \mathrm{head} \cdot h \Rightarrow s.\mathrm{head} \cdot h, \\ & \mathrm{tail} \cdot y' \Rightarrow x' \leftarrow s.\mathrm{tail} \cdot x'; \mathrm{proj}(x', y')\}; l.\mathrm{now} \cdot s', \\ \mathrm{later} \cdot x' \Rightarrow \mathrm{proj}(x', y)\}\}\end{aligned}$$

Since this process definition is complex, we turn the reader's attention specifically to $l.\text{now} \cdot s'$ *which outputs* $s' \Leftarrow \&\{\ldots\} \mid \text{is}_{\text{now}}(\text{now} \cdot s')$*. Checking* s' *using* $\&R$ *on the left-hand side of the cut then directly verifies* $\text{is}_{\text{now}}(\text{now} \cdot s')$*, as desired. Finally, the second recursive call trivially preserves the invariant.*

We finish this subsection by commenting on the seemingly dangerous interaction between non-termination and type soundness.

Remark 2.12 (Non-Termination) *In a cut, non-termination on the left allows the assumption of* $\bot$ *on the right. Thus, unrestricted lazy evaluation would be incompatible with type soundness, because an unused non-terminating computation can be discarded, exposing a potentially unsafe computation checked against* $\bot$ *[70]. This is not an issue in DRSAX, as the futures-based (as opposed to* speculations*-based [27, Chapter 38]) operational semantics defined in the next section does not discard computations.*

2.9 Summary

The process typing rules reviewed in the previous subsections are collected into Figure 4.

3 Operational Semantics, Type Soundness, and Observable Partial Correctness

In this section, we define typing and reduction for *configurations* of DRSAX processes and the future cells with which they communicate. Then, we prove syntactic type soundness. As we alluded in the introduction, this entails *observable* partial correctness, in which hereditarily non-encapsulated sub-configurations (of *purely* positive type) satisfy their associated postconditions directly.

3.1 Configuration Reduction and Typing

Definition 3.1 (Configuration) *Configurations are multisets of process and cell objects defined by the following grammar.*

$\mathcal{C} := \cdot$	*empty configuration*
$\mid \text{proc}\, a\, P$	*process* P *writing to cell addressed by* a
$\mid \text{!cell}\, a\, S$	*persistent cell addressed by* a *with contents* $S := V \mid K$
$\mid \mathcal{C}, \mathcal{C}$	*join of two configurations*

That is, the join and empty rules form a commutative monoid. A configuration $\mathcal{F}$ *is* final *when it only consists of cells.*

Configuration reduction ($\mapsto$) is defined by *multiset rewriting rules* [10] in Figure 5, which replace any subset of a configuration matching the left-hand side with the right-hand side. ! indicates objects that persist across reductions. Now, because bidirectional typing would complicate configuration typing, we first define the corresponding *non-bidirectional* process typing below.

Definition 3.2 (Non-bidirectional Typing) *Let* $x : \mathbf{A}, \ldots, y : \mathbf{B}(x, \ldots) \vdash P \div (z : \mathbf{A}(x, \ldots, y))$ *be generated by rules identical to those in Figure 4, but with* $\Rightarrow$ *and* $\Leftarrow$ *replaced by* (:) *and* AnnoL/R *removed.*

As usual, we must verify that the bidirectional process typing is sound and complete with respect to the above.

Lemma 3.3 (Soundness and Completeness of Bidirectional Typing) *Let* $|P|$ *erase type annotations in* P *and* $|J|$ *turn* $\Rightarrow$ *and* $\Leftarrow$ *to* (:)*. Extending* $|\cdot|$ *to* Γ *in the obvious way,* $\Gamma \vdash P \div J$ *iff* $|\Gamma| \vdash |P| \div |J|$*.*

Proof. Both are a routine mixed induction and coinduction on the typing derivation; going forwards erases AnnoL/R and going backwards essentially inserts AnnoL/R as dictated by P. □

From now, Γ and Δ refer to contexts associating runtime addresses to refined types. Thus, as a slight abuse of notation, we allow runtime addresses to stand in place of address variables in process typing. Finally, the *configuration typing* judgment $\Gamma \vdash \mathcal{C} \div \Delta$ is inductively generated by the rules in Figure 6,

$$\frac{\Gamma \vdash P \div (y \Rightarrow \mathbf{A}) \qquad \Gamma \vdash \mathbf{A} \leq \mathbf{B}}{\Gamma \vdash P \div (y \Leftarrow \mathbf{B})}\ {\leq}\mathrm{R} \qquad \frac{\Gamma, x \Leftarrow \mathbf{B} \vdash P \div (z \Leftarrow \mathbf{C}) \qquad \Gamma \vdash \mathbf{A} \leq \mathbf{B}}{\Gamma, x \Rightarrow \mathbf{A} \vdash P \div (z \Leftarrow \mathbf{C})}\ {\leq}\mathrm{L}$$

$$\frac{\Gamma \vdash P \div (y \Leftarrow \mathbf{A})}{\Gamma \vdash (y : \mathbf{A})\ \mathbf{in}\ P \div (y \Rightarrow \mathbf{A})}\ \textsc{anno}\mathrm{R} \qquad \frac{\Gamma, x \Rightarrow \mathbf{A} \vdash P \div (z \Leftarrow \mathbf{C})}{\Gamma, x \Leftarrow \mathbf{A} \vdash (x : \mathbf{A})\ \mathbf{in}\ P \div (z \Leftarrow \mathbf{C})}\ \textsc{anno}\mathrm{L}$$

$$\frac{\Gamma \vdash P(x) \div (x \Leftarrow \mathbf{A}) \qquad \Gamma, x \Leftarrow \mathbf{A} \vdash Q(x) \div (z \Leftarrow \mathbf{C})}{\Gamma \vdash x \leftarrow P(x); Q(x) \div (z \Leftarrow \mathbf{C})}\ \textsc{snip}^{+}$$

$$\frac{\Gamma \vdash P(x) \div (x \Rightarrow \mathbf{A}) \qquad \Gamma, x \Rightarrow \mathbf{A} \vdash Q(x) \div (z \Leftarrow \mathbf{C})}{\Gamma \vdash x \leftarrow P(x); Q(x) \div (z \Leftarrow \mathbf{C})}\ \textsc{snip}^{-}$$

$$\Gamma, x \Leftarrow \mathbf{A} \vdash y \leftarrow x \div (y \Leftarrow \mathbf{A})\ \textsc{id}^{+} \qquad \Gamma, x \Rightarrow \mathbf{A} \vdash y \leftarrow x \div (y \Rightarrow \mathbf{A})\ \textsc{id}^{-}$$

$$\frac{\Gamma \vdash \phi(\langle\rangle)}{\Gamma \vdash x.\langle\rangle \div (x \Leftarrow \mathbf{1} \mid \phi(x))}\ \mathbf{1}\mathrm{R} \qquad \frac{\Gamma, x \Rightarrow \mathbf{1} \mid \phi(x) \wedge x \equiv \langle\rangle \vdash P \div (z \Leftarrow H)}{\Gamma, x \Rightarrow \mathbf{1} \mid \phi(x) \vdash \mathbf{case}\, x\, (\langle\rangle \Rightarrow P) \div (z \Leftarrow H)}\ \mathbf{1}\mathrm{L}$$

$$\Gamma, x \Leftarrow \mathbf{A}, y \Leftarrow B(x) \mid \psi(\langle x, y\rangle) \vdash z.\langle x, y\rangle \div (z \Leftarrow (x : \mathbf{A}) \otimes B(x) \mid \psi(z))\ \otimes\mathrm{R}$$

$$\frac{\Gamma, x \Rightarrow \mathbf{A}, y \Rightarrow B(x) \mid \psi(\langle x, y\rangle), z \Rightarrow (x : \mathbf{A}) \otimes B(x) \mid z \equiv \langle x, y\rangle \vdash P(x, y) \div (w \Leftarrow \mathbf{C})}{\Gamma, z \Rightarrow (x : \mathbf{A}) \otimes B(x) \mid \psi(z) \vdash \mathbf{case}\, z\, (\langle x, y\rangle \Rightarrow P(x, y)) \div (w \Leftarrow \mathbf{C})}\ \otimes\mathrm{L}$$

$$\frac{\Gamma, x \Rightarrow A \mid \phi(x) \vdash P(x, y) \div (y \Leftarrow B(x) \mid \psi(x, y)) \qquad \Gamma, z \Rightarrow \ldots \mid \forall x.\, \phi(x) \supset \psi(x, z\text{`}x) \vdash \chi(z)}{\Gamma \vdash \mathbf{case}\, z\, (\langle x, y\rangle \Rightarrow P(x, y)) \div (z \Leftarrow (x : A \mid \phi(x)) \to (B(x) \mid \psi(x, \cdot)) \mid \chi(z))}\ {\to}\mathrm{R}$$

$$\Gamma, z \Rightarrow (x : \mathbf{A}) \to \mathbf{B}(x) \mid \phi(z), x \Leftarrow \mathbf{A} \vdash z.\langle x, y\rangle \div (y \Rightarrow \mathbf{B}(x))\ {\to}\mathrm{L}$$

$$\Gamma, x \Leftarrow A_k \mid \phi(k \cdot x) \vdash y.k \cdot x \div (y \Leftarrow \oplus\{\ell : A_\ell\}_{\ell \in S} \mid \phi(y))\ \oplus\mathrm{R},\ k \in S$$

$$\frac{\{\Gamma, x \Rightarrow A_k \mid \phi(k \cdot x), y \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in S} \mid \phi(y) \wedge y \equiv k \cdot x \vdash P_k(x) \div (z \Leftarrow \mathbf{C})\}_{k \in S}}{\Gamma, y \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in S} \mid \phi(y) \vdash \mathbf{case}\, y\, \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell \in S} \div (z \Leftarrow \mathbf{C})}\ \oplus\mathrm{L}$$

$$\frac{\{\Gamma \vdash P_\ell(x) \div (x \Leftarrow A_\ell \mid \phi_\ell(x))\}_{\ell \in S} \qquad \Gamma, y \Rightarrow \ldots \mid \bigwedge_{\ell \in S} \phi_\ell(y.\ell) \vdash \psi(y)}{\Gamma \vdash \mathbf{case}\, y\, \{\ell \cdot x \Rightarrow P_\ell(x)\}_{\ell \in S} \div (y \Leftarrow \&\{\ell : A_\ell \mid \phi_\ell(\cdot)\}_{\ell \in S} \mid \psi(y))}\ \&\mathrm{R}$$

$$\Gamma, y \Rightarrow \&\{\ell : \mathbf{A}_\ell\}_{\ell \in S} \mid \phi(y) \vdash y.k \cdot x \div (x \Rightarrow \mathbf{A}_k)\ \&\mathrm{L},\ k \in S$$

$$\frac{f(\overline{x : \mathbf{A}}, y : \mathbf{C}(\overline{x})) = P(\overline{x}, y) \qquad \infty(\overline{x \Rightarrow \mathbf{A}} \vdash P(\overline{x}, y) \div (y \Leftarrow \mathbf{C}(\overline{x})))}{\overline{x \Leftarrow \mathbf{A}} \vdash f(\overline{x}, y) \div (y \Rightarrow \mathbf{C}(\overline{x}))}\ \textsc{call}$$

Fig. 4. Process Typing

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

Fig. 5. Configuration Reduction

$$
\textsc{Proc}\ \frac{\Delta \leq \Gamma \qquad \Gamma \vdash P \div (a : \mathbf{A}) \qquad \Gamma \vdash \mathbf{A} \leq \mathbf{B}}{\Delta \vdash \mathsf{proc}\, a\, P \div (\Delta, a : \mathbf{B})}
\qquad
\textsc{CellV}\ \frac{\Gamma \vdash \mathsf{proc}\, a\, (a.V) \div \Delta}{\Gamma \vdash\ !\mathsf{cell}\, a\, V \div \Delta}
\qquad
\textsc{CellK}\ \frac{\Gamma \vdash \mathsf{proc}\, a\, (\mathbf{case}\, a\, K) \div \Delta}{\Gamma \vdash\ !\mathsf{cell}\, a\, K \div \Delta}
$$

$$
\textsc{Empty}\ \Gamma \vdash \cdot \div \Gamma
\qquad
\textsc{Join}\ \frac{\Gamma \vdash \mathcal{C} \div \Gamma' \qquad \Gamma' \vdash \mathcal{C}' \div \Delta}{\Gamma \vdash \mathcal{C}, \mathcal{C}' \div \Delta}
$$

Fig. 6. Configuration Typing

which types the objects in $\mathcal{C}$ where sources are in Γ and destinations in Δ. The rules are designed to admit the following conveniences.

Remark 3.4 (Proof Principles) *The theorems in the next subsection use the following proof principles.*

- Right-to-left induction*: a configuration typing derivation D can be viewed as a list of process typing derivations where readers of an address appear to the right of its writer. Thus, induction on D isolates the rightmost derivation and applies the induction hypothesis to the sub-configuration on the left.*
- Inversion modulo subtyping*: following [16], the* Proc *rule contains subtyping "slack" premises on both sides of the sequent. Thus, inversion on its subderivation may use a quick induction to absorb consecutive instances of $\leq R/L$ into the slack using Lemma 2.2. As a result, it suffices to only consider the non-subtyping process typing rules. Note that the premise $\Delta \leq \Gamma$ is defined by viewing Δ and Γ as iterated dependent pair types.*

3.2 Syntactic Type Soundness and Observable Partial Correctness

Now that we have reviewed configuration reduction and typing, we prove syntactic type soundness by a standard appeal to progress and preservation. Then, we define and prove observable partial correctness.

Theorem 3.5 (Progress) *If $\cdot \vdash \mathcal{C} \div \Delta$ then either $\mathcal{C}$ is final or $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$.*

Proof. By right-to-left induction on the configuration typing derivation.

(i) If $\mathcal{C} = \mathcal{C}_1, !\mathsf{cell}\, a\, S$, then by the induction hypothesis, either $\mathcal{C}_1$ is final, in which case $\mathcal{C}$ is final, or $\mathcal{C}_1 \mapsto \mathcal{C}_1'$, in which case $\mathcal{C} \mapsto \mathcal{C}_1', !\mathsf{cell}\, a\, S$.

(ii) If $\mathcal{C} = \mathcal{C}_1, \mathsf{proc}\, c\, P$, then by the induction hypothesis, either $\mathcal{C}_1 \mapsto \mathcal{C}_1'$, in which case $\mathcal{C} \mapsto \mathcal{C}_1', \mathsf{proc}\, a\, P$. Otherwise, $\mathcal{C}_1$ is final. If P is a cut, definition call, or writes, then $\mathcal{C}$ steps by P alone. Otherwise, inversion modulo subtyping on the appropriate subderivation in that for $\mathcal{C}_1$ reveals a cell of the right shape that P reads from, letting $\mathcal{C}$ step.

□

Theorem 3.6 (Preservation) *If $\Gamma \vdash \mathcal{C} \div \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$, then $\Gamma \vdash \mathcal{C} \div \Delta'$ for some $\Delta' \supseteq \Delta$.*

Proof. We proceed by induction on the reduction step and then by inversion modulo subtyping on the typing derivation D. The cases where a single process steps—cuts, definition calls, and writes—are straightforward. The identity rule and projection/application of a continuation are also straightforward by copying the derivation of the object read to that of the destination. However, pattern matching on a value is nontrivial due to path sensitivity. For example, when $\oplus$R meets $\oplus$L at address b subject to $\phi(b)$, the k^{th} premise of $\oplus$L requires the type of b to be strengthened with the equality $b \equiv k \cdot a$ where a is somewhere to the left in D. In updating the derivation for b locally, a would be flowed $\phi(k \cdot a) \wedge k \cdot a \equiv k \cdot a$, which is subsumed by $\phi(k \cdot a)$ via $\leq$L. Thus, the readers of a see the same type ascription as before. To ensure that all readers of b except for the scrutinized instance of $\oplus$L see the same type ascription as before, we inductively update their left "slack" premises noting that $\phi(b) \wedge b \equiv k \cdot a$ implies $\phi(b)$. □

For an alternate proof strategy of type preservation that grapples with this strong form of path sensitivity in a functional setting, see [52, Lemme 13.8.7 and Théorème 13.8.8]. Now, to prove observable partial correctness, we follow DeYoung et. al. [22] and refer to addresses occurring in values as *observable* with all else being *hidden*. As a result, final configurations of purely positive type are completely observable.

Lemma 3.7 (Final Configurations of Purely Positive Type [22, Corollary 12]) Purely *positive refined types* $\mathbf{A}^{++}$ *are those that only contain positive type constructors. Extending this definition to* Γ *in the obvious way, if* $\cdot \vdash \mathcal{F} \div \Gamma^{++}$*, then* $\mathcal{F}$ *only contains objects of the form* $!\text{cell}\, a\, V$ *(whose addresses are observable).*

Proof. By inversion on the configuration typing derivation. □

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

Theorem 3.8 (Observable Satisfaction) $\mathcal{F} \vDash \Gamma^{++}$ *is inductively generated by the rules below.*

$$\frac{}{\cdot \vDash \cdot} \qquad \frac{\mathcal{F} \vDash \Gamma^{++} \qquad \Gamma \vdash \phi(V)}{\mathcal{F}, !\text{cell}\, a\, V \vDash \Gamma^{++}, a : A \mid \phi(\cdot)}$$

Now, if $\cdot \vdash \mathcal{F} \div \Gamma^{++}$*, then* $\mathcal{F} \vDash \Gamma^{++}$*.*

Proof. By right-to-left induction on the configuration typing derivation, we have $\mathcal{F} = \mathcal{F}_1, !\text{cell}\, a\, V$ and $\Gamma^{++} = \Gamma_1^{++}, a : A \mid \phi(\cdot)$ where $\mathcal{F}_1 \vDash \Gamma_1^{++}$. Thus, it suffices to prove $\Gamma_1^{++} \vdash \phi(V)$. By Lemma 3.7, said derivation ends in an instance of CELLV exposing a process typing derivation. By inversion modulo subtyping, it suffices to only consider right axioms, in which case $\phi(V)$ is already assumed in Γ_1^{++} or is proved directly. For example, $\oplus$R assumes $\phi(k \cdot b)$ to type $a.k \cdot b$, whereas $\mathbf{1}$R proves $\Gamma_1^{++} \vdash \phi(\langle\rangle)$ for $a.\langle\rangle$.□

Thus, a well-typed configuration is observably partially correct—it either does not terminate or terminates at a final configuration where all of its purely positive subconfigurations observably satisfy their associated postconditions. We formalize this by the following corollary, which combines a coinductive characterization of type soundness [40] and partial correctness [11,25,46].

Corollary 3.9 (Type Soundness and Observable Partial Correctness) *Let* $\mathcal{F}$ *be* Γ*-safe iff for all* $\Gamma^{++} \subseteq \Gamma$*, there exists* $\mathcal{F}' \subseteq \mathcal{F}$ *such that* $\mathcal{F}' \vDash \Gamma^{++}$*. Then, let* $\mathcal{C}$ *be* Γ*-safe iff, coinductively,* $\mathcal{C} \mapsto \mathcal{C}'$ *and* $\mathcal{C}'$ *is* Γ*-safe. Thus, if* $\cdot \vdash \mathcal{C} \div \Gamma$*, then* $\mathcal{C}$ *is* Γ*-safe.*

We finish by commenting on the generality of our partial correctness result—because hidden addresses can be made observable by projecting or applying the continuations that hide them, we do not lose power by restricting our attention to observability.

4 Related Work

We view DRSAX on a spectrum between languages that *model* concurrency and/or parallelism without native support for them at one end and process calculi with dependent (session) types of varying expressivity at the other. Before we elaborate on this dichotomy, we note that our treatment of codata seems

to be related to logical approaches to object encapsulation in the presence of mutable state [29,30,49]. Moreover, refer to [4,3,5] for reasoning about terminating mixed inductive-coinductive programs.

4.1 Language-Based Verification, Concurrency, and Parallelism

Projects like SteelCore [60] and FCSL [47] implement a variation of *concurrent separation logic* [48,32] in a metalanguage—in these cases, F* or Coq—from which various shared memory and message-passing constructs can be modeled. Similar efforts that do not use separation logic include that in Dafny [37] and Why3 [56]. Our interest is "one level up"—determining a core language that could, in theory, be embedded in the languages discussed via the constructs that they model, intersecting with our discussion of embedded session types below. One exception to this thread is Liquid Effects [33], in which dependent type refinements are retrofitted directly onto a parallel dialect of C.

4.2 Dependent and Embedded Session Types

Toninho et al. [63] initiated the line of work on dependent session types by presenting a session-typed process calculus in Curry-Howard correspondence with first-order intuitionistic linear logic over a domain of non-linear proof terms. In particular, proof terms are not allowed to refer to the channels with which processes communicate in the linear layer. In their retrospective paper ten years later [65], they note that many subsequent developments [66,62,17] have similar restrictions precisely because non-linear dependence on linear objects is problematic. As somewhat of an exception to the rule, Toninho and Yoshida [67] allow proof terms to depend on quoted processes by way of a *contextual monad* [64], related to that of dependent linear/non-linear logic [35]. The relaxation of the restriction on type dependency comes at the cost of process/term-level duplication, since functional terms can be embedded faithfully into processes—DRSAX need not make this distinction.

Another line of work seeks to embed session type systems into existing dependent type theories, allowing meta-level reasoning about processes and the exploitation of existing language infrastructure [7,72,19,57,28,42]. Embedded implementation is certainly not opposed by DRSAX nor the line of work above, but moving the burden of proof to the meta level requires explicit reasoning about the typing and operational semantics of programs to an extent determined by the embedding depth.

5 Conclusion and Future Work

In this paper, we have developed DRSAX, a sound integration of expressive dependent type refinements into SAX, a futures-based process calculus, by adhering to its proof-theoretic discipline. The distinction between data and codata is navigated through the design of the language as well as the metatheory, which begins with typing rules respecting codata encapsulation and culminates in observable partial correctness as a result of type soundness. Moreover, our mixed inductive-coinductive view of (sub)typing gives a uniform treatment of induction, coinduction, and mixed induction and coinduction within the language. There are multiple avenues of future work:

(i) *Types*: we are interested in extending the type structure of DRSAX primarily by abstraction both over types [15] and refinements [69].

(ii) *Effects*: whether there is a proof-theoretic interpretation of various concurrent effects is still an open question. Non-mutable memory reuse can be interpreted with snips [21], thus raising the question of how mutability could be introduced. In the setting of session types, *hypersequents* have been used to introduce races in linear logic [34].

(iii) *Implementation*: the presence of quantifiers in the assertion logic and our mixed inductive-coinductive view of (sub)typing jeopardizes decidable typechecking. With an eye towards implementation, we aim to investigate various quantifier instantiation strategies as well as a restriction to *circular* [12] (sub)typing derivations which are finitely-representable and thus may admit terminating search [17].

References

[1] Abel, A., B. Pientka, D. Thibodeau and A. Setzer, *Copatterns: Programming Infinite Structures by Observations*, in: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13 (2013), p. 27–38.
URL https://doi.org/10.1145/2429069.2429075

[2] Aspinall, D. and A. Compagnoni, *Subtyping dependent types*, in: *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, 1996, pp. 86–97.

[3] Basold, H., "Mixed Inductive-Coinductive Reasoning: Types, Programs and Logic," Ph.D. thesis, Radboud University (2018).

[4] Basold, H. and H. Geuvers, *Type Theory Based on Dependent Inductive and Coinductive Types*, in: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16 (2016), p. 327–336.
URL https://doi.org/10.1145/2933575.2934514

[5] Basold, H. and H. H. Hansen, *Well-definedness and observational equivalence for inductive–coinductive programs*, Journal of Logic and Computation **29** (2019), pp. 419–468.
URL https://doi.org/10.1093/logcom/exv091

[6] Bell, C. J. and A. Chlipala, *A Coinduction Proof Rule for Hoare Doubles*, in: *The 2nd International Workshop on Coq for PL (CoqPL 2016)*, 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016), 2016.

[7] Brady, E. and K. Hammond, *Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols*, Fundam. Inf. **102** (2010), p. 145–176.

[8] Brandt, M. and F. Henglein, *Coinductive Axiomatization of Recursive Type Equality and Subtyping*, in: P. de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications* (1997), pp. 63–81.

[9] Brotherston, J., *Cyclic Proofs for First-Order Logic with Inductive Definitions*, in: B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods* (2005), pp. 78–92.

[10] Cervesato, I. and A. Scedrov, *Relating State-Based and Process-Based Concurrency through Linear Logic*, Information and Computation **207** (2009), pp. 1044–1077, special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

[11] Clarke, E. M., *Program Invariants as Fixed Points (Preliminary Reports)*, in: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977* (1977), pp. 18–29.
URL https://doi.org/10.1109/SFCS.1977.25

[12] Dagnino, F., *Foundations of regular coinduction*, Logical Methods in Computer Science **Volume 17, Issue 4** (2021).
URL https://lmcs.episciences.org/8539

[13] Danielsson, N. A. and T. Altenkirch, *Mixing Induction and Coinduction* (2009).

[14] Danielsson, N. A. and T. Altenkirch, *Subtyping, Declaratively*, in: C. Bolduc, J. Desharnais and B. Ktari, editors, *Mathematics of Program Construction* (2010), pp. 100–118.

[15] Das, A., H. Deyoung, A. Mordido and F. Pfenning, *Nested Session Types*, ACM Trans. Program. Lang. Syst. **44** (2022).
URL https://doi.org/10.1145/3539656

[16] Das, A., J. Hoffmann and F. Pfenning, *Parallel Complexity Analysis with Temporal Session Types*, Proc. ACM Program. Lang. **2** (2018).
URL https://doi.org/10.1145/3236786

[17] Das, A. and F. Pfenning, *Session Types with Arithmetic Refinements*, in: I. Konnov and L. Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, Leibniz International Proceedings in Informatics (LIPIcs) **171** (2020), pp. 13:1–13:18.

[18] de Bruijn, N., *Telescopic mappings in typed lambda calculus*, Information and Computation **91** (1991), pp. 189–204.
URL https://www.sciencedirect.com/science/article/pii/089054019190066B

[19] de Muijnck-Hughes, J., E. C. Brady and W. Vanderbauwhede, *Value-Dependent Session Design in a Dependently Typed Language*, in: F. Martins and D. Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, EPTCS **291**, 2019, pp. 47–59.
URL https://doi.org/10.4204/EPTCS.291.5

[20] Derakhshan, F. and F. Pfenning, *Circular Proofs as Session-Typed Processes: A Local Validity Condition*, CoRR **abs/1908.01909** (2019).

[21] DeYoung, H. and F. Pfenning, *Data Layout from a Type-Theoretic Perspective*, Electronic Notes in Theoretical Informatics and Computer Science **Volume 1 - Proceedings of MFPS XXXVIII** (2023).
URL https://entics.episciences.org/10507

[22] DeYoung, H., F. Pfenning and K. Pruiksma, *Semi-Axiomatic Sequent Calculus*, in: Z. M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, Leibniz International Proceedings in Informatics (LIPIcs) **167** (2020), pp. 29:1–29:22.

[23] Dunfield, J. and N. Krishnaswami, *Bidirectional Typing*, ACM Comput. Surv. **54** (2021).
URL https://doi.org/10.1145/3450952

[24] Dunfield, J. and F. Pfenning, *Tridirectional Typechecking*, in: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04 (2004), p. 281–292.
URL https://doi.org/10.1145/964001.964025

[25] Goguen, J. A. and G. Malcolm, *Hidden coinduction: behavioural correctness proofs for objects*, Mathematical Structures in Computer Science **9** (1999), p. 287–319.

[26] Halstead, R. H., *MULTILISP: A Language for Concurrent Symbolic Computation*, ACM Trans. Program. Lang. Syst. **7** (1985), p. 501–538.

[27] Harper, R., "Practical Foundations for Programming Languages," Cambridge University Press, 2016, 2 edition.

[28] Hinrichsen, J. K., J. Bengtson and R. Krebbers, *Actris: Session-Type Based Reasoning in Separation Logic*, Proc. ACM Program. Lang. **4** (2019).
URL https://doi.org/10.1145/3371074

[29] Hoare, C. A. R., *Proof of Correctness of Data Representations*, Acta Informatica **1** (1972), pp. 271–281.
URL https://doi.org/10.1007/BF00289507

[30] Hoare, C. A. R., "Towards a Theory of Parallel Programming," Springer New York, New York, NY, 2002 pp. 231–244.
URL https://doi.org/10.1007/978-1-4757-3472-0_6

[31] Jhala, R. and N. Vazou, *Refinement Types: A Tutorial*, Found. Trends Program. Lang. **6** (2021), p. 159–317.
URL https://doi.org/10.1561/2500000032

[32] Jung, R., R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal and D. Dreyer, *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*, Journal of Functional Programming **28** (2018), p. e20.

[33] Kawaguchi, M., P. Rondon, A. Bakst and R. Jhala, *Deterministic Parallelism via Liquid Effects*, in: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12 (2012), p. 45–54.
URL https://doi.org/10.1145/2254064.2254071

[34] Kokke, W., J. G. Morris and P. Wadler, *Towards Races in Linear Logic*, in: H. Riis Nielson and E. Tuosto, editors, *Coordination Models and Languages* (2019), pp. 37–53.

[35] Krishnaswami, N. R., P. Pradic and N. Benton, *Integrating Linear and Dependent Types*, in: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15 (2015), p. 17–30.

[36] Lakhani, Z., A. Das, H. DeYoung, A. Mordido and F. Pfenning, *Polarized Subtyping*, in: I. Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, Lecture Notes in Computer Science **13240** (2022), pp. 431–461.
URL https://doi.org/10.1007/978-3-030-99336-8_16

[37] Leino, K. R. M., *Modeling Concurrency in Dafny*, in: J. P. Bowen, Z. Liu and Z. Zhang, editors, *Engineering Trustworthy Software Systems* (2018), pp. 115–142.

[38] Leino, K. R. M. and M. Moskal, *Co-induction simply*, in: C. Jones, P. Pihlajasaari and J. Sun, editors, *FM 2014: Formal Methods* (2014), pp. 382–398.

[39] Lengrand, S., R. Dyckhoff and J. McKinna, *A Sequent Calculus for Type Theory*, in: Z. Ésik, editor, *Computer Science Logic* (2006), pp. 441–455.

[40] Leroy, X., *Coinductive big-step operational semantics*, in: *ESOP 2006: European Symposium on Programming*, number 3924 in LNCS (2006), pp. 54–68.

[41] Levy, P. B., *Call-by-Push-Value: A Subsuming Paradigm*, in: J.-Y. Girard, editor, *Typed Lambda Calculi and Applications* (1999), pp. 228–243.

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

[43] Martin-Löf, P., *On the Meanings of the Logical Constants and the Justifications of the Logical Laws*, Nordic Journal of Philosophical Logic **1** (1996), pp. 11–60.

[44] Mastorou, L., N. Papaspyrou and N. Vazou, *Coinduction Inductively: Mechanizing Coinductive Proofs in Liquid Haskell*, in: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, Haskell 2022 (2022), p. 1–12.
URL https://doi.org/10.1145/3546189.3549922

[45] Momigliano, A. and A. Tiu, *Induction and Co-induction in Sequent Calculus*, in: S. Berardi, M. Coppo and F. Damiani, editors, *Types for Proofs and Programs* (2004), pp. 293–308.

[46] Moore, B., L. Peña and G. Rosu, *Program Verification by Coinduction*, in: A. Ahmed, editor, *Proceedings of the 27th European Symposium on Programming (ESOP 2018) held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2018)*, 2018, pp. 589–618.

[47] Nanevski, A., R. Ley-Wild, I. Sergey and G. A. Delbianco, *Communicating State Transition Systems for Fine-Grained Concurrent Resources*, in: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410* (2014), p. 290–310.
URL https://doi.org/10.1007/978-3-642-54833-8_16

[48] O'Hearn, P. W., *Resources, Concurrency and Local Reasoning*, in: P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory* (2004), pp. 49–67.

[49] O'Hearn, P. W., H. Yang and J. C. Reynolds, *Separation and Information Hiding*, ACM Trans. Program. Lang. Syst. **31** (2009).
URL https://doi.org/10.1145/1498926.1498929

[50] Oppen, D. C., *Reasoning about Recursively Defined Data Structures*, in: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78 (1978), p. 151–157.
URL https://doi.org/10.1145/512760.512776

[51] Pierce, B. C. and D. N. Turner, *Local Type Inference*, ACM Trans. Program. Lang. Syst. **22** (2000), p. 1–44.
URL https://doi.org/10.1145/345099.345100

[52] Régis-Gianas, Y., "Des types aux assertions logiques : preuve automatique ou assistée de propriétés sur les programmes fonctionnels," Theses, Université Paris Diderot (2007).
URL https://hal.inria.fr/tel-01238703

[53] Régis-Gianas, Y. and F. Pottier, *A Hoare Logic for Call-by-Value Functional Programs*, in: P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction* (2008), pp. 305–335.

[54] Rondon, P. M., M. Kawaguci and R. Jhala, *Liquid Types*, SIGPLAN Not. **43** (2008), p. 159–169.
URL https://doi.org/10.1145/1379022.1375602

[55] Rushby, J., S. Owre and N. Shankar, *Subtypes for specifications: predicate subtyping in PVS*, IEEE Transactions on Software Engineering **24** (1998), pp. 709–720.

[56] Santos, C., F. Martins and V. T. Vasconcelos, *Deductive Verification of Parallel Programs Using Why3*, in: S. Knight, I. Lanese, A. Lluch-Lafuente and H. T. Vieira, editors, *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015*, EPTCS **189**, 2015, pp. 128–142.
URL https://doi.org/10.4204/EPTCS.189.11

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

[58] Smullyan, R. M., *Analytic cut*, The Journal of Symbolic Logic **33** (1969), p. 560–564.

[59] Somayyajula, S. and F. Pfenning, *Type-Based Termination for Futures*, in: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, 2022.

[60] Swamy, N., A. Rastogi, A. Fromherz, D. Merigoux, D. Ahman and G. Martínez, *SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs*, Proc. ACM Program. Lang. **4** (2020).
URL https://doi.org/10.1145/3409003

[61] Tennant, N., "Natural Logic," Edinburgh University Press, 1978.

[62] Thiemann, P. and V. T. Vasconcelos, *Label-Dependent Session Types*, Proc. ACM Program. Lang. **4** (2019).
URL https://doi.org/10.1145/3371135

[63] Toninho, B., L. Caires and F. 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 (2011), p. 161–172.
URL https://doi.org/10.1145/2003476.2003499

[64] Toninho, B., L. Caires and F. Pfenning, *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*, in: M. Felleisen and P. Gardner, editors, *Programming Languages and Systems* (2013), pp. 350–369.

[65] Toninho, B., L. Caires and F. Pfenning, *A Decade of Dependent Session Types*, in: *23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2021 (2021).
URL https://doi.org/10.1145/3479394.3479398

[66] Toninho, B. and N. Yoshida, *Certifying data in multiparty session types*, Journal of Logical and Algebraic Methods in Programming **90** (2017), pp. 61–83.
URL https://www.sciencedirect.com/science/article/pii/S2352220816300864

[67] Toninho, B. and N. Yoshida, *Depending on Session-Typed Processes*, in: C. Baier and U. D. Lago, editors, *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*, Lecture Notes in Computer Science **10803** (2018), pp. 128–145.
URL https://doi.org/10.1007/978-3-319-89366-2_7

[68] Vazou, N. and M. Greenberg, *How to Safely Use Extensionality in Liquid Haskell*, in: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, Haskell 2022 (2022), p. 13–26.
URL https://doi.org/10.1145/3546189.3549919

[69] Vazou, N., P. M. Rondon and R. Jhala, *Abstract Refinement Types*, in: M. Felleisen and P. Gardner, editors, *Programming Languages and Systems* (2013), pp. 209–228.

[70] Vazou, N., E. L. Seidel, R. Jhala, D. Vytiniotis and S. Peyton-Jones, *Refinement Types for Haskell*, in: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14 (2014), p. 269–282.
URL https://doi.org/10.1145/2628136.2628161

[71] Vazou, N., A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler and R. Jhala, *Refinement Reflection: Complete Verification with SMT*, Proc. ACM Program. Lang. **2** (2017).

[72] Wu, H. and H. Xi, *Dependent Session Types*, CoRR **abs/1704.07004** (2017).
URL http://arxiv.org/abs/1704.07004

[73] Xi, H. and F. Pfenning, *Dependent Types in Practical Programming*, in: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99 (1999), p. 214–227.
URL https://doi.org/10.1145/292540.292560

[74] Zeilberger, N., *Balanced polymorphism and linear lambda calculus* (2015), http://noamz.org/papers/linprin.pdf.