Session Types with Arithmetic Refinements

Ankush Das
Carnegie Mellon University, Pittsburgh, PA, USA
http://www.cs.cmu.edu/~ankushd
ankushd@cs.cmu.edu

Frank Pfenning
Carnegie Mellon University, Pittsburgh, PA, USA
http://www.cs.cmu.edu/~fp
fp@cs.cmu.edu

Abstract

Session types statically prescribe bidirectional communication protocols for message-passing processes. However, simple session types cannot specify properties beyond the type of exchanged messages. In this paper we extend the type system by using index refinements from linear arithmetic capturing intrinsic attributes of data structures and algorithms. We show that, despite the decidability of Presburger arithmetic, type equality and therefore also subtyping and type checking are now undecidable, which stands in contrast to analogous dependent refinement type systems from functional languages. We also present a practical, but incomplete algorithm for type equality, which we have used in our implementation of Rast, a concurrent session-typed language with arithmetic index refinements as well as ergometric and temporal types. Moreover, if necessary, the programmer can propose additional type bisimulations that are smoothly integrated into the type equality algorithm.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Linear logic; Theory of computation → Logic and verification; Computing methodologies → Concurrent programming languages; Theory of computation → Type theory

Keywords and phrases Session Types, Refinement Types, Type Equality

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2020.13

Funding *Ankush Das*: funded by the National Science Foundation under SaTC Award 1801369 and CAREER Award 1845514.
Frank Pfenning: funded by the National Science Foundation under Grant No. 1718276.

1 Introduction

Session types [24, 42] provide a structured way of prescribing communication protocols in message-passing systems. This paper focuses on *binary session types* governing the interactions along channels with two endpoints. They arise either directly as part of a program notation [25], or as the result of *endpoint projection* of multi-party session types [26] and are thus of central importance in the study of message-passing concurrency. Moreover, a Curry-Howard correspondence relates propositions of linear logic to session types [8, 43, 9], further evidence for their fundamental nature.

Once recursion is introduced for session types as well as processes, we are confronted with the question as to what is the correct notion of type equality since its use in type checking is inescapable. Gay and Hole [17] convincingly answer this question and also provide a practical algorithm for subtyping (which implies an algorithm for type equality). First, since the endpoints of channels need to agree on a type (or possibly two dual types) for communication, recursive types should be a priori *structural* rather than *nominal*. Second, types should be equal if their observable communication behaviors are indistinguishable. This means that two types should be equal if there is a *bisimulation* between them. This is particularly elegant since the definition is independent of any particular programming

language in which session types are embedded, or whether they are checked statically or dynamically. The algorithm for type equality then constructs a bisimulation. It terminates because the number of pairs of types that might be related by the bisimulation is finite.

Like any type system, basic session types are limited in the kind of properties they can express, which has led to some generalizations such as polymorphic [43, 7, 21] and context-free [38] session types, each with its own questions for type equality. In this paper we propose a natural *linear arithmetic refinement* of session types, which allows us to capture a number of significant properties of message-passing communication such as size or value of data structures, number of messages exchanged or delay in those messages. In conception, this refinement is closely related to *indexed types* or *value-dependent types* familiar from functional languages [47, 46, 37], where the indices are arithmetic expressions.

To our surprise, despite an eminently decidable index domain, the type equality problem becomes undecidable. We show this via a reduction from the non-halting problem for two-counter machines [30]. Analyzing this reduction in detail shows that the problem is already undecidable for a single type constructor (pick either internal ($\oplus$) or external ($\&$) choice, in addition to arithmetic refinements). While our type system is equirecursive to aid in the simplicity of programming, even retreating to isorecursive types leaves the problem undecidable. Finally, one may be tempted to blame the quantifiers in Presburger arithmetic, but our reduction shows that even if we restrict ourselves to linear arithmetic with universal prefix quantification only, type equality remains undecidable.

A retrenchment to a *nominal* interpretation of recursive types would rule out too many programs and complicate communications, so we develop a sound but incomplete algorithm. Our experience with the Rast implementation [14] to date shows that it is effective in practice (see Section 6 for further discussion).

Most closely related is the design of LiquidPi [22], but it refines only basic data types such as int rather than equirecursively defined session types. The resulting system has a decidable subtyping problem and even type inference (under reasonable assumptions on the constraint domain), but it cannot express many of our motivating examples. Along similar lines, refinements of basic data types together with subtyping have been proposed for *runtime monitoring* of binary session-typed communication [20, 19]. Label-dependent session types [39] also support types indexed by natural numbers using a fixed schema of iteration with a particular unfolding equality, rather than arbitrary recursion and bisimulation. Zhou et al. [49, 48] refine base types with arithmetic expressions in the context of multiparty session types without recursive types. In this simpler setting, they obtain a decidable notion of type equality. Further related work can be found in Section 7.

2 Basic Session Types

We review the basic language of binary session types. We take the intuitionistic point of view [8, 9], since our experiments and motivating examples have been carried out in Rast [14]. Changes for a classical view [43] are minimal and do not affect our results or algorithms. We would add a type $\bot$ dual to $\mathbf{1}$, and replace the $\multimap$ operator with $\parr$ with only minor changes to the remainder of the development.

A, B, C	$::=$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send label $k \in L$	continue at type A_k
	$\mid$	$\&\{\ell : A_\ell\}_{\ell \in L}$	receive label $k \in L$	continue at type A_k
	$\mid$	$A \otimes B$	send channel $a : A$	continue at type B
	$\mid$	$A \multimap \mathrm{B}$	receive channel $a : A$	continue at type B
	$\mid$	$\mathbf{1}$	send close message	no continuation
	$\mid$	V	defined type variable	

We provide a brief description of the operational behavior of the types from the provider's point of view. The provider of $\oplus\{\ell : A_\ell\}_{\ell\in L}$ sends a label $k \in L$ and continues to provide A_k. Dually, the provider of $\&\{\ell : A_\ell\}_{\ell\in L}$ receives one of the labels in L. The provider of $A \otimes B$ sends a channel of type A and continues to provide B, whereas the process providing $A \multimap B$ receives a channel of type A and provides B. Finally, the provider of $\mathbf{1}$ sends a close message and terminates.

We assume that labels $\ell \in L$ (for a finite, nonempty set L) and close messages can be observed, but the identity of channels can not. Instead any *communication* along channels that are sent and received can be observed in turn. Based on this notion, we adopt *type bisimulations* from Gay and Hole [17]. Rather than an explicit recursive type constructor μ we postulate a *signature* Σ with definitions of type variables V.

$$\text{Signature} \quad \Sigma \quad ::= \quad \cdot \mid \Sigma, V = A$$

In a *valid signature* all definitions $V = A$ are *contractive*, that is, A is not itself a type variable. This allows us to take an *equirecursive* view of type definitions, which means that unfolding a type definition does not require communication. We can easily adapt our definitions to an *isorecursive* view [28, 15] with explicit unfold messages (see the remark at the end of Section 4). All type variables V occurring in a valid signature may refer to each other and must be defined, and all type variables defined in a signature must be distinct.

▶ **Definition 1.** *We define* $\mathsf{unfold}_\Sigma(V) = A$ *if* $V = A \in \Sigma$ *and* $\mathsf{unfold}_\Sigma(A) = A$ *otherwise.*

▶ **Definition 2.** *A relation* $\mathcal{R}$ *on types is a* type bisimulation *if* $(A, B) \in \mathcal{R}$ *implies that for* $S = \mathsf{unfold}_\Sigma(A)$, $T = \mathsf{unfold}_\Sigma(B)$ *we have*

- *If* $S = \oplus\{\ell : A_\ell\}_{\ell\in L}$ *then* $T = \oplus\{\ell : B_\ell\}_{\ell\in L}$ *and* $(A_\ell, B_\ell) \in \mathcal{R}$ *for all* $\ell \in L$.
- *If* $S = \&\{\ell : A_\ell\}_{\ell\in L}$ *then* $T = \&\{\ell : B_\ell\}_{\ell\in L}$ *and* $(A_\ell, B_\ell) \in \mathcal{R}$ *for all* $\ell \in L$.
- *If* $S = A_1 \otimes A_2$, *then* $T = B_1 \otimes B_2$ *and* $(A_1, B_1) \in \mathcal{R}$ *and* $(A_2, B_2) \in \mathcal{R}$.
- *If* $S = A_1 \multimap A_2$, *then* $T = B_1 \multimap B_2$ *and* $(A_1, B_1) \in \mathcal{R}$ *and* $(A_2, B_2) \in \mathcal{R}$.
- *If* $S = \mathbf{1}$ *then* $T = \mathbf{1}$.

▶ **Definition 3.** *We say that* A *is* equal *to* B, *written* $A \equiv B$, *if there is a type bisimulation* $\mathcal{R}$ *such that* $(A, B) \in \mathcal{R}$.

As two simple running examples we use an interface to a queue and the representation of binary numbers as sequences of bits.

▶ **Example 4** (Queues, v1)**.** A queue provider offers a choice (indicated by &) of either receiving an **ins** label followed by a channel of type A (denoted by $\multimap$) to insert into the queue, or a **del** label to delete an element from the queue. In the latter case, the queue provider has a choice (indicated by $\oplus$) of either responding with the label **none** (if there is no element in the queue) and closes the channel (indicated by $\mathbf{1}$), or the label some followed by an element of type A (denoted by $\otimes$) and recurses to await the next round of interactions. We view queue_A as a family of types, one for each A, to avoid introducing explicit polymorphic type constructors.

$$\begin{aligned}\mathsf{queue}_A = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}_A,\\ &\mathbf{del} : \oplus\{\mathbf{none} : \mathbf{1},\\ &\qquad \mathbf{some} : A \otimes \mathsf{queue}_A\}\}\end{aligned}$$

▶ **Example 5** (Binary Numbers, v1)**.** A process representing a binary number either sends a label **e** representing the number 0 and closes the channel, or one of the labels **b0** (bit 0) or **b1** (bit 1) followed by remaining bits (by recursing). We assume a "little endian" form, that is, the least significant bit is sent first.

$$\mathsf{bin} = \oplus\{\mathbf{b0} : \mathsf{bin}, \mathbf{b1} : \mathsf{bin}, \mathbf{e} : \mathbf{1}\}$$

As examples of message sequences along a fixed channel, we would have

$\mathbf{e}$; close	representing 0
$\mathbf{b0}$; $\mathbf{e}$; close	also representing 0
$\mathbf{b0}$; $\mathbf{b1}$; $\mathbf{e}$; close	representing 2
$\mathbf{b1}$; $\mathbf{b0}$; $\mathbf{b1}$; $\mathbf{b1}$; $\mathbf{e}$; close	representing 13

3 Arithmetic Refinements

Before we extend our language of types formally, we revisit the examples in order to motivate the specific constructs available. We write $V[\overline{e}]$ for a type indexed by a sequence of arithmetic expressions e. Since it has been appropriate for most of our examples, we restrict ourselves to natural numbers rather than arbitrary integers.

▶ **Example 6** (Queues, v2)**.** The provider of a queue should be constrained to answer **none** exactly if the queue contains no elements and **some** if it is nonempty. The queue type from Example 4 does not express this. This means a client may need to have some redundant branches to account for responses that should be impossible. We now define the type $\mathsf{queue}_A[n]$ to stand for a queue with exactly n elements.

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}_A[n+1],\\ &\mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 0\}.\, \mathbf{1},\\ &\qquad \mathbf{some} : ?\{n > 0\}.\, A \otimes \mathsf{queue}_A[n-1]\}\}\end{aligned}$$

The first branch is easy to understand: if we add an element to a queue of length n, it subsequently contains $n + 1$ elements. In the second branch we *constrain* the arithmetic variable n to be equal to 0 if the provider sends **none** and positive if the provider sends **some**. In the latter case, we subtract one from the length after an element has been dequeued.

Conceptually, the type $?\{\phi\}.\, A$ means that the provider must *send* a proof p of ϕ, so it corresponds to $\exists p : \phi.\, A$. A characteristic of *type refinement*, in contrast to fully dependent types, is that the computation of A and thus, the execution of processes can only depend on the *existence* of a proof, but not on its form (known in type theory as *proof irrelevance*). More concretely, the process types and terms cannot refer to the proof p. This irrelevance property combined with the decidability of our index domain means that no actual proof needs to be sent (since one can be constructed from ϕ automatically, if needed), just a token *asserting* its existence. There is also a dual constructor $!\{\phi\}.\, A$ that licenses the *assumption* of ϕ, which, conceptually, corresponds to *receiving* a proof of ϕ.

▶ **Example 7** (Binary Numbers, v2)**.** The indexed type $\mathsf{bin}[n]$ should represent a binary number with value n. Because the least significant bit comes first, we expect, for example, that $\mathsf{bin}[n] = \oplus\{\mathbf{b0} : ?\{2 \mid n\}.\, \mathsf{bin}[n/2], \ldots\}$ ($a \mid b$ denotes a divides b). However, while divisibility is available in Presburger arithmetic, division itself is not; instead, we can express the constraint and the index of the recursive occurrence using quantification.

$$\begin{aligned}\mathsf{bin}[n] = \oplus\{&\mathbf{b0} : \exists k.\, ?\{n = 2 * k\}.\, \mathsf{bin}[k],\\ &\mathbf{b1} : \exists k.\, ?\{n = 2 * k + 1\}.\, \mathsf{bin}[k],\\ &\mathbf{e} : ?\{n = 0\}.\, \mathbf{1}\}\end{aligned}$$

As a further refinement, we could rule out leading zeros by adding the constraint $n > 0$ in the branch for **b0** (in branch **b1**, $n = 2k + 1$ implies $n > 0$ so the constraint implicitly holds).

The type $\exists n.\, A$ means that the provider must send a natural number i and proceed at type $A[i/n]$, corresponding to existential quantification in arithmetic. The dual universal quantifier $\forall n.\, A$ requires the provider to receive a number i and proceed at type $A[i/n]$.

We now extend our definitions to account for these new constructs. Below, i represents a constant, n is a natural number variable and $(i \mid e)$ means i divides e.

Types	A	$::=$	$\ldots$		
		$\mid$	$?\{\phi\}.\, A$	assert ϕ	continue at type A
		$\mid$	$!\{\phi\}.\, A$	assume ϕ	continue at type A
		$\mid$	$\exists n.\, A$	send number i	continue at type $A[i/n]$
		$\mid$	$\forall n.\, A$	receive number i	continue at type $A[i/n]$
		$\mid$	$V[\overline{e}]$	variable instantiation	
Arith. Expressions	e	$::=$	$i \mid e + e \mid e - e \mid i \times e \mid (i \mid e) \mid n$		
Arith. Propositions	ϕ	$::=$	$e = e \mid e > e \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \exists n.\, \phi \mid \forall n.\, \phi$		
Signature	Σ	$::=$	$\cdot \mid \Sigma, V[\overline{n} \mid \phi] = A$		

An indexed type definition $V[\overline{n} \mid \phi] = A$ containing an optional proposition ϕ requires every instantiation $\overline{e}$ (in $V[\overline{e}]$) of the sequence of variables $\overline{n}$ to satisfy $\phi[\overline{e}/\overline{n}]$. This is verified statically when a type signature is checked for validity, as defined below. We use $\mathcal{V}$ for a collection of arithmetic variables and $\mathcal{C}$ (to signify *constraints*) for an arithmetic proposition occurring among the antecedents of a judgment. We then have the following rules defining the validity of signatures ($\vdash \Sigma$ *signature*), declarations ($\vdash_\Sigma \Sigma'$ *valid*), and types ($\mathcal{V}\ ;\ \mathcal{C} \vdash_\Sigma A$ *valid*) where $\mathcal{V}$ is a collection of arithmetic variables including all free variables in constraint $\mathcal{C}$ and type A. We silently rename variables so that n does not already occur in $\mathcal{V}$ in the $\exists V$ and $\forall V$ rules. We also call upon the semantic entailment judgment $\mathcal{V}\ ;\ \mathcal{C} \vDash \phi$ which means that $\forall \mathcal{V}.\, \mathcal{C} \supset \phi$ holds in arithmetic and $\vDash \phi$ abbreviates $\cdot\ ;\ \top \vDash \phi$.

$$\frac{\vdash_\Sigma \Sigma\ \mathit{valid}}{\vdash \Sigma\ \mathit{signature}} \qquad \frac{}{\vdash_\Sigma (\cdot)\ \mathit{valid}} \qquad \frac{\vdash_\Sigma \Sigma'\ \mathit{valid} \quad \overline{n}\ ;\ \phi \vdash_\Sigma A\ \mathit{valid} \quad A \neq V'[\overline{e}']}{\vdash_\Sigma \Sigma', V[\overline{n} \mid \phi] = A\ \mathit{valid}}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \wedge \phi \vdash_\Sigma A\ \mathit{valid}}{\mathcal{V}\ ;\ \mathcal{C} \vdash_\Sigma ?\{\phi\}.\, A\ \mathit{valid}}\ ?V \qquad \frac{\mathcal{V}\ ;\ \mathcal{C} \wedge \phi \vdash_\Sigma A\ \mathit{valid}}{\mathcal{V}\ ;\ \mathcal{C} \vdash_\Sigma !\{\phi\}.\, A\ \mathit{valid}}\ !V$$

$$\frac{\mathcal{V}, n\ ;\ \mathcal{C} \vdash_\Sigma A\ \mathit{valid}}{\mathcal{V}\ ;\ \mathcal{C} \vdash_\Sigma \exists n.\, A\ \mathit{valid}}\ \exists V^n \qquad \frac{\mathcal{V}, n\ ;\ \mathcal{C} \vdash_\Sigma A\ \mathit{valid}}{\mathcal{V}\ ;\ \mathcal{C} \vdash_\Sigma \forall n.\, A\ \mathit{valid}}\ \forall V^n$$

$$\frac{V[\overline{n} \mid \phi] = A \in \Sigma \quad \mathcal{V}\ ;\ \mathcal{C} \vDash \phi[\overline{e}/\overline{n}]}{\mathcal{V}\ ;\ \mathcal{C} \vdash_\Sigma V[\overline{e}]\ \mathit{valid}}\ \mathsf{tdef}$$

We elide the compositional rules for all the other type constructors. Since we like to work over natural numbers rather than integers, it is convenient to assume that every definition $V[\overline{n}] = A$ abbreviates $V[\overline{n} \mid \overline{n} \geq 0] = A$. This means that in valid signatures every occurrence $V[\overline{e}]$ is such that $\overline{e} \geq 0$ follows from the known constraints.

▶ **Example 8.** The declaration

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}_A[n+1],\\ &\mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 0\}.\, \mathbf{1},\\ &\qquad \mathbf{some} : ?\{n > 0\}.\, A \otimes \mathsf{queue}_A[n-1]\}\}\end{aligned}$$

is valid: in the **ins** branch, we verify $(n\ ;\ n \geq 0 \vDash n + 1 \geq 0)$ while checking validity of $\mathsf{queue}_A[n+1]$ with rule tdef; in the **some** branch, we add $n > 0$ to our constraint $\mathcal{C}$ (due to rule $?V$) and verify $(n\ ;\ n \geq 0 \wedge n > 0 \vDash n - 1 \geq 0)$ while checking validity of $\mathsf{queue}_A[n-1]$.

Unfolding a definition must now substitute for the arithmetic variables we abstract over.

▶ **Definition 9.** $\mathsf{unfold}_\Sigma(V[\overline{e}]) = A[\overline{e}/\overline{n}]$ *if* $V[\overline{n} \mid \phi] = A \in \Sigma$ *and* $\mathsf{unfold}_\Sigma(A) = A$ *otherwise.*

We say that a type is *closed* if it contains no free arithmetic variables n.

▶ **Definition 10.** *A relation $\mathcal{R}$ on closed valid types is a type bisimulation if $(A, B) \in \mathcal{R}$ implies that for $S = \mathsf{unfold}_\Sigma(A)$, $T = \mathsf{unfold}_\Sigma(B)$ we have the following conditions (in addition to those of Definition 2):*

- *If $S = ?\{\phi\}.\, A'$ then $T = ?\{\psi\}.\, B'$ and either (i) $\vDash \phi$, $\vDash \psi$, and $(A', B') \in \mathcal{R}$, or (ii) $\vDash \neg\phi$ and $\vDash \neg\psi$.*
- *If $S = !\{\phi\}.\, A'$ then $T = !\{\psi\}.\, B'$ and either (i) $\vDash \phi$, $\vDash \psi$, and $(A', B') \in \mathcal{R}$, or (ii) $\vDash \neg\phi$ and $\vDash \neg\psi$*
- *If $S = \exists m.\, A'$ then $T = \exists n.\, B'$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in \mathcal{R}$.*
- *If $S = \forall m.\, A'$ then $T = \forall n.\, B'$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in \mathcal{R}$.*

We also extend the notation $A \equiv B$ to this richer set of types.

An interesting point here is provided by the cases *(ii)* in the first two clauses. Because the type must be closed, we know that ϕ and ψ will be either true or false. If both are false, no messages can be sent along a channel of either type and therefore the continuation types A' and B' are irrelevant when considering type equality.

Fundamentally, due to the presence of arbitrary recursion and therefore non-termination, we always view a type as a restriction of what a process *might* send or receive along some channel, but it is neither *required* to send a message nor *guaranteed* to receive one. This is similar to functional programming with unrestricted recursion where an expression may not return a value. The definition based on observability of messages is then somewhat strict, as exemplified by the next example.

▶ **Example 11.** Consider the following two types

$$\begin{aligned}\mathsf{bin}[n] = \oplus\{&\mathbf{b0} : \exists k.\, ?\{n = 2 * k\}.\, \mathsf{bin}[k],\\ &\mathbf{b1} : \exists k.\, ?\{n = 2 * k + 1\}.\, \mathsf{bin}[k],\\ &\mathbf{e} : ?\{n = 0\}.\, \mathbf{1}\}\end{aligned} \qquad \begin{aligned}\mathsf{zero} = \oplus\{&\mathbf{b0} : \exists k.\, ?\{k = 0\}.\, \mathsf{zero},\\ &\mathbf{e} : ?\{0 = 0\}.\, \mathbf{1}\}\end{aligned}$$

We might expect $\mathsf{bin}[0] \equiv \mathsf{zero}$, but this is not so. A process of type $\mathsf{bin}[0]$ could send the label **b1** and an arbitrary value for k and then just loop forever (because there is no proof of $0 = 2k + 1$). The type zero cannot exhibit this behavior so the types are not equivalent.

In our implementation, missing branches for a choice in process definitions are *reconstructed* with a continuation that marks it as impossible, which is then verified by the type checker. We found this simple technique significantly limited the need for subtyping or explicit definition of types such as zero – instead, we just work with $\mathsf{bin}[0]$.

The following properties of type equality are straightforward.

▶ **Lemma 12** (Properties of Type Equality)**.** *The relation $\equiv$ is reflexive, symmetric, transitive and a congruence on closed valid types.*

4 Undecidability of Type Equality

We prove the undecidability of type equality by exhibiting a reduction from an undecidable problem about two counter machines.

The type system allows us to simulate two counter machines [30]. Intuitively, arithmetic constraints allow us to model branching zero-tests available in the machine. This, coupled with recursion in the language of types, establishes undecidability. Remarkably, a small fragment

of our language containing only type definitions, internal choice ($\oplus$) and assertions ($?\{\phi\}.\, A$) where ϕ just contains constraints $n = 0$ and $n > 0$ is sufficient to prove undecidability. Moreover, the proof still applies if we treat types isorecursively. In the remainder of this section we provide some details of the undecidability proof.

▶ **Definition 13** (Two Counter Machine). *A two counter machine $\mathcal{M}$ is defined as a sequence of instructions $\iota_1, \iota_2, \ldots, \iota_m$ and c_j ($j \in \{1, 2\}$) as the two counters. Each instruction has one of the following forms.*

- *"*$\mathsf{inc}(c_j); \mathsf{goto}\ k$*" (increment counter j by 1 and go to instruction k)*
- *"*$\mathsf{zero}(c_j)?\ \mathsf{goto}\ k : \mathsf{dec}(c_j); \mathsf{goto}\ l$*" (if the value of counter j is 0, go to instruction k, else decrement the counter by 1 and go to instruction l)*
- *"*halt*" (stop computation)*

A configuration C of the machine $\mathcal{M}$ is defined as a triple (i, c_1, c_2), where i denotes the number of the current instruction and c_j's denote the value of the counters. A configuration C' is defined as the successor configuration of C, written as $C \mapsto C'$ if C' is the result of executing the i-th instruction on C. If $\iota_i = \mathsf{halt}$, then $C = (i, c_1, c_2)$ has no successor configuration. The computation of $\mathcal{M}$ is the unique maximal sequence $\rho = \rho(0)\rho(1)\ldots$ such that $\rho(i) \mapsto \rho(i+1)$ and $\rho(0) = (1, 0, 0)$. Either ρ is infinite, or ends in (i, c_1, c_2) such that $\iota_i = \mathsf{halt}$ and $c_1, c_2 \in \mathbb{N}$.

The *halting problem* refers to determining whether the computation of a two counter machine $\mathcal{M}$ with given initial values $c_1, c_2 \in \mathbb{N}$ is finite. Both the halting problem and its dual, the *non-halting problem*, are undecidable.

▶ **Theorem 14.** *Given a valid signature Σ, two natural number variables m and n, and two types A and B such that $m, n\ ;\ \top \vdash_\Sigma A, B$ valid. Then it is undecidable whether for concrete $i, j \in \mathbb{N}$ we have $A[i/m, j/n] \equiv B[i/m, j/n]$.*

Proof. Given a two counter machine, we construct a signature Σ and two types A and B with free arithmetic variables m and n such that the computation of the machine starting with initial counter values i and j is infinite iff $A[i/m, j/n] \equiv B[i/m, j/n]$ in Σ.

We define types $T_{\text{inf}} = \oplus\{\ell : T_{\text{inf}}\}$ and $T'_{\text{inf}} = \oplus\{\ell' : T'_{\text{inf}}\}$ for *distinct* labels ℓ and ℓ'. Next, for every instruction ι_i, we define types T_i and T'_i based on the form of the instruction.

- Case ($\iota_i = \mathsf{inc}(c_1); \mathsf{goto}\ k$): We define

$$\begin{array}{lcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_1 : T_k[c_1 + 1, c_2]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_1 : T'_k[c_1 + 1, c_2]\} \end{array}$$

- Case ($\iota_i = \mathsf{inc}(c_2); \mathsf{goto}\ k$): We define

$$\begin{array}{lcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_2 : T_k[c_1, c_2 + 1]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_2 : T'_k[c_1, c_2 + 1]\} \end{array}$$

- Case ($\iota_i = \mathsf{zero}(c_1)?\ \mathsf{goto}\ k : \mathsf{dec}(c_1); \mathsf{goto}\ l$): We define

$$\begin{array}{lcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_1 : ?\{c_1 = 0\}.\, T_k[c_1, c_2], \mathsf{dec}_1 : ?\{c_1 > 0\}.\, T_l[c_1 - 1, c_2]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_1 : ?\{c_1 = 0\}.\, T'_k[c_1, c_2], \mathsf{dec}_1 : ?\{c_1 > 0\}.\, T'_l[c_1 - 1, c_2]\} \end{array}$$

- Case ($\iota_i = \mathsf{zero}(c_2)?\ \mathsf{goto}\ k : \mathsf{dec}(c_2); \mathsf{goto}\ l$): We define

$$\begin{array}{lcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_2 : ?\{c_2 = 0\}.\, T_k[c_1, c_2], \mathsf{dec}_2 : ?\{c_2 > 0\}.\, T_l[c_1, c_2 - 1]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_2 : ?\{c_2 = 0\}.\, T'_k[c_1, c_2], \mathsf{dec}_2 : ?\{c_2 > 0\}.\, T'_l[c_1, c_2 - 1]\} \end{array}$$

- Case ($\iota_i = \mathsf{halt}$): We define

$$\begin{array}{lcl} T_i[c_1, c_2] & = & T_{\text{inf}} \\ T'_i[c_1, c_2] & = & T'_{\text{inf}} \end{array}$$

We set type $A = T_1[m, n]$ and $B = T'_1[m, n]$. Now suppose, the counter machine $\mathcal{M}$ is initialized in the state $(1, i, j)$. The type equality question we ask is whether $T_1[i, j] \equiv T'_1[i, j]$. The two types only differ at the halting instruction. If $\mathcal{M}$ does not halt, the two types capture exactly the same communication behavior, since the halting instruction is never reached and they agree on all other instructions. If $\mathcal{M}$ halts, the first type proceeds with label ℓ and the second with ℓ' and they are therefore not equal. Hence, the two types are equal iff $\mathcal{M}$ does not halt. ◀

We can easily modify this reduction for an isorecursive interpretation of types, by wrapping $\oplus\{\mathbf{unfold} : \_\_\}$ around the right-hand side of each type definition to simulate the unfold message. We also see that a host of other problems are undecidable, such as determining whether two types with free arithmetic variables are equal for all instances. This is the problem that arises while type-checking parametric process definitions.

5 A Practical Algorithm for Type Equality

Despite its undecidability, we have designed a coinductive algorithm for soundly approximating type equality. Similar to Gay and Hole's algorithm, it proceeds by attempting to construct a bisimulation. Due to the undecidability of the problem, our algorithm can terminate in three different states: (1) we have succeeded in constructing a bisimulation, (2) we have found a counterexample to type equality by finding a place where the types may exhibit different behavior, or (3) we have terminated the search without a definitive answer. From the point of view of type-checking, both (2) and (3) are interpreted as a failure to type-check (but there is a recourse; see Subsection 5.2). Our algorithm is expressed as a set of inference rules where the execution of the algorithm corresponds to the bottom-up construction of a deduction. The algorithm is deterministic (no backtracking) and the implementation is quite efficient in practice (see Section 6).

One of the basic operations in Gay and Hole's algorithm is *loop detection*, that is, we have to determine that we have already added an equation $A \equiv B$ to the bisimulation we are constructing. Since we must treat *open types*, that is, types with free arithmetic variables subject to some constraints, determining if we have considered an equation already becomes a difficult operation. To that purpose we make an initial pass over the given type and introduce fresh *internal names* abstracted over their free type variables and known constraints. In the resulting signature defined type variables and type constructors alternate and we can perform loop detection entirely on type definitions (whether internal or external).

▶ **Example 15** (Queues, v3)**.** After creating internal names $\%i$ for the type of queue we obtain the following signature (here parametric in A).

$$\begin{array}{ll} \mathsf{queue}_A[n] = \&\{\mathbf{ins} : \%0[n], \mathbf{del} : \%1[n]\} & \\ \%0[n] = A \multimap \mathsf{queue}_A[n+1] & \%3 = \mathbf{1} \\ \%1[n] = \oplus\{\mathbf{none} : \%2[n], \mathbf{some} : \%4[n]\} & \%4[n] = ?\{n > 0\}.\,\%5[n] \\ \%2[n] = ?\{n = 0\}.\,\%3 & \%5[n \mid n > 0] = A \otimes \mathsf{queue}_A[n-1] \end{array}$$

Based on the invariants established by internal names, the algorithm only needs to compare two type variables or two structural types. The rules are shown in Figure 1. The judgment has the form $\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A \equiv B$ where $\mathcal{V}$ contains the free arithmetic

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_\ell \equiv B_\ell \quad (\forall \ell \in L)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \oplus\{\ell : A_\ell\}_{\ell\in L} \equiv \oplus\{\ell : B_\ell\}_{\ell\in L}}\ \oplus \qquad \frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_\ell \equiv B_\ell \quad (\forall \ell \in L)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \&\{\ell : A_\ell\}_{\ell\in L} \equiv \&\{\ell : B_\ell\}_{\ell\in L}}\ \&$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_1 \equiv B_1 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_2 \equiv B_2}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_1 \otimes A_2 \equiv B_1 \otimes B_2}\ \otimes$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_1 \equiv B_1 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_2 \equiv B_2}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A_1 \multimap A_2 \equiv B_1 \multimap B_2}\ \multimap \qquad \frac{}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \mathbf{1} \equiv \mathbf{1}}\ \mathbf{1}$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \leftrightarrow \psi \qquad \mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Gamma \vdash A \equiv B}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash ?\{\phi\}.\, A \equiv ?\{\psi\}.\, B}\ ? \qquad \frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \leftrightarrow \psi \qquad \mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Gamma \vdash A \equiv B}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash !\{\phi\}.\, A \equiv !\{\psi\}.\, B}\ !$$

$$\frac{\mathcal{V}, k\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A[k/m] \equiv B[k/n]}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \exists m.\, A \equiv \exists n.\, B}\ \exists^k \qquad \frac{\mathcal{V}, k\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A[k/m] \equiv B[k/n]}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash \forall m.\, A \equiv \forall n.\, B}\ \forall^k$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \bot}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash A \equiv B}\ \bot \qquad \frac{\mathcal{V}\ ;\ \mathcal{C} \vDash e_1 = e_1' \wedge \ldots \wedge e_n = e_n'}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash V[\overline{e}] \equiv V[\overline{e}']}\ \mathsf{refl}$$

$$\frac{\begin{array}{c} V_1[\overline{v_1} \mid \phi_1] = A \in \Sigma \qquad V_2[\overline{v_2} \mid \phi_2] = B \in \Sigma \\ \gamma = \langle \mathcal{V}\ ;\ \mathcal{C}\ ;\ V_1[\overline{e_1}] \equiv V_2[\overline{e_2}] \rangle \\ \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma, \gamma \vdash A[\overline{e_1}/\overline{v_1}] \equiv B[\overline{e_2}/\overline{v_2}] \end{array}}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]}\ \mathsf{expd}$$

$$\frac{\langle \mathcal{V}'\ ;\ \mathcal{C}'\ ;\ V_1[\overline{e_1}'] \equiv V_2[\overline{e_2}'] \rangle \in \Gamma \qquad \mathcal{V}\ ;\ \mathcal{C} \vDash \exists \mathcal{V}'.\, \mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \vdash V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]}\ \mathsf{def}$$

Figure 1 Algorithmic Rules for Type Equality.

variables in the constraints $\mathcal{C}$ and the types A and B, and Γ is a collection of *closures* $\langle \mathcal{V}'\ ;\ \mathcal{C}'\ ;\ V_1'[\overline{e_1}'] \equiv V_2'[\overline{e_2}'] \rangle$. If a derivation can be constructed, all ground instances of all closures are included in the resulting bisimulation (see the proof of Theorem 20). A ground instance $V_1'[\overline{e_1}'[\sigma']] \equiv V_2'[\overline{e_2}'[\sigma']]$ is given by a substitution σ' over variables in $\mathcal{V}'$ such that $\vDash \mathcal{C}'[\sigma']$.

The rules for type constructors simply compare the components. If the type constructors (or the label sets in the $\oplus$ and & rules) do not match, then type equality fails (having constructed a counterexample to bisimulation) unless the $\bot$ rule applies. This rules handles the case where the constraints are contradictory and no communication is possible.

The rule of reflexivity is needed explicitly here (but not in the version of Gay and Hole) because due to the incompleteness of the algorithm we may otherwise fail to recognize type variables with equal index expressions as equal.

Now we come to the key rules, expd and def. In the expd rule we expand the definitions of $V_1[\overline{e_1}]$ and $V_2[\overline{e_2}]$, and we also add the closure $\langle \mathcal{V}\ ;\ \mathcal{C}\ ;\ V_1[\overline{e_1}] \equiv V_2[\overline{e_2}] \rangle$ to Γ. Since the equality of $V_1[\overline{e_1}]$ and $V_2[\overline{e_2}]$ must hold for all its ground instances, the extension of Γ with the corresponding closure remembers exactly that. We can ignore the propositions ϕ_1 and ϕ_2 since the validity of types (rule tdef in Section 3) ensures that both $\vDash \phi_1[\overline{e_1}/\overline{v_1}]$ and $\vDash \phi_2[\overline{e_2}/\overline{v_2}]$ hold.

In the def rule we close off the derivation successfully if all instances of the equation $V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]$ are already instances of a closure in Γ. This is checked by the entailment in the second premise, $\mathcal{V}\ ;\ \mathcal{C} \vDash \exists \mathcal{V}'.\, \mathcal{C}' \wedge \overline{E_1} = \overline{e_1} \wedge \overline{E_2} = \overline{e_2}$. This entailment is verified as a

closed $\forall\exists$ arithmetic formula, even if the original constraints $\mathcal{C}$ and $\mathcal{C}'$ do not contain any quantifiers. While for Presburger arithmetic we can decide such a proposition using quantifier elimination, other constraint domains may not permit such a decision procedure.

The algorithm so far is sound, but potentially nonterminating because when encountering variable/variable equations, we can use the expd rule indefinitely. To ensure termination, we restrict the expd rule to the case where *no* closure with the same type variables V_1 and V_2 is already present in Γ. This also removes the overlap between these two rules. Note that if type variables have no parameters, our algorithm specializes to Gay and Hole's (with the small optimizations of reflexivity and internal naming), which means our algorithm is sound and complete on unindexed types.

As an extension, our algorithm also allows the programmer to specify a *depth bound* k. This informs the algorithm to apply the expd rule until there are at most k closures with the same type variables V_1 and V_2 in Γ.

▶ **Example 16** (Integer Counter)**.** An integer counter with increment (**inc**), decrement (**dec**) and sign-test (**sgn**) operations provides type $\mathsf{intctr}[x, y]$, where the current value of the counter is $x - y$ for natural numbers x and y.

$$\begin{aligned}\mathsf{intctr}[x,y] = \&\{&\mathbf{inc} : \mathsf{intctr}[x+1,y],\\ &\mathbf{dec} : \mathsf{intctr}[x,y+1],\\ &\mathbf{sgn} : \oplus\{\mathbf{neg} : ?\{x<y\}.\,\mathsf{intctr}[x,y],\\ &\qquad\mathbf{zer} : ?\{x=y\}.\,\mathsf{intctr}[x,y],\\ &\qquad\mathbf{pos} : ?\{x>y\}.\,\mathsf{intctr}[x,y]\}\}\end{aligned}$$

Under this definition our algorithm verifies, for example, that an increment followed by a decrement does not change the counter value. That is,

$$x, y\ ;\ \top\ ;\ \cdot \vdash \mathsf{intctr}[x,y] \equiv \mathsf{intctr}[x+1, y+1]$$

where we have elided the assumptions $x, y \geq 0$. When applying expd, we assume $\gamma = \langle x', y'\ ;\ \top\ ;\ \mathsf{intctr}[x', y'] \equiv \mathsf{intctr}[x'+1, y'+1]\rangle$. Then, for example, in the first branch (for inc) we conclude $x, y\ ;\ \top\ ;\ \gamma \vdash \mathsf{intctr}[x+1, y] \equiv \mathsf{intcr}[x+2, y+1]$ using the def rule and the entailment $x, y\ ;\ \top \vDash \exists x'.\,\exists y'.\,x' = x+1 \wedge y' = y \wedge x'+1 = x+2 \wedge y'+1 = y+1$. The other branches are similar.

As exemplified by the above example, a distinguishing feature of our algorithm is that it goes beyond *reflexivity*. Essentially, $V[\overline{e_1}] \equiv V[\overline{e_2}]$ can hold even if $\overline{e_1} \neq \overline{e_2}$. This is in contrast with traditional refinement languages such as DML [46] that use reflexivity as the only criterion for equality on indexed type names.

5.1 Soundness of the Type Equality Algorithm

We prove that the type equality algorithm is sound with respect to the definition of type equality. The soundness is proved by constructing a type bisimulation from a derivation of the algorithmic type equality judgment. We sketch the key points of the proofs.

The first gap we have to bridge is that the type bisimulation is defined only for closed types, because observations can only arise from communication along channels which, at runtime, will be of closed type. So, if we can derive $\mathcal{V}\ ;\ \mathcal{C}\ ;\ \cdot \vdash A \equiv B$ then we should interpret this as stating that for all ground substitutions σ over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$ we have $A[\sigma] \equiv B[\sigma]$.

▶ **Definition 17.** *Given a relation $\mathcal{R}$ on valid ground types and two types A and B such that $\mathcal{V}$; $\mathcal{C} \vdash A, B$ valid, we write $\forall\mathcal{V}.\mathcal{C} \Rightarrow A \equiv_{\mathcal{R}} B$ if for all ground substitutions σ over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$ we have $(A[\sigma], B[\sigma]) \in \mathcal{R}$.*

Furthermore, we write $\forall\mathcal{V}.\mathcal{C} \Rightarrow A \equiv B$ if there exists a type bisimulation $\mathcal{R}$ such that $\forall\mathcal{V}.\mathcal{C} \Rightarrow A \equiv_{\mathcal{R}} B$.

Note that if $\mathcal{V}$; $\mathcal{C} \vDash \bot$, then $\forall\mathcal{V}.\mathcal{C} \Rightarrow A \equiv B$ is vacuously true, since there does not exist a ground substitution σ such that $\vDash \mathcal{C}[\sigma]$.

A key lemma is the following, which is needed to show the soundness of the def rule.

▶ **Lemma 18.** *Suppose $\forall\mathcal{V}'.\mathcal{C}' \Rightarrow V_1[\overline{e_1}'] \equiv_{\mathcal{R}} V_2[\overline{e_2}']$ holds. Further assume that $\mathcal{V}$; $\mathcal{C} \vDash \exists\mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}$ for some $\mathcal{V}, \mathcal{C}, \overline{e_1}, \overline{e_2}$. Then, $\forall\mathcal{V}.\mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv_{\mathcal{R}} V_2[\overline{e_2}]$ holds.*

Proof. To prove $\forall\mathcal{V}.\mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv_{\mathcal{R}} V_2[\overline{e_2}]$, it is sufficient to show that $V_1[\overline{e_1}[\sigma]] \equiv_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$ for any substitution σ over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$. Applying this substitution to $\mathcal{V}$; $\mathcal{C} \vDash \exists\mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}$, we infer $\exists\mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1}[\sigma] \wedge \overline{e_2}' = \overline{e_2}[\sigma]$ since $\vDash \mathcal{C}[\sigma]$. Thus, there exists σ' over $\mathcal{V}'$ such that $\vDash \mathcal{C}'[\sigma']$ holds, and $\overline{e_1}'[\sigma'] = \overline{e_1}[\sigma]$ and $\overline{e_2}'[\sigma'] = \overline{e_2}[\sigma]$. And since $\forall\mathcal{V}'.\mathcal{C}' \Rightarrow V_1[\overline{e_1}'] \equiv_{\mathcal{R}} V_2[\overline{e_2}']$, we deduce that for any ground substitution (including the current one) σ' over $\mathcal{V}'$, $V_1[\overline{e_1}'[\sigma']] \equiv_{\mathcal{R}} V_2[\overline{e_2}'[\sigma']]$ holds. This implies that $V_1[\overline{e_1}[\sigma]] \equiv_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$ since $\overline{e_1}'[\sigma'] = \overline{e_1}[\sigma]$ and $\overline{e_2}'[\sigma'] = \overline{e_2}[\sigma]$. ◀

We construct the bisimulation from a derivation of $\mathcal{V}$; $\mathcal{C}$; $\Gamma \vdash A \equiv B$ by (i) collecting the conclusions of all the sequents, excepting only the def rule, and (ii) forming all ground instances from them.

▶ **Definition 19.** *Given a derivation $\mathcal{D}$ of $\mathcal{V}$; $\mathcal{C}$; $\Gamma \vdash A \equiv B$, we define the set $\mathcal{S}(\mathcal{D})$ of closures. For each sequent $\mathcal{V}'$; $\mathcal{C}'$; $\Gamma' \vdash A' \equiv B'$ (except the conclusion of the* def *rule) we include the closure $\langle \mathcal{V}' ; \mathcal{C}' ; A' \equiv B' \rangle$ in $\mathcal{S}(\mathcal{D})$.*

▶ **Theorem 20** (Soundness). *If $\mathcal{V}$; $\mathcal{C}$; $\cdot \vdash A \equiv B$, then $\forall\mathcal{V}.\mathcal{C} \Rightarrow A \equiv B$.*

Proof. We are given a derivation $\mathcal{D}_0$ of $\mathcal{V}_0$; $\mathcal{C}_0$; $\cdot \vdash A_0 \equiv B_0$. Construct $\mathcal{S}(\mathcal{D}_0)$ and define a relation $\mathcal{R}$ on closed valid types as follows:

$$\mathcal{R} = \{(A[\sigma], B[\sigma]) \mid \langle \mathcal{V} ; \mathcal{C} ; A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0) \text{ and } \sigma \text{ over } \mathcal{V} \text{ with } \vDash \mathcal{C}[\sigma]\}$$

We prove that $\mathcal{R}$ is a type bisimulation. Then our theorem follows since the closure $\langle \mathcal{V}_0 ; \mathcal{C}_0 ; A_0 \equiv B_0 \rangle \in \mathcal{S}(\mathcal{D}_0)$.

Consider $(A[\sigma], B[\sigma]) \in \mathcal{R}$ where $\langle \mathcal{V} ; \mathcal{C} ; A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0)$ for some σ over $\mathcal{V}$ and $\vDash \mathcal{C}[\sigma]$.

First, consider the case where $\mathcal{V}$; $\mathcal{C} \vDash \bot$. Under such a constraint, $\mathcal{V}$; $\mathcal{C}$; $\cdot \vdash A \equiv B$ holds true due to the $\bot$ rule. Furthermore, $\forall\mathcal{V}.\mathcal{C} \Rightarrow A \equiv B$ holds vacuously, and the algorithm is sound. For the remaining cases, we case analyze on the structure of $A[\sigma]$ and assume that there exists a ground substitution σ such that $\vDash \mathcal{C}[\sigma]$.

Consider the case where $A = \oplus\{\ell : A_\ell\}_{\ell \in L}$. Since A and B are both structural, $B = \oplus\{\ell : B_\ell\}_{\ell \in L}$. Since $\langle \mathcal{V} ; \mathcal{C} ; A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0)$, by definition of $\mathcal{S}(\mathcal{D}_0)$, we get $\langle \mathcal{V} ; \mathcal{C} ; A_\ell \equiv B_\ell \rangle \in \mathcal{S}(\mathcal{D}_0)$ for all $\ell \in L$. By the definition of $\mathcal{R}$, we get that $(A_\ell[\sigma], B_\ell[\sigma]) \in \mathcal{R}$. Also, $A[\sigma] = \oplus\{\ell : A_\ell[\sigma]\}_{\ell \in L}$ and similarly, $B[\sigma] = \oplus\{\ell : B_\ell[\sigma]\}_{\ell \in L}$. Hence, $\mathcal{R}$ satisfies the appropriate closure condition for a type bisimulation.

Next, consider the case where $A = ?\{\phi\}.A'$. Since A and B are both structural, $B = ?\{\psi\}.B'$. Since $\langle \mathcal{V} ; \mathcal{C} ; A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0)$, we obtain $\mathcal{V}$; $\mathcal{C} \vDash \phi \leftrightarrow \psi$ and $\langle \mathcal{V} ; \mathcal{C} \wedge \phi ; A' \equiv B' \rangle \in \mathcal{S}(\mathcal{D}_0)$. Thus, for any substitution σ such that $\vDash \mathcal{C}[\sigma] \wedge \phi[\sigma]$, we get that $(A'[\sigma], B'[\sigma]) \in \mathcal{R}$ with $A[\sigma] = ?\{\phi[\sigma]\}.A'[\sigma]$ and $B[\sigma] = ?\{\psi[\sigma]\}.B'[\sigma]$. Since $\vDash \phi[\sigma]$ and and $\mathcal{V}$; $\mathcal{C} \vDash \phi \leftrightarrow \psi$ we also obtain $\vDash \psi[\sigma]$ and the closure condition is satisfied.

Next, consider the case where $A = \exists m.\, A'$. Since A and B are both structural, $B = \exists n.\, B'$. Since $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0)$, we get that $\langle \mathcal{V}, k \,;\, \mathcal{C} \,;\, A'[k/m] \equiv B'[k/n] \rangle \in \mathcal{S}(\mathcal{D}_0)$. Since k was chosen fresh and does not occur in $\mathcal{C}$, we obtain that for any $i \in \mathbb{N}$ we have $\vDash \mathcal{C}[\sigma, i/k]$ and therefore $(A'[\sigma, i/k], B'[\sigma, i/k]) \in \mathcal{R}$ for all $i \in \mathbb{N}$ and the closure condition is satisfied.

The only case where a conclusion is not added to $\mathcal{S}(\mathcal{D}_0)$ is the def rule. In this case, adding $(\forall \mathcal{V}.\, \mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv V_2[\overline{e_2}])$ is redundant: Lemma 18 states that $V_1[\overline{e_1}[\sigma]] \equiv_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$ which implies $(V_1[\overline{e_1}[\sigma]], V_2[\overline{e_2}[\sigma]]) \in \mathcal{R}$. ◀

5.2 Type Equality Declarations

Even though the type equality algorithm in Section 5 is incomplete, we have yet to find a natural example where it fails after we added reflexivity as a general rule. But since we cannot see a simple reason why this should be so, we made our type equality algorithm extensible by the programmer via an additional form of declaration

$$\forall \mathcal{V}.\, \mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]$$

in signatures. Let Γ_Σ denote the set of all such declarations. Then we check

$$\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma_\Sigma \vdash V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]$$

for each such declaration, seeding the construction of a bisimulation with all the given equations. Then, when type-checking has to decide the equality of two types, it starts not with the empty context Γ but with Γ_Σ. Our soundness proof can easily accommodate this more general algorithm.

▶ **Example 21** (Queues, v4)**.** Consider the two types $\mathsf{queue}_A[n]$ and $\mathsf{queue}'_A[n]$, both representing queue data structures, but $\mathsf{queue}'_A[n]$ is rooted at 1.

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}_A[n+1],\\ &\mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 0\}.\, \mathbf{1},\\ &\qquad\mathbf{some} : ?\{n > 0\}.\, A \otimes \mathsf{queue}_A[n-1]\}\}\end{aligned}$$

$$\begin{aligned}\mathsf{queue}'_A[n] = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}'_A[n+1],\\ &\mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 1\}.\, \mathbf{1},\\ &\qquad\mathbf{some} : ?\{n > 1\}.\, A \otimes \mathsf{queue}'_A[n-1]\}\}\end{aligned}$$

Our intuition would suggest that $\mathsf{queue}_A[0] \equiv \mathsf{queue}'_A[1]$. But this cannot be directly proved by our equality algorithm. While checking this equality, our algorithm would add $\langle \cdot \,;\, \top \,;\, \mathsf{queue}_A[0] \equiv \mathsf{queue}'_A[1] \rangle$ to Γ and would continue to check $\mathsf{queue}_A[1] \equiv \mathsf{queue}'_A[2]$ (the **ins** branch). However, our closure in Γ is not sufficient to prove this goal (the def rule fails), and our algorithm reports the types may not be equal. However, we can add a general equality declaration $\forall n.\, \mathsf{queue}_A[n] \equiv \mathsf{queue}'_A[n+1]$ to the signature. This can be verified by our algorithm since it would add $\langle n \,;\, \top \,;\, \mathsf{queue}_A[n] \equiv \mathsf{queue}'_A[n+1] \rangle$ to Γ and use it to prove $\mathsf{queue}_A[n+1] \equiv \mathsf{queue}'_A[n+2]$ in the **ins** branch. Then, we will use the same equality declaration from the signature to verify $\mathsf{queue}_A[0] \equiv \mathsf{queue}'_A[1]$ by instantiating $n = 0$.

6 Implementation and Further Examples

We have implemented the algorithm presented in Section 5 as part of the Rast programming language [14], whose name derives from "Resource-Aware Session Types". Rast is based on intuitionistic linear sessions [8, 9] extended with general equirecursive types and recursively

Table 1 Case Studies.

Module	LOC	#Defs	T (ms)
arithmetic	143	8	1.325
integers	114	8	1.074
linlam	67	6	4.003
list	441	29	3.419
primes	118	8	1.646
segments	65	9	0.195
ternary	235	16	1.967
theorems	141	16	0.894
tries	308	9	5.283
Total	**1632**	**109**	**19.806**

defined processes. We do not explicitly dualize types [43] but distinguish providers and clients that are connected by a private channel. In parallel work we have proved type safety for Rast, which includes type preservation (session fidelity) and global progress (deadlock freedom). The open-source implementation is written in Standard ML and currently comprises about 7500 lines of source code [36].

Rast supports indexed types, quantifiers, and arithmetic constraints, following the presentation in this paper with minor syntactic differences. In addition, Rast has temporal [12] and ergometric [13] types that capture parallel and sequential complexity of programs. These bounds often depend on intrinsic properties of the data structures (such as the length of a queue or the value of a binary number) which are expressed as arithmetic indices.

Rast's linear type checker is bidirectional, which means that only process definitions need to be annotated with their types. In the so-called *explicit syntax* type checking is then straightforward, breaking down the structure of the type and unfolding definitions, except for calls to type equality (which are necessary for forwarding, process invocations, and sending of channels). The implementation also supports an *implicit syntax* in which some parts of the program, specifically those concerning missing branches that can be proved to be impossible using refinements, can be omitted from the source and are reconstructed. The reconstructed code is then passed through the type checker as ultimate arbiter.

We use a straightforward implementation of Cooper's algorithm [10] to decide Presburger arithmetic with two small but significant optimizations. One takes advantage of the fact that we are working over natural numbers rather than integers, the other is to eliminate constraints of the form $x = e$ by substituting e for x in order to reduce the number of variables. We also extend our solver to handle non-linear constraints. Since non-linear arithmetic is undecidable, in general, we use a normalizer which collects coefficients of each term in the multinomial expression. To check $e_1 = e_2$, we normalize $e_1 - e_2$ and check that each coefficient of the normal form is 0. To check $e_1 \geq e_2$, we normalize $e_1 - e_2$ and check that each coefficient is non-negative.

We have a variety of 21 examples implemented, totaling about 3700 lines of code, for which complete code can be found in our open source repository [36]. Table 1 describes the results for nine representative case studies: LOC describes the lines of code, #Defs shows the number of process definitions, and T (ms) shows the type-checking time in milliseconds respectively. The experiments were run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory. We briefly describe each case study.

1. **arithmetic**: natural numbers in unary and binary representation indexed by their value and processes implementing standard arithmetic operations.
2. **integers**: an integer counter represented using two indices x and y with value $x - y$.
3. **linlam**: expressions in the linear λ-calculus indexed by their size with an *eval* process to evaluate them (see below for an excerpt).
4. **list**: lists indexed by their size with standard operations (e.g., *append, reverse, map*).
5. **primes**: implementation of the sieve of Eratosthenes.
6. **segments**: type $\mathsf{seg}[n] = \forall k.\mathsf{list}[k] \multimap \mathsf{list}[n+k]$ representing partial lists with constant-work append operation.
7. **ternary**: natural numbers represented in balanced ternary form with digits $0, 1, -1$, indexed by their value, and some standard operations on them.
8. **theorems**: processes representing (circular [15]) proofs of simple arithmetic theorems.
9. **tries**: a trie data structure to store multisets of binary numbers, with constant amortized work insertion and deletion, verified with ergometric types.

Linear λ-calculus. We briefly sketch the types in an implementation of the (untyped) linear λ-calculus in which the index objects track the size of the expression, because it uses multiple feature of the type system.

$$\begin{aligned}\mathsf{exp}[n] = \oplus\{&\mathbf{lam} : ?\{n > 0\}.\, \forall n_1.\, \mathsf{exp}[n_1] \multimap \mathsf{exp}[n_1 + n - 1],\\ &\mathbf{app} : \exists n_1.\, \exists n_2.\, ?\{n = n_1 + n_2 + 1\}.\, \mathsf{exp}[n_1] \otimes \mathsf{exp}[n_2]\}\end{aligned}$$

An expression is either a λ-abstraction (sending label **lam**) or an application (sending label **app**). In case of **lam**, the continuation receives a number n_1 and an argument of size n_1 and then behaves like the body of the λ-abstraction of size $n_1 + n - 1$. In case of **app**, it will send n_1 and n_2 such that $n = n_1 + n_2 + 1$ followed an expression of size n_1 and then behave as an expression of size n_2.

A value can only be a λ-abstraction

$$\mathsf{val}[n] = \oplus\{\mathbf{lam} : ?\{n > 0\}.\, \forall n_1.\, \mathsf{exp}[n_1] \multimap \mathsf{exp}[n_1 + n - 1]\}$$

so the **app** label is not permitted. Type checking verifies that that the result of evaluating a linear λ-term is no larger than the original term. The declaration below expresses that $\mathit{eval}\,[n]$ is client to a process sending a λ-expression of size n along channel e and provides a value of size k, where $k \leq n$.

$$(e : \mathsf{exp}[n]) \vdash \mathit{eval}\,[n] :: (v : \exists k.\, ?\{k \leq n\}.\, \mathsf{val}[k])$$

7 Further Related Work

Traditional languages with dependent type refinements such as Zenger's [47] or DML [46] only use the rule of reflexivity as a criterion for equality of indexed types. This is justified in the context of these functional languages because data types are generative and therefore *nominal* in nature. This is also true for more recent languages with linearity and value-dependent types such as Granule [32].

Session type systems that allow dependencies are label-dependent session types [39] and richer linear type theories [40, 33, 41]. Toninho et al. [40, 33] allow sufficient dependencies that, in general, proofs must be sent in some circumstances. They do not provide a type equality algorithm or implementation. In a more recent paper, Toninho et al. [41] propose a dependent type theory with rich notions of value and process equality based on $\beta\eta$-congruences

and certain process equalities, but they do not discuss decidability or algorithms for type checking or type equality. Wu and Xi [44] propose a dependent session type system based on ATS [45] formalizing type equality in terms of subtyping and regular constraint relations. They mention recursive session types as a possible extension, but do not develop them nor investigate properties of the required type equality.

Linearly refined session types [2, 16] extend the π-calculus with capabilities from a fragment of multiplicative linear logic. These capabilities encode an authorization logic enabling fine-grained specifications and are thus not directly comparable to arithmetic refinements. Session types with limited arithmetic refinements (only base types could be refined) have been proposed for the purpose of runtime monitoring [20, 19], which is complementary to our uses for static verification. They have also been proposed to capture work [13, 11] and parallel time [12], but parameterization over index objects was left to an informal meta-level and not part of the object language. Consequently, these languages contain neither constraints nor quantifiers, and the metatheory of type equality, type checking, and reconstruction in the presence of index variables was not developed.

Several other generalizations of session types for specification and verification have been proposed. Generalizing the idea of "Design by Contract" [29] to distributed domains, session types have been elaborated with logical predicates to obtain *global assertions* [4]. Actris [23] combines concurrent separation logics with session types for reasoning about message passing in the presence of other concurrency paradigms. Actris is able to prove functional correctness of a distributed merge sort, a distributed load-balancing mapper, and a variant of the map-reduce model. Context-free session types [38] are another generalization of basic session types in a different direction, essentially allowing the concatenation of sessions. This generalization has decidable type checking and type equality problems that have been shown to be efficient in practice [1].

Asynchronous session types [18] have a notion of subtyping under different assumptions regarding communication behavior [31]. The resulting subtyping relation also turns out to be undecidable [6, 27] with the development of recent practical incomplete algorithms [5]. The expressive power of asynchronous session subtyping seems incomparable to our arithmetically refined session types.

8 Conclusion

This paper explored the metatheory of session types with arithmetic refinements, showing the undecidability of type equality. Nevertheless, we have shown a sound, but incomplete algorithm that has performed well over a range of examples in our Rast implementation.

Natural extensions include nonlinear arithmetic and other constraint domains, balancing practicality of type checking with expressive power. We would also like to generalize from type equality to subtyping, replacing the notion of bisimulation with a simulation. Clearly, this will be undecidable as well, but the pioneering work by Gay and Hole and the characteristics of our algorithms suggest that it should extend cleanly and remain practical.

Finally, we would also like to generalize our approach to a mixed linear/nonlinear language [3] or all the way to adjoint session types [34, 35]. Since the main issues of type equality are orthogonal to the presence or absence of structural properties, we conjecture that the algorithm proposed here will extend to this more general setting.

References

1 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Deciding the bisimilarity of context-free session types. In A. Biere and D. Parker, editors, *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, pages 39–56, Dublin, Ireland, April 2020. Springer LNCS 12079.

2 Pedro Baltazar, Dimitris Mostrous, and Vasco T. Vasconcelos. Linearly refined session types. In S. Alves and I. Mackie, editors, *International Workshop on Linearity (LINEARITY 2012)*, pages 38–49, Tallinn, Estonia, April 2012. EPTCS 101.

3 Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In L. Pacholski and J. Tiuryn, editors, *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL 1994)*, pages 121–135, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

4 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 162–176, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping. In W. Fokkink and R. van Glabbeek, editors, *30th International Conference on Concurrency Theory (CONCUR 2019)*, pages 38:1–38:16, Amsterdam, The Netherlands, August 2019. LIPIcs 140.

6 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Information & Computation*, 256:300–320, 2017.

7 Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 330–349, Rome, Italy, March 2013. Springer LNCS 7792.

8 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In P.Gastin and F.Laroussinie, editors, *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

9 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types.

10 David C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

11 Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts, 2019. `arXiv:1902.06056`.

12 Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. In M. Flatt, editor, *Proceedings of International Conference on Functional Programming (ICFP 2018)*, pages 91:1–91:30, St. Louis, Missouri, USA, September 2018. ACM.

13 Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In A. Dawar and E. Grädel, editors, *Proceedings of 33rd Symposium on Logic in Computer Science (LICS'18)*, pages 305–314, Oxford, UK, July 2018.

14 Ankush Das and Frank Pfenning. Rast: Resource-aware session types with arithmetic refinements. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 4:1–4:17. LIPIcs 167, June 2020. System description. To appear.

15 Farzaneh Derakhshan and Frank Pfenning. Circular proofs as session-typed processes: A local validity condition, August 2019. `arXiv:1908.01909`.

16 Juliana Franco and Vasco T. Vasconcelos. A concurrent programming language with refined session types. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods (SEFM 2013)*, pages 15–28, Madrid, Spain, September 2013. Springer LNCS 8368.

17 Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

18 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

19 Hannah Gommerstadt. *Session-Typed Concurrent Contracts*. PhD thesis, Carnegie Mellon University, September 2019. Available as Technical Report CMU-CS-19-119.

20 Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. In A. Ahmed, editor, *European Symposium on Programming (ESOP'18)*, pages 771–798, Thessaloniki, Greece, April 2018. Springer LNCS 10801.

21 Dennis Griffith. *Polarized Substructural Session Types*. PhD thesis, University of Illinois at Urbana-Champaign, April 2016.

22 Dennis Griffith and Elsa L. Gunter. LiquidPi: Inferrable dependent session types. In *Proceedings of the NASA Formal Methods Symposium*, pages 186–197. Springer LNCS 7871, 2013.

23 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371074.

24 Kohei Honda. Types for dyadic interaction. In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR 1993)*, pages 509–523. Springer LNCS 715, 1993.

25 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, pages 122–138. Springer LNCS 1381, 1998.

26 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In G. Necula and P. Wadler, editors, *Proceedings of the 35th Symposium on Principles of Programming Language (POPL 2008)*, pages 273–284, San Francisco, California, USA, January 2008. ACM.

27 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2017)*, pages 441–457. Springer LNCS 10203, 2017.

28 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st International Conference on Functional Programming*, pages 434–447, Nara, Japan, September 2016. ACM.

29 Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992. doi:10.1109/2.161279.

30 Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA, 1967.

31 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Information & Computation*, 241:227–263, 2015.

32 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades, III. Quantitative program reasoning with graded modal types. In *International Conference on Functional Programming (ICFP 2019)*, pages 110:1–110:30, Berlin, Germany, August 2019. ACM.

33 Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-carrying code in a session-typed process calculus. In *1st International Conference on Certified Programs and Proofs (CPP 2011)*, pages 21–36, Kenting, Taiwan, December 2011. Springer LNCS 7086.

34 Frank Pfenning and Dennis Griffith. Polarized substructural session types. In A. Pitts, editor, *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, pages 3–22, London, England, April 2015. Springer LNCS 9034. Invited talk.

35 Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. In F. Martins and D. Orchard, editors, *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES 2019)*, pages 60–79, Prague, Czech Republic, April 2019. EPTCS 291.

36 Rast language, February 2020. Version 1.01. URL: `https://bitbucket.org/fpfenning/rast/src/master/rast/`.

37 Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In R. Gupta and S. Amarasinghe, editors, *Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 159–169, Tuscon, Arizona, June 2008. ACM.

38 Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *Proceedings of the 21st International Conference on Functional Programming (ICFP 2016)*, pages 462–475, Nara, Japan, September 2016. ACM.

39 Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. In L. Birkedal, editor, *Proceedings of the Symposium on Programming Languages (POPL 2020)*, pages 67:1–67:29, New Orleans, Louisiana, USA, January 2020. ACM Proceedings on Programming Languages 4.

40 Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In P.Schneider-Kamp and M.Hanus, editors, *Proceedings of the 13th International Conference on Principles and Practice of Declarative Programming (PPDP 2011)*, pages 161–172, Odense, Denmark, July 2011. ACM.

41 Bernardo Toninho and Nobuko Yoshida. Depending on session-types processes. In C. Baier and U. Dal Lago, editors, *21st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2018)*, pages 128–145, Thessaloniki, Greece, April 2018. Springer LNCS 10803.

42 Vasco T. Vasconcelos. Fundamentals of session types. *Information & Computation*, 217:52–70, 2012.

43 Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012)*, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.

44 Hanwen Wu and Hongwei Xi. Dependent session types, 2017. `arXiv:1704.07004`.

45 Hongwei Xi. Applied type system: Extended abstract. In S. Berardi, M. Coppo, and F. Damiani, editors, *International Workshop on Types for Proofs and Programming (TYPES 2003)*, pages 394–408, Torino, Italy, April 2003. Springer LNCS 3085.

46 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 1999)*, pages 214–227, San Antonio, Texas, USA, January 1999. ACM Press.

47 Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

48 Fangyi Zhou. Refinement session types. Master's thesis, Imperial College London, 2019.

49 Fangyi Zhou, Francisco Ferreira, Rumyana Neykova, and Nobuko Yoshida. Fluid Types: Statically Verified Distributed Protocols with Refinements. In *11th Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*, 2019.