Session Types with Arithmetic Refinements

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

Frank Pfenning
Carnegie Mellon University, 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.2020.16

Funding This material is based upon work supported 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 (∨) 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.

## 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 \( \perp \) dual to \( \top \), and replace the \( \rightarrow \) operator with \( \exists \) with only minor changes to the remainder of the development.

\[
A, B, C ::= \begin{align*}
\top & \mid \perp \mid A \& B \mid A \oplus B \mid A \rightarrow B \\
& \mid \{ k : A \mid k \in L \} \mid \{ k : A \mid k \in L \} \\
& \mid A \otimes B \\
& \mid 1 \\
& \mid V \end{align*}
\]

- \( \top \): send label \( k \in L \) continue at type \( A_k \)
- \( \perp \): receive label \( k \in L \) continue at type \( A_k \)
- \( A \& B \): send channel \( a : A \) continue at type \( B \)
- \( A \oplus B \): receive channel \( a : A \) continue at type \( B \)
- \( A \rightarrow B \): send close message no continuation
- \( V \): 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 \( \oplus \{ \ell : A_\ell \} \in L \) sends a label \( k \in L \) and continues to provide \( A_k \). Dually, the provider of \( \& \{ \ell : A_\ell \} \in L \) 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 \( A \rightarrow B \) receives a channel of type \( A \) and provides \( B \). Finally, the provider of 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 \( \mu \) we postulate a signature \( \Sigma \) with definitions of type variables \( V \).

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

**Definition 2.** A relation \( \mathcal{R} \) on types is a type bisimulation if \((A, B) \in \mathcal{R}\) implies that for \( S = \text{unfold}_\Sigma \{ A \}, T = \text{unfold}_\Sigma \{ B \} \) we have
- If \( S = \oplus \{ \ell : A_\ell \} \in L \) then \( T = \oplus \{ \ell : B_\ell \} \in L \) and \((A_\ell, B_\ell) \in \mathcal{R} \) for all \( \ell \in L \).
- If \( S = \& \{ \ell : A_\ell \} \in L \) then \( T = \& \{ \ell : B_\ell \} \in L \) 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 \( S = A_1 \rightarrow A_2 \), then \( T = B_1 \rightarrow B_2 \) and \((A_1, B_1) \in \mathcal{R} \) and \((A_2, B_2) \in \mathcal{R} \).
- If \( S = 1 \) then \( T = 1 \).

**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 \( \rightarrow \)) 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 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 \( \text{queue}_A \) as a family of types, one for each \( A \), to avoid introducing explicit polymorphic type constructors.

\[
\text{queue}_A = \& \{ \text{ins} : A \rightarrow \text{queue}_A , \\
\text{del} : \oplus \{ \text{none} : 1 , \\
\text{some} : A \otimes \text{queue}_A \} \}
\]

**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.
bin = ⊕{b0 : bin, b1 : bin, e : 1}

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

- `e : close` representing 0
- `b0 : e ; close` also representing 0
- `b0 : b1 ; e ; close` representing 2
- `b1 : b0 ; b1 ; b1 ; 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[φ]` 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 `queue_A[n]` to stand for a queue with exactly `n` elements.

```
queue_A[n] = &{ins : A → queue_A[n + 1],
                   del : ⊕{none : ?{n = 0}. 1,
                          some : ?{n > 0}. A ⊗ queue_A[n − 1]})
```

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 `?{φ}. A` means that the provider must send a proof `p` of `φ`, so it corresponds to `∃p : φ. 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 `!{φ}. A` that licenses the assumption of `φ`, which, conceptually, corresponds to receiving a proof of `φ`.

**Example 7 (Binary Numbers, v2).** The indexed type `bin[n]` should represent a binary number with value `n`. Because the least significant bit comes first, we expect, for example, that `bin[n] = ⊕{b0 : ?{2 | n}. bin[n/2], ...} (a | 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.

```
bin[n] = ⊕{b0 : ∃k. ?{n = 2 * k}. bin[k],
            b1 : ∃k. ?{n = 2 * k + 1}. bin[k],
            e : ?{n = 0}. 1}
```

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 \( (i | e) \) means \( i \) divides \( e \).

**Types**

\[
A ::= \ldots \\
| \{\phi\}. A \quad \text{assert } \phi \quad \text{continue at type } A \\
| !\{\phi\}. A \quad \text{assume } \phi \quad \text{continue at type } A \\
| \exists n. A \quad \text{send number } i \quad \text{continue at type } A[i/n] \\
| \forall n. A \quad \text{receive number } i \quad \text{continue at type } A[i/n] \\
| V[\pi] \quad \text{variable instantiation}
\]

**Arith. Expressions**

\[
e ::= i \mid e + e \mid e - e \mid i \times e \mid (i | e) \mid n
\]

**Arith. Propositions**

\[
\phi ::= e = e \mid e > e \mid \top \mid \bot \mid \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid \exists n. \phi \mid \forall n. \phi
\]

**Signature**

\[
\Sigma ::= \cdots \mid \Sigma, V[\pi] | \phi = A
\]

An indexed type definition \( V[\pi | \phi] = A \) containing an optional proposition \( \phi \) requires every instantiation \( \tau \) (in \( V[\pi] \)) of the sequence of variables \( \pi \) to satisfy \( \phi[\pi/\tau] \). This is verified statically when a type signature is checked for validity, as defined below. We use \( V \) for a collection of arithmetic variables and \( C \) (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 (\( \vdash \Sigma' \) valid), and types (\( \vdash \Sigma \vdash A \) valid) where \( V \) is a collection of arithmetic variables including all free variables in constraint \( C \) and type \( A \). We silently rename variables so that \( n \) does not already occur in \( V \) in the \( \exists n \) and \( \forall V \) rules. We also call upon the semantic entailment judgment \( \vdash \Sigma \vdash C \models \phi \) which means that \( \forall V. C \models \phi \) holds in arithmetic and \( \vdash \phi \) abbreviates \( \vdash \phi : \top \models \phi \).

\[
\begin{align*}
\vdash \Sigma \text{ signature} & \quad \vdash \Sigma' \text{ valid} \\
\vdash \Sigma \vdash (\cdot) \text{ valid} & \quad \vdash \Sigma' \vdash \Sigma, V[\pi | \phi] = A \text{ valid}
\end{align*}
\]

\[
\begin{align*}
\vdash \Sigma \vdash C \models \phi & \quad \vdash \Sigma' \vdash C \models \phi \text{ valid} \\
\vdash \Sigma \vdash \{\phi\}. A \text{ valid} & \quad \vdash \Sigma' \vdash C \models !\{\phi\}. A \text{ valid}
\end{align*}
\]

\[
\begin{align*}
\vdash \Sigma \vdash V[n]. A \text{ valid} & \quad \vdash \Sigma \vdash \exists n. A \text{ valid} \\
\vdash \Sigma \vdash \forall n. A \text{ valid} & \quad \forall V^n
\end{align*}
\]

\[
\begin{align*}
V[\pi | \phi] & = A \in \Sigma \\
\vdash \Sigma \vdash C \models \phi[\pi/\tau] \text{ valid}
\end{align*}
\]

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 \( V[\pi] = A \) abbreviates \( V[\pi | \pi \geq 0] = A \). This means that in valid signatures every occurrence \( V[\pi] \) is such that \( \pi \geq 0 \) follows from the known constraints.

**Example 8.** The declaration

\[
\begin{array}{c}
\text{queue}_{A}[[n]] = & \&\{\text{ins} : A \rightarrow \text{queue}_{A}[n + 1], \\
\text{del} : \oplus\{\text{none} : \{n = 0\}, 1, \\
\text{some} : \{n > 0\}. A \otimes \text{queue}_{A}[n - 1]\}\}
\end{array}
\]

is valid: in the \text{ins} branch, we verify \( (n : n \geq 0 \models n + 1 \geq 0) \) while checking validity of \( \text{queue}_{A}[n + 1] \) with rule \text{tdef}; in the \text{some} branch, we add \( n > 0 \) to our constraint \( C \) (due to rule \text{?V}) and verify \( (n : n \geq 0 \land n > 0 \models n - 1 \geq 0) \) while checking validity of \( \text{queue}_{A}[n - 1] \).
We also extend the notation and a congruence on closed valid types.

Definition 9. unfold$_S(V[\pi]) = A[\pi/\pi]$ if $V[\pi | \phi] = A \in \Sigma$ and unfold$_S(A) = A$ otherwise.

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

Definition 10. A relation $R$ on closed valid types is a type bisimulation if $(A, B) \in R$ implies that for $S = \text{unfold}_S(A)$, $T = \text{unfold}_S(B)$ we have the following conditions (in addition to those of Definition 2):
- If $S = \exists \{ \phi \}. A’$ then $T = \exists \{ \psi \}. B’$ and either (i) $\vDash \phi \Rightarrow \vDash \psi$, and $(A’, B’) \in R$, or (ii) $\vDash \neg \phi$ and $\vDash \neg \psi$.
- If $S = \forall \exists \{ \phi \}. A’$ then $T = \forall \exists \{ \psi \}. B’$ and either (i) $\vDash \phi \Rightarrow \vDash \psi$, and $(A’, B’) \in R$, or (ii) $\vDash \neg \phi$ and $\vDash \neg \psi$.
- If $S = \exists m. A’$ then $T = \exists n. B’$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in R$.
- If $S = \forall m. A’$ then $T = \forall n. B’$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in R$.

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 $\phi$ and $\psi$ 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 where unrestricted recursion 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

\[
\begin{align*}
\text{bin}[n] &= \{ \text{b}0 : \exists k. \{ n = 2 \ast k \}. \text{bin}[k], \text{b}1 : \exists k. \{ n = 2 \ast k + 1 \}. \text{bin}[k], \\
\text{e} : \{ n = 0 \}. \text{1} \} \\
\text{zero} &= \{ \text{b}0 : \exists k. \{ k = 0 \}. \text{zero}, \\
\text{e} : \{ 0 = 0 \}. \text{1} \}
\end{align*}
\]

We might expect $\text{bin}[0] \equiv \text{zero}$, but this is not so. A process of type $\text{bin}[0]$ could send the label $\text{b}1$ and an arbitrary value for $k$ and then just loop forever (because there is no proof of $0 = 2k + 1$). The type $\text{zero}$ 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 $\text{zero}$—instead, we just work with $\text{bin}[0]$.

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 (\{?\} \cdot A) where $\phi$ 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 $M$ 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.

- “inc($c_j$); goto $k$” (increment counter $j$ by 1 and go to instruction $k$)
- “zero($c_j$)? goto $k$ : dec($c_j$); 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 $M$ 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 $C \rightarrow C'$ if $C'$ is the result of executing the $i$-th instruction on $C$. If $i = \text{halt}$, then $C = (i, c_1, c_2)$ has no successor configuration. The computation of $M$ is the unique maximal sequence $\rho = \rho(0)\rho(1)\ldots$ such that $\rho(i) \rightarrow \rho(i+1)$ and $\rho(0) = (1,0,0)$. Either $\rho$ is infinite, or ends in $(i, c_1, c_2)$ such that $i = \text{halt}$ and $c_1, c_2 \in \mathbb{N}$.

The halting problem refers to determining whether the computation of a two counter machine $M$ 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 $\Sigma$, two natural numbers $m$ 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 $\Sigma$ 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 $\Sigma$.

We define types $T_{\text{inf}} = \oplus \{ \ell : T_{\text{inf}} \}$ and $T_{\text{inf}}' = \oplus \{ \ell' : T_{\text{inf}}' \}$ for distinct labels $\ell$ and $\ell'$. Next, for every instruction $\iota_i$, we define types $T_i$ and $T_i'$ based on the form of the instruction.

- Case ($\iota_i = \text{inc}(c_1); \text{goto } k$): We define
  
  \[
  T_i[c_1, c_2] = \oplus \{ \text{inc}_1 : T_k[c_1 + 1, c_2] \}
  \]
  
  \[
  T'_i[c_1, c_2] = \oplus \{ \text{inc}_1' : T'_k[c_1 + 1, c_2] \}
  \]

- Case ($\iota_i = \text{inc}(c_2); \text{goto } k$): We define
  
  \[
  T_i[c_1, c_2] = \oplus \{ \text{inc}_2 : T_k[c_1, c_2 + 1] \}
  \]
  
  \[
  T'_i[c_1, c_2] = \oplus \{ \text{inc}_2' : T'_k[c_1, c_2 + 1] \}
  \]

- Case ($\iota_i = \text{zero}(c_1)? \text{goto } k : \text{dec}(c_1); \text{goto } l$): We define
  
  \[
  T_i[c_1, c_2] = \oplus \{ \text{zero}_1 : ?\{c_1 = 0\}.T_k[c_1, c_2], \text{dec}_1 : ?\{c_1 > 0\}.T_i[c_1 - 1, c_2] \}
  \]
  
  \[
  T'_i[c_1, c_2] = \oplus \{ \text{zero}_1' : ?\{c_1 = 0\}.T'_k[c_1, c_2], \text{dec}_1' : ?\{c_1 > 0\}.T'_i[c_1 - 1, c_2] \}
  \]

- Case ($\iota_i = \text{zero}(c_2)? \text{goto } k : \text{dec}(c_2); \text{goto } l$): We define
  
  \[
  T_i[c_1, c_2] = \oplus \{ \text{zero}_2 : ?\{c_2 = 0\}.T_k[c_1, c_2], \text{dec}_2 : ?\{c_2 > 0\}.T_i[c_1, c_2 - 1] \}
  \]
  
  \[
  T'_i[c_1, c_2] = \oplus \{ \text{zero}_2' : ?\{c_2 = 0\}.T'_k[c_1, c_2], \text{dec}_2' : ?\{c_2 > 0\}.T'_i[c_1, c_2 - 1] \}
  \]
Case ($i = \text{halt}$): We define

\[
T_i[c_1, c_2] = T_{\text{inf}} \\
T'_i[c_1, c_2] = T'_{\text{inf}}
\]

We set $A = T_1[m, n]$ and $B = T'_1[m, n]$. Now suppose, the counter machine $M$ 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 $M$ 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 $M$ halts, the first type proceeds with label $\ell$ and the second with $\ell'$ and they are therefore not equal. Hence, the two types are equal iff $M$ does not halt.

We can easily modify this reduction for an isorecursive interpretation of types, by wrapping $\oplus\{\text{unfold} : \_\}$ 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 $A$).

\[
\begin{align*}
\text{queue}_A[n] &= \&\{\text{ins : } \%0[n], \text{del : } \%1[n]\} \\
\%0[n] &= A \rightarrow \text{queue}_A[n + 1] & \%3 = 1 \\
\%1[n] &= \oplus\{\text{none : } \%2[n], \text{some : } \%4[n]\} & \%4[n] = \{n > 0\}.\%5[n] \\
\%2[n] &= ?\{n = 0\}.\%3 & \%5[n | n > 0] = A \otimes \text{queue}_A[n - 1]
\end{align*}
\]

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.
\[
\mathcal{V} ; C ; \Gamma \vdash A \equiv B \quad (\forall \ell \in L)
\]
\[
\mathcal{V} ; C \vdash \bigoplus \{\ell : A_{\ell}\}_{\ell \in L} \equiv \bigoplus \{\ell : B_{\ell}\}_{\ell \in L} \quad \mathcal{V} ; C ; \Gamma \vdash & \{\ell : A_{\ell}\}_{\ell \in L} \equiv & \{\ell : B_{\ell}\}_{\ell \in L}
\]
\[
\mathcal{V} ; C ; \Gamma \vdash A_1 \equiv B_1 \\
\mathcal{V} ; C ; \Gamma \vdash A_2 \equiv B_2 \\
\mathcal{V} ; C ; \Gamma \vdash A_1 \otimes A_2 \equiv B_1 \otimes B_2
\]
\[
\mathcal{V} ; C ; \Gamma \vdash A_1 \equiv B_1 \\
\mathcal{V} ; C ; \Gamma \vdash A_2 \equiv B_2 \\
\mathcal{V} ; C ; \Gamma \vdash A_1 \rightarrow A_2 \equiv B_1 \rightarrow B_2 \\
\mathcal{V} ; C ; \Gamma \vdash 1 \equiv 1
\]
\[
\mathcal{V} ; C ; \Gamma \vdash ?\phi \leftrightarrow \psi \\
\mathcal{V} ; C ; \Gamma \vdash \phi \wedge \psi \rightarrow \phi \wedge \psi \\
\mathcal{V} ; C ; \Gamma \vdash A \equiv B \\
\mathcal{V} ; C ; \Gamma \vdash \exists m. A \equiv \exists n. B
\]
\[
\mathcal{V} ; C ; \Gamma \vdash 1 \\
\mathcal{V} ; C ; \Gamma \vdash \varphi
\]
\[
\mathcal{V} ; C ; \Gamma \vdash e_1 = e_1' \ldots = e_n = e_n' \\
\mathcal{V} ; C ; \Gamma \vdash V[\tau] \equiv V[\tau']
\]

**Figure 1** Algorithmic Rules for Type Equality

The judgment has the form \( \mathcal{V} ; C ; \Gamma \vdash A \equiv B \) where \( \mathcal{V} \) contains the free arithmetic variables in the constraints \( C \) and the types \( A \) and \( B \), and \( \Gamma \) is a collection of closures \( \langle \mathcal{V}' ; C' ; V_1[\tau_1] \equiv V_2[\tau_2]\rangle \). 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 \( V_1[\tau_1][\sigma'] \equiv V_2[\tau_2][\sigma'] \) is given by a substitution \( \sigma' \) over variables in \( \mathcal{V}' \) such that \( \vdash C'[\sigma'] \).

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, \texttt{expd} and \texttt{def}. In the \texttt{expd} rule we expand the definitions of \( V_1[\tau_1] \) and \( V_2[\tau_2] \), and we also add the closure \( \langle \mathcal{V} ; C ; V_1[\tau_1] \equiv V_2[\tau_2]\rangle \) to \( \Gamma \). Since the equality of \( V_1[\tau_1] \) and \( V_2[\tau_2] \) must hold for all its ground instances, the extension of \( \Gamma \) with the corresponding closure remembers exactly that. We can ignore the propositions \( \phi_1 \) and \( \phi_2 \) since the validity of types (rule \texttt{tdef} in Section 3) ensures that both \( \vdash \phi_1[\tau_1/\tau] \) and \( \vdash \phi_2[\tau_2/\tau] \) hold.

In the \texttt{def} rule we close off the derivation successfully if all instances of the equation
$V_1[\overline{\sigma_1}] \equiv V_2[\overline{\sigma_2}]$ are already instances of a closure in $\Gamma$. This is checked by the entailment in the second premise, $\forall : C \vdash \exists V'. C' \land \overline{E_1} \equiv \overline{\sigma'_1} \land \overline{E_2} = \overline{\sigma'_2}$. This entailment is verified as a closed $\forall \exists$ arithmetic formula, even if the original constraints $C$ and $C'$ 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 formula with the same type variables $V_1$ and $V_2$ is already present in $\Gamma$. 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 $k$ formulas with the same type variables $V_1$ and $V_2$ in $\Gamma$.

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

\[
\begin{align*}
\text{intctr}[x, y] &= \& \{ \text{inc} : \text{intctr}[x + 1, y], \\
\text{dec} : \text{intctr}[x, y + 1], \\
\text{sgn} : \oplus \{ \text{neg} : ?\{x < y\}. \text{intctr}[x, y], \\
\text{zer} : ?\{x = y\}. \text{intctr}[x, y], \\
\text{pos} : ?\{x > y\}. \text{intctr}[x, y]\}
\end{align*}
\]

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

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

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

As exemplified by the above example, a distinguishing feature of our algorithm is that it goes beyond reflexivity. Essentially, $V[\overline{\sigma_1}] \equiv V[\overline{\sigma_2}]$ can hold even if $\overline{\sigma_1} \neq \overline{\sigma_2}$. 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 $\forall : C ; \cdot \vdash A \equiv B$ then we should interpret this as stating that for all ground substitutions $\sigma$ over $\forall$ such that $\vdash C[\sigma]$ we have $A[\sigma] \equiv B[\sigma]$. 

Definition 17. Given a relation $R$ on valid ground types and two types $A$ and $B$ such that $\forall \phi : C |- A, B \text{ valid}$, we write $\forall \phi : C |- A \equiv R B$ if for all ground substitutions $\sigma$ over $\phi$ such that $\vdash C[\sigma]$ we have $(A[\sigma], B[\sigma]) \in R$.

Furthermore, we write $\forall \phi : C |- A \equiv B$ if there exists a type bisimulation $R$ such that $\forall \phi : C |- A \equiv R B$.

Note that if $\forall \phi : C |- \bot$, then $\forall \phi : C |- A \equiv B$ is vacuously true, since there does not exist a ground substitution $\sigma$ such that $\vdash C[\sigma]$.

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

Lemma 18. Suppose $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$ holds. Further assume that $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$. Then, $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$ holds.

Proof. To prove $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$, it is sufficient to show that $\exists V \equiv R V_1$ for any substitution $\sigma$ over $\phi$ such that $\vdash C[\sigma]$. Applying this substitution to $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$, we infer $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$ since $\vdash C[\sigma]$.

We construct the bisimulation from a derivation of $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$ 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 $D$ of $\forall \phi : C |- A \equiv B$, we define the set $S(D)$ of closures. For each sequent $\forall \phi : C’ |- \exists V \equiv R V_1 \equiv R V_2$, we include the closure $\forall \phi : C’ |- \exists V \equiv R V_1 \equiv R V_2$ in $S(D)$.

Theorem 20 (Soundness). If $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$, then $\forall \phi : C |- A \equiv B$.

Proof. We are given a derivation $D_0$ of $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$. Construct $S(D_0)$ and define a relation $R$ on closed valid types as follows:

$$R = \{ (A[\sigma], B[\sigma]) \mid \forall \phi : C |- A \equiv B \in S(D_0) \text{ and } \sigma \text{ over } \phi \text{ with } \vdash C[\sigma] \}$$

We prove that $R$ is a type bisimulation. Then our theorem follows since the closure $(V_0 : C_0 |- \exists V \equiv R V_1 \equiv R V_2) \in S(D_0)$. Consider $(A[\sigma], B[\sigma]) \in R$ such that $\forall \phi : C |- A \equiv B$ for some $\sigma$ over $\phi$ and $\vdash C[\sigma]$. First, consider the case where $\forall \phi : C |- \bot$. Under such a constraint, $\forall \phi : C |- \exists V \equiv R V_1 \equiv R V_2$ holds true due to the $\bot$ rule. Furthermore, $\forall \phi : C |- A \equiv B$ holds vacuously, and the algorithm is sound. For the remaining cases, we case analyze on the structure of $A[\sigma]$ and $B[\sigma]$. Consider the case where $A = \exists \{ \ell : A_\ell \} \in L$. Since $A$ and $B$ are both structural, $B = \exists \{ \ell : B_\ell \} \in L$. Since $\forall \phi : C |- A \equiv B$ in $S(D_0)$, we get $(\forall \phi : C |- A \equiv B) \in S(D_0)$ for all $\ell \in L$. By the definition of $R$, we get that $(A_\ell[\sigma], B_\ell[\sigma]) \in R$. Also, $A[\sigma] = \exists \{ \ell : A_\ell[\sigma] \} \in L$ and similarly, $B[\sigma] = \exists \{ \ell : B_\ell[\sigma] \} \in L$. Hence, $R$ satisfies the appropriate closure condition for a type bisimulation.

Next, consider the case where $A = \exists \{ \phi : A’ \}$. Since $A$ and $B$ are both structural, $B = \exists \{ \psi : B’ \}$. Since $\forall \phi : C |- A \equiv B$ in $S(D_0)$, we obtain $\forall \phi : C |- \phi \leftrightarrow \psi$ and $\forall \phi : C \equiv \phi \leftrightarrow \psi$. For the remaining cases, we case analyze on the structure of $A[\sigma]$ and $B[\sigma]$ and assume that there exists a ground substitution $\sigma$ such that $\vdash C[\sigma]$.
Next, consider the case where $A = \exists m. A'$. Since $A$ and $B$ are both structural, $B = \exists n. B'$. Since $(V : C ; A \equiv B) \in S(D_0)$, we get that $(\forall k ; C ; A'[k/m] \equiv B'[k/n]) \in S(D_0)$. Since $k$ was chosen fresh and does not occur in $C$, we obtain that for any $i \in \mathbb{N}$ we have $i \vdash C[\sigma, i/k]$ 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 $S(D_0)$ is the def rule. In this case, adding $(\forall C \Rightarrow V_1[\bar{r}_1] \equiv V_2[\bar{r}_2])$ is redundant: Lemma 18 states that $V_1[\bar{r}_1[\sigma]] \equiv_R V_2[\bar{r}_2[\sigma]]$ which implies $(V_1[\bar{r}_1[\sigma]], V_2[\bar{r}_2[\sigma]]) \in \mathcal{R}$.

### 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

$$\forall V. C \Rightarrow V_1[\bar{r}_1] \equiv V_2[\bar{r}_2]$$

in signatures. Let $\Gamma_\Sigma$ denote the set of all such declarations. Then we check

$$V : C ; \Gamma_\Sigma \vdash V_1[\bar{r}_1] \equiv V_2[\bar{r}_2]$$

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

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

$$\text{queue}_A[n] = \&\{\text{ins} : A \rightarrow \text{queue}_A[n + 1],$$

$$\text{del} : \oplus\{\text{none} : \{n = 0\}. 1,$$

$$\text{some} : \{n > 0\}. A \otimes \text{queue}_A[n - 1]\}$$

$$\text{queue'}_A[n] = \&\{\text{ins} : A \rightarrow \text{queue'}_A[n + 1],$$

$$\text{del} : \oplus\{\text{none} : \{n = 1\}. 1,$$

$$\text{some} : \{n > 1\}. A \otimes \text{queue'}_A[n - 1]\}$$

Our intuition would suggest that $\text{queue}_A[0] \equiv \text{queue'}_A[1]$. But this cannot be directly proved by our equality algorithm. While checking this equality, our algorithm would add $\langle \cdot : \top ; \text{queue}_A[0] \equiv \text{queue'}_A[1]\rangle$ to $\Gamma$ and would continue to check $\text{queue}_A[1] \equiv \text{queue'}_A[2]$ (the ins branch). However, our closure in $\Gamma$ is not sufficient to prove this goal (the def rule fails), and our algorithm reports the types are unequal. However, we can add a general equality declaration $\forall n. \text{queue}_A[n] \equiv \text{queue'}_A[n + 1]$ to the signature. This can be verified by our algorithm since it would add $\langle n : \top ; \text{queue}_A[n] \equiv \text{queue'}_A[n + 1]\rangle$ to $\Gamma$ and use it to prove $\text{queue}_A[n + 1] \equiv \text{queue'}_A[n + 2]$ in the ins branch. Then, we will use the same equality declaration from the signature to verify $\text{queue}_A[0] \equiv \text{queue'}_A[1]$ 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
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 presenta-
tion 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, R (ms) and T (ms) show the reconstruction and type-checking

<table>
<thead>
<tr>
<th>Module</th>
<th>LOC</th>
<th>#Defs</th>
<th>R (ms)</th>
<th>T (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>143</td>
<td>8</td>
<td>0.353</td>
<td>1.325</td>
</tr>
<tr>
<td>integers</td>
<td>114</td>
<td>8</td>
<td>0.200</td>
<td>1.074</td>
</tr>
<tr>
<td>linlam</td>
<td>67</td>
<td>6</td>
<td>0.734</td>
<td>4.003</td>
</tr>
<tr>
<td>list</td>
<td>441</td>
<td>29</td>
<td>1.534</td>
<td>3.419</td>
</tr>
<tr>
<td>primes</td>
<td>118</td>
<td>8</td>
<td>0.196</td>
<td>1.646</td>
</tr>
<tr>
<td>segments</td>
<td>65</td>
<td>9</td>
<td>0.239</td>
<td>0.195</td>
</tr>
<tr>
<td>ternary</td>
<td>235</td>
<td>16</td>
<td>0.550</td>
<td>1.967</td>
</tr>
<tr>
<td>theorems</td>
<td>141</td>
<td>16</td>
<td>0.361</td>
<td>0.894</td>
</tr>
<tr>
<td>tries</td>
<td>308</td>
<td>9</td>
<td>1.113</td>
<td>5.283</td>
</tr>
</tbody>
</table>

Total   1632 109  5.280  19.806
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 $\lambda$-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 $\text{seg}[n] = \forall k. \text{list}[k] \rightarrow \text{list}[n + k]$ 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 $\lambda$-calculus

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

An expression is either a $\lambda$-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 $\lambda$-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 $\lambda$-abstraction

$\text{val}[n] = \oplus\{\text{lam} : \exists \{n > 0\}. \forall n_1. \text{exp}[n_1] \rightarrow \text{exp}[n_1 + n - 1]\}$

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

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

### 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 \( \pi \)-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 \textit{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


Bertrand Meyer. Applying “design by contract”.


