Verified Linear Session-Typed Concurrent Programming

Ankush Das
Carnegie Mellon University
Pittsburgh, PA, USA

Frank Pfenning
Carnegie Mellon University
Pittsburgh, PA, USA

ABSTRACT

We present a system of linear session types that integrates several features aimed at verification of different properties of concurrent programs, specifically types indexed with arithmetic expressions, linear constraints and quantification. We prove the standard type safety properties of session fidelity and deadlock freedom. In order to control the verbosity of programs we introduce implicit syntax and an algorithm for reconstruction, which is complete under some mild assumptions on the structure of types. We then illustrate the expressive power of our language (called Rast) with a variety of examples, including normalization for the linear \( \lambda \)-calculus, balanced ternary arithmetic, binary counters and tries.

CCS CONCEPTS

- Theory of computation → Linear logic; Type theory; Concurrency, Linear types, Verification

KEYWORDS

Session types, Concurrence, Linear types, Verification

ACM Reference Format:

1 INTRODUCTION

Session types [20–22, 33] provide a structured way of prescribing communication protocols of message-passing systems. This paper focuses on binary session types governing the interactions along channels with two endpoints. Binary session types without general recursion exhibit a Curry-Howard isomorphism with linear logic [5, 6, 34] and are therefore of particular foundational significance. Moreover, type safety derives from properties of cut reduction and guarantees deadlock freedom (global progress) and session fidelity (type preservation) ensuring that at runtime the sender and receiver exchange messages conforming to the channel’s type.

However, even in the presence of recursive types, the kinds of protocols that can be specified are limited, which has led to a number of extensions, such as context-free session types [1, 28], label-dependent session types [29], and general dependent session types [17, 24, 30, 31]. In prior work, we have proposed arithmetically refined session types [12] and have investigated their properties independently of any specific programming language. With arithmetically refined types we can, for example, express a protocol that sends a natural number \( n \) and then a sequence of messages exactly of length \( n \), and many more complex protocols (for additional examples, see Section 6). We found that type equality, naturally defined via a bisimulation between observable communication behaviors, is undecidable, but also proposed a simple and practical algorithm.

In this paper we present the design, theory, and pragmatics of a programming language for processes in which type checking guarantees compliance with arithmetically refined session types. Here, type checking is defined over a language where constructs related to arithmetic constraints have explicit communication counterparts.

We observe, however, that many programs in this explicit language are unnecessarily verbose and therefore tedious for the programmer to write, because the process constructs pertaining to the refinement layer contribute only to verifying its properties, but not its observable computational outcomes. As is common for refinement types, we therefore also designed an implicit language for processes where most constructs related to index refinements are omitted. The problem of reconstruction is then to map such an implicit program to an explicit one. We provide an algorithm for reconstruction that is complete (if there is a reconstruction, it can be found). This algorithm exploits proof-theoretic properties of the sequent calculus akin to focusing [2] to avoid backtracking and consequently provides precise error messages that we have found to be helpful.

Thus, our main results are the following:

1. The design of an explicit language with a bidirectional type-checking algorithm that is sound and complete relative to an oracle for type equality.
2. A type soundness theorem that establishes session fidelity (type preservation) and deadlock freedom (global progress) for well-typed programs.
3. The design of a significantly more compact implicit syntax and a reconstruction algorithm producing explicit programs. Reconstruction is complete under some mild conditions on the language of types.
4. Several case studies that explore the possibilities and limitations of program properties that can be captured with arithmetic refinements.

We have already reported on the implementation of the design and theory presented here in a system description that overviews the Rast programming language [11]. All examples in this paper have been type-checked and executed in Rast and are publicly available [27]. Due to space constraints, there is an important aspect of Rast that we do not cover in this paper: it provides ergonomic [10] and temporal [9] types to measure and verify (amortized) work and
span of concurrently executing session-typed programs. The above-cited prior works manually compute and check complexity bounds at an informal metalevel. The arithmetic refinements in the Rast language allow us to internally express these bounds and, for the first time, verify them automatically.

The rest of the paper is organized as follows: Section 2 overviews the Rast language with an illustrative queue data structure. Section 3 formalizes the type system, semantics and type safety of Rast; Section 4 presents the reconstruction algorithm. Section 5 describes the implementation and Section 6 highlights some interesting examples with their key properties verified in Rast. Finally, Section 7 describes related work and Section 8 concludes.

2 OVERVIEW OF REFINED SESSION TYPES

Basic session types have limited expressivity. As a simple example, consider the session type provided by a queue data structure storing elements of type \( A \).

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

This type describes a queue interface supporting insertion and deletion. The \textit{external choice} operator \& dictates that the process providing this data structure accepts either one of two messages: the labels \text{ins} or \text{del}. In the case of the label \text{ins}, it then receives an element of type \( A \) denoted by the \( \rightarrow \) operator, and then the type recurses back to \text{queue}^1_A. On receiving a \text{del} request, the process can respond with one of two labels (\text{none} or \text{some}), indicated by the \textit{internal choice} operator @. It responds with \text{none} and then \textit{terminates} (indicated by 1) if the queue is empty, or with \text{some} followed by the element of type \( A \) (expressed with the \( \otimes \) operator) and recurses if the queue is nonempty. However, the simple session type does not express the conditions under which the \text{none} and \text{some} branches must be chosen, which requires tracking the length of the queue in the type.

We enhance the session type with a simple arithmetic refinement. The more precise type

\[
\text{queue}^n_A = \&\{ \text{ins} : A \rightarrow \text{queue}^1_A[n + 1], \\
\text{del} : @\{\text{none} : ![n = 0], 1, \\
\text{some} : ![n > 0] \otimes \text{queue}^1_A[n - 1]\}\}
\]

uses the index refinement \( n \) to indicate the size of the queue. In addition, the refined type uses a type \textit{constraint} \(![\phi]\). A which can be read as “there exists a proof of \( \phi \)”. Here, the process providing the queue must (conceptually) send a proof of \( n = 0 \) after it sends \text{none}, and a proof of \( n > 0 \) after it sends \text{some}. It is therefore constrained in its choice between the two branches based on the value of the index \( n \). Because the index domain from which the propositions \( \phi \) are drawn is Presburger arithmetic and hence decidable, no proof of \( \phi \) will actually be sent, but we can nevertheless verify the constraint statically. The dual to \( ![\phi] \). A is the type constraint \( ![\phi] \). A to be interpreted as “for all proofs of \( \phi \).” The refinement type system also supports explicit quantifiers \( \exists n \). A and \( \forall n \). A that send and receive natural numbers, respectively. Because intrinsic properties of data structures (such as the number of elements) must be nonnegative we work over the natural numbers \( 0, 1, \ldots \) rather than general integers. This includes a static validity check for types to ensure that all index refinements are nonnegative. For example, while checking the validity of \( \text{queue}^n_A[n] \), we encounter the constraint \( n > 0 \) in the \text{some} branch, so we assume it and then verify that \( n - 1 \geq 0 \), ensuring the validity of \( \text{queue}^n_A[n-1] \).

Our language design is based on two key dual principles: the type \( ![\phi] \). A corresponds to an \textit{assertion} of \( \phi \), whereas the type \( ![\phi] \). A corresponds to an \textit{assumption} of \( \phi \). Consequently, we introduce dual process terms: (i) assert \( x \{ \phi \} \) to assert constraint \( \phi \) on channel \( x \), and dually, (ii) assume \( x \{ \phi \} \) to assume \( \phi \) on \( x \). Following the same principle, we observe that \( \exists n \). A requires the provider to send a natural number and \( \forall n \). A mandates the receiver to receive a natural number. Thus, we introduce (i) send \( x \{ e \} \) to send an arithmetic expression \( e \) on channel \( x \) and (ii) \( ![n] \leftarrow x \) recv \( x \) to receive a natural number on channel \( x \) and bind it to variable \( n \).

One parallel implementation of such a queue data structure is a sequence of \textit{elem} processes, each storing an element of the queue, terminated by an \textit{empty} process, representing the empty queue. Figures 1 and 2 describe the sequence diagrams for insertion and deletion w.r.t. this implementation of queues. In Figure 1, the initial queue (of size 1) consists of an \textit{empty} process that \textit{provides} along \( c_1 : \text{queue}^1_A[0] \) (indicated by \( \bullet \) between \textit{empty} and \( c_1 \)) and does not use any channels, and an \textit{elem} process that uses \( c_1 : \text{queue}^1_A[0] \), and an element of type \( A \) (not shown) and \textit{provides} \( c_2 : \text{queue}^1_A[1] \). The \textit{ins} message is first received on \( c_2 \), then passed on to \( c_1 \), which spawns a new \textit{empty} process offering on \( c_0 : \text{queue}^1_A[0] \). The original process offering on \( c_1 \) then transitions to \textit{elem}. In Figure 2, the \textit{del} message is received on \( c_2 \), which replies with the \textit{some} label and an element \( x \) stored inside (not shown) the offering \textit{elem} process. The process then terminates by \textit{forwarding} channel \( c_2 \) onto \( c_1 \). As demonstrated by Figures 1 and 2, insertions take place at the tail of the queue while deletions occur at the head of the queue.
1: \( \vdash \) empty :: \( (s : \text{queue}_A[0]) \)
2: \( s \leftarrow \) empty =
3: \( \text{case} \ s (\)
4: \( \text{ins} \Rightarrow x \leftarrow \text{recv} \ s ; \quad \% (x : A) \vdash (s : \text{queue}_A[1]) \)
5: \( e \leftarrow \) empty ; \quad \% (x : A), (e : \text{queue}_A[0]) \vdash (s : \text{queue}_A[1]) \)
6: \( s \leftarrow \text{elem}[0] \times e \)
7: \( | \text{del} \Rightarrow s.\text{none} ; \quad \% \cdot (s : \{0 \leq \ 0, \ 1\}) \)
8: \( \text{assert} s \{0 = 0\} ; \quad \% \cdot (s : 1) \)
9: \( \text{close} s \)
10: \( (x : A), (t : \text{queue}_A[n]) \vdash \text{elem}[n] : (s : \text{queue}_A[n + 1]) \)
11: \( s \leftarrow \text{elem}[n] \times t = \)
12: \( \text{case} \ s (\)
13: \( \text{ins} \Rightarrow y \leftarrow \text{recv} \ s ; \)
14: \( t.\text{ins} ; \)
15: \( \text{send} t \ y ; \quad \% (x : A), (t : \text{queue}_A[n + 1]) \vdash (s : \text{queue}_A[n + 2]) \)
16: \( s \leftarrow \text{elem}[n + 1] \times t \)
17: \( | \text{del} \Rightarrow s.\text{some} \)
18: \( \text{assert} s \{n + 1 > 0\} ; \)
19: \( \text{send} s \ x ; \quad \% (t : \text{queue}_A[n]) \vdash (s : \text{queue}_A[n]) \)
20: \( s \leftarrow t \)

Figure 3: Implementations for the empty and elem processes.

Formally, the empty process offers type \( \text{queue}_A[0] \) while the elem\( [n] \) process (indexed by arithmetic variable \( n \)) uses channels of type \( \text{queue}_A[n] \) and \( A \) and offers type \( \text{queue}_A[n + 1] \). In our notation, the process declarations will be written as (used channels on the left and provided channel on the right)

\[ \vdash \text{empty} :: (s : \text{queue}_A[0]) \]

\[ (x : A) (t : \text{queue}_A[n]) \vdash \text{elem}[n] : (s : \text{queue}_A[n + 1]) \]

Figure 3 presents the implementation of empty and elem processes along with their derivations on the right (type \( \text{queue}_A[n] \) abbreviated to \( \text{queue}_A[n] \)). Upon receiving the ins label and element \( x : A \) (line 4), the empty process spawns a new empty process (line 5), binds it to channel \( e \), and tail calls elem\( [0] \) (line 6). On inputting the del label, the empty process takes the none branch (line 7) since it stores no elements. Therefore, it needs to send a proof of \( n = 0 \), and since it provides \( \text{queue}_A[0] \), it sends the trivial proof of \( 0 = 0 \) (line 8), and closes the channel terminating communication (line 9). The elem process receives the ins label and element \( y : A \) (line 13), passes on these two messages on the tail \( t \) (lines 14,15), and recurses with elem\( [n + 1] \) (line 16). The type expected by elem\( [n + 1] \) indeed matches the type of the input and output channels, as confirmed by the process declaration. On receiving the del label, the elem process replies with the some label (line 17) and the proof of \( n + 1 > 0 \) (line 18), again trivial since \( n \) is a natural number. It terminates with forwarding \( s \) along the edge (line 20). This forwarding is valid since the types of \( s \) and \( t \) exactly match as described by the id rule in Section 3.1 (corresponds to identity). The programmer is not burdened with writing the asserts (in blue) as they are automatically inserted by our reconstruction algorithm (Section 4).

At runtime, each arithmetic proposition will be closed, so if it has no quantifiers it can simply be evaluated. In the presence of quantifiers, a decision procedure for Presburger arithmetic can be applied dynamically (if desired, or if a provider or client may not be trusted), but no actual proof object needs to be transmitted.

An interesting corner case would be, say, if a process with one element (\( n = 1 \)) responded with none to the del request. It would have to follow up with a proof that \( 1 = 0 \), which is of course impossible. Therefore, our refinements guarantee that no further communication along this channel could take place.

### 3 BASIC AND REFINED SESSION TYPES

This section presents the basic system of session types and its arithmetic refinement along with corresponding process terms and typing rules. The underlying base system of session types is derived from a Curry-Howard interpretation \([5, 6]\) of intuitionistic linear logic \([16]\). The key idea is that an intuitionistic linear sequent \( A_1, A_2, \ldots, A_n \vdash A \) is interpreted as the interface to a process expression \( P \). We label each of the antecedents with a channel name \( x_i \) and the succedent with channel name \( z \). The \( x_i \)'s are channels used by \( P \) and \( z \) is the channel provided by \( P \).

\[ x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash P :: (z : C) \]

The resulting judgment formally states that process \( P \) provides a service of session type \( C \) along channel \( z \), while using the services of session types \( A_1, \ldots, A_n \) provided along channels \( x_1, \ldots, x_n \) respectively. We abbreviate the antecedent of the sequent by \( \Delta \).

In addition to the type constructors arising from the connectives of intuitionistic linear logic \((\&), (\sqsupseteq), (\neg), \rightarrow)\), we have type names, indexed by a sequence of arithmetic expressions \( V(F) \), existential and universal quantification over natural numbers \((\exists n. A), (\forall n. A)\) and existential and universal constraints \((?\phi, A), (!\phi, A)\). We write \( i \) for constant and \( n \) for variable natural numbers.

**Types**

\[ A, B ::= \psi(F), \psi(\ell : A \Gamma), \psi(P), \psi(A), \psi(A, B) \]

**Arith. Expns.**

\[ e ::= i | e + e | e - e | i * x e | n \]

**Arith. Props.**

\[ \phi ::= e = e | e > e | T \lor T \lor \phi \land \phi \]

**Proc**

\[ P, Q ::= x.k \cdot P | \text{case} x (l \Rightarrow) \Gamma \cdot L | \text{send} x ; P | \text{close} x ; P | x \leftarrow y ; P | \text{assert} x \{ \phi \} ; P | \text{send} x \{ e \} ; P \]

Our implementation does not support type polymorphism but it is convenient in some of the examples. We therefore allow definitions such as \( \text{queue}_A[n] = \ldots \) and interpret them as a family of definitions, one for each possible type \( A \).

The typing judgment has the form of a sequent

\[ V \vdash C ; \Delta \vdash P :: (x : A) \]

where \( V \) are index variables, \( n, C \) are constraints over these variables expressed as a single proposition, \( \Delta \) are the linear antecedents \( x_1 : A_1, P \) is a process expression, and \( x : A \) is the linear succedent.

We propose and maintain that the \( x_1 \)'s and \( x \) are all distinct, and that all free index variables in \( C, \Delta, P \) and \( A \) are contained among \( V \). Finally, \( \Sigma \) is a fixed signature containing type and process definitions (explained in Section 3.1) Because it is fixed, we elide it from
In addition we write \( V : C \vdash \phi \) for semantic entailment (proving \( \phi \) assuming \( C \)) in the constraint domain where \( V \) contains all arithmetic variables in \( C \) and \( \phi \). Table 1 presents the session types and their associated process terms, their continuation (both in types and terms) and operational description.

We formalize the operational semantics as a system of multiset rewriting rules \[7\]. We introduce semantic objects \( \text{proc}(c, P) \) and \( \text{msg}(c, M) \) which mean that process \( P \) or message \( M \) provide along channel \( c \). A process configuration is a multiset of such objects, where any two channels provided are distinct (formally described in Section 3.3).

### 3.1 Basic Session Types

In this subsection, we review the syntax and semantics for the basic session type operators \( \& \), \( \otimes \), \( \rightsquigarrow \), \( \leftarrow \) and \( 1 \). A summary of the corresponding process terms and intuitive explanation for semantics is provided in Table 1.

#### External Choice

The \textbf{external choice} type constructor \( \& \ell : A_{\ell} \ell \in L \) is an \( n \)-ary labeled generalization of the additive conjunction \( A \& B \). Operationally, it requires the provision of \( x : \& \ell : A_{\ell} \ell \in L \) to branch based on the label \( k \in L \) it receives from the client and continue to provide type \( A_k \). The corresponding process term is written as case \( x (\ell \Rightarrow P \ell \in L) \). Dually, the client must send one of the labels \( k \in L \) using the process term \( (x.k : Q) \) where \( Q \) is the continuation.

\[
\begin{align*}
\forall \ell \in L \quad V : C; \Delta \vdash P \ell : (x : A_{\ell}) & \quad \&R \\
(k \in L) \quad V : C; \Delta, (x : A_k) \vdash Q \ell : (z : C) & \quad \&L \\
(k \in L) \quad V : C; \Delta, (x : \& \ell : A_{\ell} \ell \in L) \vdash (x.k : Q) \ell : (z : C) & \quad \&S
\end{align*}
\]

Communication is asynchronous, so that the client \( x.k : Q \) sends a message \( k \) along \( c \) and continues as \( Q \) without waiting for it to be received. As a technical device to ensure that consecutive messages on a channel arrive in order, the sender also creates a fresh continuation channel \( c' \) so that the message \( k \) is actually represented as \( (c.k : c \Rightarrow c') \) (read: send \( k \) along \( c \) and continue along \( c' \)). When the message \( k \) is received along \( c \), we select branch \( k \) and also substitute the continuation channel \( c' \) for \( c \). Rules \( \&S \) and \( \&C \) below describe the operational behavior of the provider and client respectively (\( c' \) fresh).

\[
\begin{align*}
(\&S) & : \text{proc}(d, c.k : Q) \Rightarrow \text{msg}(c', c.k : c' \Rightarrow c), \text{proc}(d, Q[c'/c]) \\
(\&C) & : \text{proc}(c', c.k : c' \Rightarrow c) \Rightarrow \text{proc}(c', Q[c'/c])
\end{align*}
\]

The \textbf{internal choice} constructor \( \ell : A_{\ell} \ell \in L \) is the dual of external choice requiring the provider to send one of the labels \( k \in L \) that the client must branch on.

\[
\begin{align*}
(k \in L) \quad V : C; \Delta \vdash P \ell : (x : A_k) & \quad \otimes R \\
V : C; \Delta \vdash (x.k : P) : (x : \otimes (\ell : A_{\ell} \ell \in L)) & \quad \otimes L
\end{align*}
\]

This dual constructor reverses the role of the provider and client. The provider \( (x.k : P) \) of \( x : \otimes (\ell : A_{\ell} \ell \in L) \) sends the label \( k \) along \( x \) and continues to provide \( x : A_k \). Correspondingly, the client branches on the label received using channel \( x : A_{\ell} \) in branch \( \ell \) with process term \( Q \ell \). The rules of operational semantics \( \otimes S, \otimes C \) are exact dual of \( \& S \) and \( \& C \) and omitted for brevity.

#### Channel Passing

The \textbf{tensor} operator \( A \otimes B \) prescribes that the provider of \( x : A \otimes B \) sends a channel \( y \) of type \( A \) and continues to provide type \( B \). The corresponding process term is send \( x.y : P \) where \( P \) is the continuation. Correspondingly, its client receives a channel using the term \( y \Rightarrow \text{recv} x : Q \), binding it to variable \( y \) and continuing to execute \( Q \).

\[
\begin{align*}
V : C; \Delta \vdash P : (x : B) & \quad \otimes R \\
V : C; \Delta, (y : A) \vdash (\text{send} x.y : P) : (x : A \otimes B) & \quad \otimes L
\end{align*}
\]

Operationally, the provider \( \text{send} c.d : P \) sends the channel \( d \) and the continuation channel \( c' \) along \( c \) as a message and continues with executing \( P \). The client receives the channel \( d \) and continuation channel \( c' \) and substitutes \( d \) for \( x \) and \( c' \) for \( c \).

\[
\begin{align*}
(\otimes S) & : \text{proc}(c, c.d : P) \Rightarrow \text{proc}(c', P[c'/c]) \\
\text{msg}(c, c.d : c \Rightarrow c') & \Rightarrow \text{proc}(c, Q[c'/c])
\end{align*}
\]
The dual operator \(A \rightarrow B\) allows the provider to receive a channel of type \(A\) and continue to provide type \(B\). The client of \(A \rightarrow B\), on the other hand, sends the channel of type \(A\) and continues to use \(B\).

\[
\alpha R \quad \bar{\nu} ; C ; A ; \Delta (y : A) \vdash P :: (x : B) \\
\alpha L \quad \nu ; C ; \Delta + (y \leftarrow \text{recv} \ x ; P) :: (x : A \rightarrow B)
\]

**Termination.** The type \(1\), the multiplicative unit of linear logic, indicates termination requiring that the provider send a close message followed by terminating the communication. Linearity enforces that the provider not use any channels.

\[
\nu ; C ; A ; \Delta (x : A) \vdash Q :: (z : C) \\
\nu ; C ; \Delta (x : A \rightarrow B), (y : A) + (\text{send} \ x \ y ; Q) :: (z : C)
\]

Operationally, the provider waits for the closing message, which has no continuation channel since the provider terminates.

**Forwarding.** A process \(x \leftrightarrow y\) identifies the channels \(x\) and \(y\) so that any further communication along either \(x\) or \(y\) will be along the unified channel. Its typing rule corresponds to the logical rule of identity.

\[
\nu ; C ; y : A \vdash (x \leftarrow y) :: (x : A)
\]

Operationally, a process \(c \leftrightarrow d\) forwards any message \(M\) that arrives on \(d\) and \(c\) vice-versa. Since channels are used linearly, the forwarding process can then terminate, ensuring proper renaming, as exemplified in the rules below.

\[
\nu C (d, M) ; \text{proc}(c, \text{close} \ c) \mapsto \text{msg}(c, \text{close} \ c) \\
\nu C (d, M) ; \text{proc}(c, \text{wait} \ c ; Q) \mapsto \text{msg}(d, Q)
\]

**Type Definitions.** As our queue example already showed, session types can be defined recursively, departing from a strict Curry-Howard interpretation of linear logic, analogous to the way pure ML or Haskell depart from a pure interpretation of intuitionistic logic. For this purpose we allow (possibly mutually recursive) type definitions \(V[\bar{n}] = A\) in the signature \(\Sigma\). Here, \(\bar{n}\) denotes a sequence of arithmetic variables. Again, for a well-formed signature, we require \(A\) to be contractive [15] meaning \(A\) should not itself be a type name. Our type definitions are *equirecursive* so we can silently replace type names \(V[\bar{f}]\) indexed with arithmetic refinements by \(A[\bar{f}]/\bar{n}\) during type checking, and no explicit rules for recursive types are needed.

**Type Definitions**

All types in a signature must be *valid*, formally denoted with the judgment \(\nu C ; A \vdash \text{valid}\), which requires that all free arithmetic variables of \(C\) and \(A\) be contained in \(V\), and that for each arithmetic expression \(e\) in \(A\) we can prove \(V' ; C' + e : \text{nat}\) for the constraints \(C'\) known at the occurrence of \(e\) (implicitly proving that \(e \geq 0\)).

### 3.2 The Refinement Layer

We now describe quantifiers (\(\exists n, A, \forall n, A\)) and constraints (\(\exists n, A, \forall n, A\)) [12]. An overview of the types, process expressions, and their operational meaning can be found in Table 2.

**Quantification.** The provider of \((c : \exists n, A)\) should send a witness \(n\) along channel \(c\) and then continue as \(A[n]/[\bar{n}]\). The witness is specified by an arithmetic expression \(e\) which, since it must be closed at runtime, can be evaluated to a number \(i\) (following standard evaluation rules of arithmetic). From the typing perspective, we just need to check that the expression \(e\) denotes a natural number, using only the permitted variables in \(V\). This is represented with the auxiliary judgment \(V ; C ; e : \text{nat}\) (implicitly proving that \(e \geq 0\) under constraint \(C\)).

\[
\nu V ; C ; e : \text{nat} \quad \nu V ; C ; A \vdash P :: (x : A[\bar{e}/\bar{n}]) \\
\nu V ; C ; A \vdash \text{send} \ x ; P :: (x : A[\bar{n}]) \quad \exists R
\]

\[
\nu V ; n : C ; A ; (x : A) + Q :: (z : C) \quad (n \text{ fresh}) \quad \exists L
\]

Statically, the client adds \(n\) to \(V\) to ensure that \(Q\) and \(A\) are closed w.r.t. \(V\). Operationally, the provider sends the arithmetic expression with the continuation channel as a message that the client receives and appropriately substitutes.

\[
\nu C (c, \text{send} \ e ; P) \mapsto \text{proc}(e', P[c'/c]), \text{msg}(c, \text{send} \ e ; c \leftrightarrow c') \\
\nu C (c, \text{send} \ e ; c \leftrightarrow c') \mapsto \text{proc}(d, Q[e/n][e'/c])
\]

The declaration of \(f\) is looked up in the signature \(\Sigma\) (first premise), and \(\bar{\Sigma}\) is substituted for \(\bar{\Sigma}\) while matching the types in \(\Delta'\) and \(\bar{\Sigma}\) (second premise). Similarly, the freshly created channel \(x\) has type \(A\) from the signature with \(\bar{\Sigma}\) substituted for \(\bar{\Sigma}\). The corresponding semantics rule also performs a similar substitution (a fresh).

\[
\text{def} \quad \text{proc}(c, x \leftarrow f[\bar{f}] ; Q) \mapsto \text{proc}(a, P[f/a/x, \bar{f}][\bar{f}/\bar{n}], \text{proc}(c, Q[a/x]))
\]

where \(\bar{f}/B ; f[\bar{f}] = P_f :: (x' : A) \in \Sigma\).

\[
\nu V \vdash C ; \Delta (x : A \rightarrow B), (y : A) + (\text{send} \ x \ y ; Q) :: (z : C)
\]

\[
\nu V ; C ; \Delta (x : A) + Q :: (z : C) \quad \text{def}
\]

The declaration of \(f\) is looked up in the signature \(\Sigma\) (first premise), and \(\bar{\Sigma}\) is substituted for \(\bar{\Sigma}\) while matching the types in \(\Delta'\) and \(\bar{\Sigma}\) (second premise). Similarly, the freshly created channel \(x\) has type \(A\) from the signature with \(\bar{\Sigma}\) substituted for \(\bar{\Sigma}\). The corresponding semantics rule also performs a similar substitution (a fresh).

\[
\text{def} \quad \text{proc}(c, x \leftarrow f[\bar{f}] ; Q) \mapsto \text{proc}(a, P[f/a/x, \bar{f}][\bar{f}/\bar{n}], \text{proc}(c, Q[a/x]))
\]

where \(\bar{f}/B ; f[\bar{f}] = P_f :: (x' : A) \in \Sigma\).

\[
\nu V \vdash C ; \Delta (x : A \rightarrow B), (y : A) + (\text{send} \ x \ y ; Q) :: (z : C)
\]

\[
\nu V ; C ; \Delta (x : A) + Q :: (z : C) \quad \text{def}
\]

The declaration of \(f\) is looked up in the signature \(\Sigma\) (first premise), and \(\bar{\Sigma}\) is substituted for \(\bar{\Sigma}\) while matching the types in \(\Delta'\) and \(\bar{\Sigma}\) (second premise). Similarly, the freshly created channel \(x\) has type \(A\) from the signature with \(\bar{\Sigma}\) substituted for \(\bar{\Sigma}\). The corresponding semantics rule also performs a similar substitution (a fresh).

\[
\text{def} \quad \text{proc}(c, x \leftarrow f[\bar{f}] ; Q) \mapsto \text{proc}(a, P[f/a/x, \bar{f}][\bar{f}/\bar{n}], \text{proc}(c, Q[a/x]))
\]

where \(\bar{f}/B ; f[\bar{f}] = P_f :: (x' : A) \in \Sigma\).

\[
\nu V \vdash C ; \Delta (x : A \rightarrow B), (y : A) + (\text{send} \ x \ y ; Q) :: (z : C)
\]

\[
\nu V ; C ; \Delta (x : A) + Q :: (z : C) \quad \text{def}
\]
The client sends (the value of) an arithmetic expression \( e \) which the provider receives and binds to \( n \).

\[
\frac{\forall \ n. \ A \quad \Delta \vdash P_n :: (x : A)}{\forall \ n. \ A \quad \Delta \vdash (n) \leftarrow \text{recv } c \ ; \ P_n :: (x : \forall n. A) \quad \forall R}
\]

\[
\frac{\forall \ n. \ A \quad \Delta \vdash (n) \leftarrow \text{recv } c \ ; \ P_n :: (x : \forall n. A)}{\forall \ n. \ A \quad \Delta \vdash \text{send } x \ [e] \ ; \ Q :: (z : C) \quad \forall L}
\]

\[\text{(YS)} : \text{proc}(d, \text{send } c \ [e] \ ; P) \mapsto \text{msg}(c', \text{send } c \ [e] ; \ c' \mapsto c), \text{proc}(d, [c'/c]P)\]

\[\text{(YC)} : \text{proc}(d, [n] \leftarrow \text{recv } c ; Q), \text{msg}(c', \text{send } c \ [e] ; \ c' \mapsto c) \mapsto \text{proc}(d, [e/n][c'/c]Q)\]

**Constraints.** Refined session types also allow constraints over index variables. As we have already seen in the examples, these critically govern permissible messages. From the message-passing perspective, the provider of \( (c : \forall \phi. A) \) should send a proof of \( \phi \) along \( c \) and the client should receive such a proof. However, since the index domain is decidable and future computation cannot depend on the form of the proof (what is known in type theory as *proof irrelevance*) such messages are not actually exchanged. Instead, it is the provider’s responsibility to ensure that \( \phi \) holds, while the client is permitted to assume that \( \phi \) is true. Therefore, and in an analogy with imperative languages, we write assert \( c \ [\phi] \ ; P \) for a process that asserts \( \phi \) for channel \( c \) and continues with \( P \), while assume \( c \ [\phi] ; Q \) assumes \( \phi \) and continues with \( Q \).

Thus, the typing rules for this new type constructor are

\[
\frac{\forall \ n. \ A \quad \Delta \vdash \text{assert } x \ [\phi] \ ; P :: (x : ?\phi. A)}{\forall \ n. \ A \quad \Delta \vdash \text{assert } x \ [\phi] \ ; P :: (x : ?\phi. A) \quad \forall R}
\]

\[
\frac{\forall \ n. \ A \quad \Delta \vdash \text{assert } x \ [\phi] \ ; P :: (x : ?\phi. A)}{\forall \ n. \ A \quad \Delta \vdash \text{assert } x \ [\phi] \ ; P :: (x : ?\phi. A) \quad \forall R}
\]

Notice how the provider must verify the truth of \( \phi \) given the currently known constraints \( C \), while the client assumes \( \phi \) by adding it to \( C \).

Operationally, the provider creates a message containing the constraint that is received by the client \( (c' \text{ fresh}) \).

\[
\text{(YS)} : \text{proc}(c, \text{assert } c \ [\phi] \ ; P) \mapsto \text{proc}(c', [c'/c]P), \text{msg}(c, \text{assert } c \ [\phi] ; c \mapsto c')
\]

\[
\text{(YC)} : \text{msg}(c, \text{assert } c \ [\phi] ; c \mapsto c'), \text{proc}(d, \text{assume } c \ [\phi'] ; Q) \mapsto \text{proc}(d, [c'/c]Q)
\]

**Table 2:** Refined session types with operational description

In well-typed configurations (which arise from executing well-typed processes) the constraint \( \phi \) in these rules will always be closed and true so there is no need to check this explicitly.

The dual operator \( !\phi \).\( A \) reverses the role of provider and client. The provider of \( x : !\phi \).\( A \) may assume the truth of \( \phi \), while the client must verify it. The dual rules are

\[
\frac{\forall \ n. \ A \quad \Delta \vdash \text{assert } x \ [\phi] \ ; P :: (x : !\phi. A)}{\forall \ n. \ A \quad \Delta \vdash \text{assert } x \ [\phi] \ ; P :: (x : !\phi. A) \quad \forall R}
\]

The remaining issue is how to type-check a branch that is impossible due to unsatisfiable constraints. For example, if a client sends a del request to a provider along \( c : \text{queue}_A[0] \), the type then becomes

\[
c : @\{\text{none} : ?0=0\}, \text{ some} : ?0>0. \ A \otimes \text{queue}_A[0-1]\}
\]

The client would have to branch on the label received and then assume the constraint asserted by the provider

\[
\text{case } c \ {\text{none}} \Rightarrow \text{assume } c \ [0 = 0] ; P_1 \quad | \quad \text{some} \Rightarrow \text{assume } c \ [0 > 0] ; P_2
\]

but what could we write for \( P_2 \) in the some branch? Intuitively, computation should never get there because the provider cannot assert \( 0 > 0 \) formally, we use the process expression ‘impossible’ to indicate that computation can never reach this spot:

\[
\text{case } c \ {\text{none}} \Rightarrow \text{assume } c \ [0 = 0] ; P_1 \quad | \quad \text{some} \Rightarrow \text{assume } c \ [0 > 0] ; \text{impossible}
\]

In implicit syntax (see Section 4) we could omit the some branch altogether and it would be reconstructed in the form shown above. Abstracting away from this example, the typing rule for impossibility simply checks that the constraints are indeed unsatisfiable

\[
\forall \ n. \ A \quad \Delta \vdash \text{impossible} :: (x : A) \quad \text{unsat}
\]

There is no operational rule for this scenario since in well-typed configurations the process expression ‘impossible’ is dead code and can never be reached.

**Type Equality.** At the core of an algorithm for type checking is type equality. Informally, two types are equal if they permit exactly the same communication behaviors. This is captured in the recently proposed type equality algorithm [12] that takes two types as input,
We say that we found the algorithm to be sufficient for all our examples.

The rules of operational semantics, applying inversion to the given externally process in the configuration is poised. Conceptually, this means is sending along c. A process poised process is closed, i.e. they do not refer to any free types in process terms are closed, i.e., they do not refer to any free variables. Hence, we use \( V = \cdot \) and \( C = \top \) when typing process \( P \) (premise in proc rule). Similar to proc, the rule \( \text{msg} \) creates a configuration out of a single message (where a message is also represented as a process).

Global Progress. To state progress, we need the notion of a poised process [25]. A process \( \text{proc}(c, P) \) is poised if it is trying to receive a message on \( c \). Dually, a message \( \text{msg}(c, M) \) is poised if it is sending along \( c \). A configuration is poised if every message or process in the configuration is poised. Conceptually, this means that the configuration is trying to communicate externally along one of the channels it uses or provides.

Theorem 1 (Type Safety). For a well-typed configuration \( \Delta_1 \vdash \Sigma \) \( S : \Delta_2 \):

(i) (Preservation) If \( S \rightsquigarrow S' \), then \( \Delta_1 \vdash \Sigma \) \( S' : \Delta_2 \).

(ii) (Progress) Either \( S \) is poised, or \( S \rightsquigarrow S' \).

Proof. The proof of preservation proceeds by case analysis on the rules of operational semantics, applying inversion to the given typing derivation of \( S \), and then assembling a new derivation of \( S' \).

Progress is proved by induction on the right-to-left typing of \( S \) so that either \( S \) is empty (and therefore poised) or \( S = (\mathcal{D}, \text{proc}(c, P)) \) or \( S = (\mathcal{D}, \text{msg}(c, M)) \). By induction hypothesis, \( \mathcal{D} \) can either take a step (and then so can \( S \)), or \( S \) is poised. In the latter case, we analyze the cases for \( P \) and \( M \), applying multiple steps of inversion to show that in each case either \( S \) can take a step or is poised. □

### 4 CONSTRAINT RECONSTRUCTION

The process expressions introduced so far in the language follow simple syntax-directed typing rules. This means they are immediately amenable to be interpreted as an algorithm for type-checking, calling upon a decision procedure where arithmetic entailments and type equalities need to be verified. However, this requires the programmer to maintain a significant number of explicit process constructs pertaining to the refinement layer in their code. Relatively, this hinders reuse: we are unable to provide multiple types to the same program so that it can be used in different contexts.

This section introduces an implicit type system in which the source program never contains the assume and assert expressions, i.e., constructs corresponding to proof constraints. Moreover, impossible branches may be omitted from case expressions. The missing branches and other constructs are restored by a type-directed process of reconstruction.

Interestingly, the nature of Presburger arithmetic makes full reconstruction impossible. For example, the proposition \( \forall n \exists k. (n = 2k \lor n = 2k + 1) \) is true but the witness for \( k \) as a Skolem function of \( n \) (namely \( \lceil n/2 \rceil \)) cannot be expressed in Presburger arithmetic. Since witnesses are critical for establishing correctness of programs, we require that type quantifiers \( \forall n. A \) and \( \exists n. A \) have explicit witnesses in processes and we do not reconstruct them.

In the first phase, a case expression with a missing branch for label \( \ell \) is transformed into a branch \( \ell \Rightarrow \) impossible so that type checking later verifies that the omitted branch is indeed impossible. Then assumes and asserts are inserted according to a reconstruction algorithm described in this section.

Following branch reconstruction, the resulting process expression is checked with the implicit typing judgment \( \Gamma ; \Delta \vdash P : (x : A) \). The implicit system differs from the explicit system in only one way: for the implicit constructs related to constraints (IR, ?L, ?R, ?I), the process expression does not change on application of these.
rules. Selected typing rules are described in Figure 5 and illustrate that expressions $P$ and $Q$ are unchanged in the premise and conclusion. For the remaining rules pertaining to base session types (Section 3.1) and quantifiers ($\exists R, \forall L, \forall R, Y L$), no reconstruction is involved and the implicit rules exactly match the explicit rules.

The implicit rules are sound and complete with respect to the explicit system, since from an implicit typing derivation we can read off the corresponding explicit process expression and vice versa. The rules are also manifestly decidable since the types in the premise are smaller than the conclusion for all the rules presented.

However, the implicit type system is highly nondeterministic. Since the process expressions do not change on the application of implicit rules in Figure 5, they can be applied in many different orders. And each valid order corresponds to a different explicit program, intuitively changing the order in which constraints are sent and received. Thus, an implicit source program may correspond to many different explicit programs. The necessary backtracking would greatly complicate error messages and would also be exponential and severely inefficient.

To solve this problem, we introduce a novel forcing calculus which enforces an order among these implicit constructs. The core idea of this calculus is to follow the structure of each type, but within that assume should be inserted as early as possible, and assert should be inserted as late as possible. This reasoning is sound since the constraints obey a monotonicity property: if a constraint is true at a program point, it will always be true later in the program. Thus, eagerly assuming and lazily asserting constraints is sound: if a constraint can be proved now, it can be proved later. It is also inferential property the constraints obey a monotonicity property since all constraints have been received. We send a constraint lazily, i.e., just before communicating on that channel. We realize this by forcing the channel just before communicating on it. As an example, while sending (or receiving) a label on channel $x$, we force it effectively sending any pending constraints.

The ordered context $\Omega$ imposes an order on the channels on which these invertible rules are applied.

Once all the invertible rules are applied, we reach a stable sequent of the form $\forall V; C; \Delta; \ast \cdot P : (x : A)^{\ast}$, i.e., the ordered context is empty and the provided type $A^{\ast}$ is positive. A stable sequent implies that all constraints have been received. We send a constraint lazily, i.e., just before communicating on that channel. We realize this by forcing the channel just before communicating on it. As an example, while sending (or receiving) a label on channel $x$, we force it effectively sending any pending constraints.

Not all types can be polarized in this manner, particularly types containing alternating proof constraints e.g., $!\phi, ?\psi$. When checking the validity of types before performing reconstruction we reject such types with alternating polarities. We also require that all process declarations contain only structural types at the top-level. Both these restrictions turn out to be mild in practice and can be resolved by introducing additional communications.

Thus, the $? \ast$ operator is positive, while $! \ast$ is negative. The structural types, denoted by $S$ are considered neutral. In the forcing calculus, the invertible rules are applied first.

\[
\begin{align*}
V & : C \land \phi; \Delta^\ast; \Omega + P : (x : A)^\ast \\
\vdash V; C; \Delta^\ast; \Omega + P : (x : [\phi], A)^\ast & \text{IR} \\
V & : C \land \phi; \Delta^\ast; \Omega + P : (x : A^\ast) + P : (z : C^+)^\ast \\
\vdash V; C; \Delta^\ast; \Omega \cdot (x : [?\phi], A^\ast) + P : (z : C^+) & \text{?L}
\end{align*}
\]

If a negative type is encountered in the ordered context, it is considered stable (invertible rules applied) and moved to $\Delta^\ast$.

\[
\begin{align*}
V & : C; \Delta^\ast; (x : A)^\ast; \Omega + P : (z : C^+)^\ast \\
\vdash V; C; \Delta^\ast; \Omega \cdot (x : ?[?\phi], A^\ast) + P : (z : C^+) & \text{move}
\end{align*}
\]

The ordered context $\Omega$ imposes an order on the channels on which these invertible rules are applied.

Finally, if a forced channel has a structural type, we apply the corresponding structural rule and lose the forcing. Again, as an example, we consider the internal choice operator.

\[
\begin{align*}
V & : C \vdash \phi; V; C; \Delta^\ast; \ast \cdot P : (x : A^\ast)^\ast \\
\vdash V; C; \Delta^\ast; \ast \cdot P : (x : ?[?\phi], A^\ast) & \text{IR} \\
V & : C ; ?[?\phi]; C; \Delta^\ast; [(x : !\phi, A^\ast); \ast \cdot P : (x : C^+) ]^\ast \\
\vdash V; C; \Delta^\ast; [(x : !\phi, A^\ast); \ast \cdot P : (x : C^+) ] & \text{?L}
\end{align*}
\]

In either case, applying the structural rule creates a possibly unstable sequent, thereby restarting the inversion phase.

Remarkably, the forcing calculus is sound and complete with respect to the implicit type system, assuming types can be polarized. Since every rule in the forcing calculus is also present in the implicit system, it is trivially sound. Moreover, applying assume eagerly, and assert lazily also turns out to be complete due to the monotonicity property of constraints.

**Theorem 2 (Soundness and Completeness).** For (valid) polarized types $A$ and contexts $\Delta$ we have:

1. If $\forall V; C; \Delta \vdash P : (x : A)$, then $\forall V; C; \Delta ; \ast \cdot P : (x : A)$.
We have implemented a prototype for the language in Standard
validity of types, the implementation reconstructs missing branches
in an abstract syntax. Our Rast implementation, however, uses a
pass of type-checking the reconstructed process expression.

For example, if a process is well-typed in the implicit system, it is well-typed
using the forcing calculus. Once the typing derivation, i.e., ordering
of the typing rules is fixed by the forcing calculus, a unique explicit
program is constructed by applying the explicit typing rules to the
derivation. Thus, if a reconstruction is possible, the forcing calculus
will find it! We use this calculus to reconstruct the explicit program,
which is then typechecked using the explicit typing system.

5 IMPLEMENTATION

We have implemented a prototype for the language in Standard
ML (about 6500 lines of code) available open-source that closely
adheres to the theory presented here. Command line options determine
whether to use explicit or implicit syntax, and the result of
reconstruction can be displayed if desired. We use a straightforward
implementation of Cooper’s algorithm [8] to decide Presburger
arithmetic with two small but significant optimizations. First, we
leverage the fact that we are working over natural numbers rather
than integers which bounds possible solutions from below, and the
second is to eliminate constraints of the form \( x = e \) by substituting \( e \)
for \( x \) in order to reduce the number of variables. After checking the
validity of types, the implementation reconstructs missing branches
and then constraints. Verifying constraints is postponed to the final
pass of type-checking the reconstructed process expression.

Syntax. So far, we have described the types and process terms
in an abstract syntax. Our Rast implementation, however, uses a
concrete syntax. Table 3 describes the abstract syntax of each type
operator, its corresponding concrete type, and the concrete syntax
of the process term of a provider of that type. More details about
the Rast implementation are presented in a system description [11].

A program contains a series of mutually recursive type and
process declarations and definitions.

type \( \nu \{ n \} \ldots \{ nk \} = A \)

dec1 \( f : (x1 : A1) \ldots (xn : An) \bullet (x : A) \)

proc \( x <= f \ x1 \ldots \ x = P \)

The first line is a type definition, where \( \nu \) is the name with index
variables \( n \) and \( A \) is its definition. The second line is a process
declaration, where \( f \) is the process name, \( (x1 : A1) \ldots (xn : An) \)
are the used channels and corresponding types, while the provided
channel is \( x \) of type \( A \). Finally, the last line is a process definition
for the same process \( f \) defined using the process expression \( P \).
We use a hand-written lexer and shift-reduce parser to read an
input file and generate the corresponding abstract syntax tree of
the program. The reason to use a hand-written parser instead of a
parser generator is to anticipate the most common syntax errors
that programmers make and respond with the best possible error
messages.

We describe the results for 9 representative case studies in Ta-
ble 4. We present the module name (Module), the lines of code in
implicit syntax before reconstruction (iLOC), the lines of code after
reconstruction (eLOC), and the time taken by the reconstruction
engine (R (ms)). The experiments were run on an Intel Core i5 2.7
GHz processor with 16 GB 1867 MHz DDR3 memory. We briefly de-
scribe each case study: arithmetic: natural numbers in unary (Sec-
tion 6.1) and binary (Section 6.2) representation; integers: standard
operations on an integer counter; linlam: the linear L-calculus
implementation (Section 6.4); list: lists indexed by their size; primes:
prime sieve of Eratosthenes; segments: partial lists with constant-
work append operation; ternary: natural numbers represented in
balanced 6.3; theorems: circular [13] proofs of simple arithmetic
theorems; tries: a trie data structure to store multisets of binary
numbers 6.6. More details about each module can be found in the
Rast system description [11] or the open-source repository [27].

The two main observations from Table 4 are (i) reconstruction
reduces a significant amount of programmer overhead. The recon-
structed code is almost twice in size compared to the implicit code,
and (ii) reconstruction is very efficient. The forcing calculus
completely eliminates any backtracking in the reconstruction process,
thus converting an exponential task to a linear one. Moreover, the

<table>
<thead>
<tr>
<th>Abstract Type</th>
<th>Concrete Type</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \otimes { l : A \ldots } )</td>
<td>( + { l : A \ldots } )</td>
<td>x,k</td>
</tr>
<tr>
<td>( &amp; { l : A \ldots } )</td>
<td>( &amp; { l : A \ldots } )</td>
<td>case x ( (1 \Rightarrow P \</td>
</tr>
<tr>
<td>( A \otimes B )</td>
<td>( A \times B )</td>
<td>send x w</td>
</tr>
<tr>
<td>( A \rightarrow B )</td>
<td>( A \rightarrow B )</td>
<td>y &lt;= recv x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>close x</td>
</tr>
<tr>
<td>( \exists n. A )</td>
<td>?n. A</td>
<td>send x {e}</td>
</tr>
<tr>
<td>( \forall n. A )</td>
<td>!n. A</td>
<td>{n} &lt;= recv x</td>
</tr>
<tr>
<td>?{n = 0}. A</td>
<td>?{n = 0}. A</td>
<td>assert x {n = 0}</td>
</tr>
<tr>
<td>!{n = 0}. A</td>
<td>!{n = 0}. A</td>
<td>assume x {n = 0}</td>
</tr>
<tr>
<td>( V[\exists] )</td>
<td>( V[\exists 1] \ldots [\exists k] )</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Abstract and Corresponding Concrete Syntax

<table>
<thead>
<tr>
<th>Module</th>
<th>iLOC</th>
<th>eLOC</th>
<th>R (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>69</td>
<td>143</td>
<td>0.353</td>
</tr>
<tr>
<td>integers</td>
<td>90</td>
<td>114</td>
<td>0.200</td>
</tr>
<tr>
<td>linlam</td>
<td>54</td>
<td>67</td>
<td>0.734</td>
</tr>
<tr>
<td>list</td>
<td>244</td>
<td>441</td>
<td>1.534</td>
</tr>
<tr>
<td>primes</td>
<td>90</td>
<td>118</td>
<td>0.196</td>
</tr>
<tr>
<td>segments</td>
<td>48</td>
<td>65</td>
<td>0.239</td>
</tr>
<tr>
<td>ternary</td>
<td>156</td>
<td>235</td>
<td>0.550</td>
</tr>
<tr>
<td>theorems</td>
<td>79</td>
<td>141</td>
<td>0.361</td>
</tr>
<tr>
<td>tries</td>
<td>147</td>
<td>308</td>
<td>1.113</td>
</tr>
</tbody>
</table>

Table 4: Case Studies
forcing calculus does not even need to solve arithmetic constraints to reconstruct the refinement constructs, further improving its efficiency.

6 EXAMPLES
We draw some representative examples from our case studies and study their key properties. In particular, we describe how our arithmetic refinements can help in lightweight verification and complexity analysis of standard concurrent programs.

6.1 Unary Natural Numbers
As a first simple example consider natural numbers in unary form, as usually defined in Peano arithmetic.

\[
\text{type } \text{nat} = \{ \text{zero} : 1, \text{succ} : \text{nat} \}
\]

A process \( P :: \:(c : \text{nat}) \) is required to send a stream of \( \text{succ} \) labels, possibly followed by \( \text{zero} \) and \( \text{close} \). Except for the infinite stream of \( \text{succ} \) labels, every such stream represents a natural number. We can force finiteness and also track the value of the natural number by indexing the type.

\[
\text{type } \text{nat}(n) = \{ \text{zero} : ?(n = 0). 1, \text{succ} : ?(n > 0). \text{nat}(n-1) \}
\]

A process \( P :: \:(c : \text{nat}[i]) \) will now send exactly \( i \) \( \text{succ} \) labels followed by \( \text{zero} \) and \( \text{close} \).

We can use indexing to verify the correctness of some simple processes. We start with "constructor" processes \( \text{zero} \) and \( \text{succ} \) that correspond to the given labels.

\[
\text{proc } x \leftarrow \text{zero} = x.\text{zero} ; \text{close } x \\
\text{proc } x \leftarrow \text{succ}(n) y = x.\text{succ} ; x \leftarrow y
\]

The type of \( \text{succ} \) ensures that it definitely increments the value of the input. Slightly more interesting is a \( \text{half} \) process which is constrained to take an even number of value \( 2 \ast n \) and output a number of value \( n \).

\[
\text{decl } \text{half}(n) : (y : \text{nat}(2\ast n)) \mid \text{(x : nat(n))} \\
\text{proc } x \leftarrow \text{half}(n) y = \\
\text{case } y \Rightarrow \text{wait } y ; x.\text{zero} ; \text{close } x \\
\text{| succ } \Rightarrow \text{case } y \Rightarrow (\% \text{no branch for zero} \\
\text{ succ } \Rightarrow x.\text{succ} ; x \leftarrow \text{half}(n-1) y)
\]

Since \( y : \text{nat}(2 \ast n) \) initially, in the \( \text{succ} \) branch, the type of \( y \) becomes \( \text{nat}(2 \ast n-1) \), thus guaranteeing that the inner \( \text{zero} \) branch is now impossible since \( 2 \ast n-1 \neq 0 \). Reconstruction will fill in the branch for \( \text{zero} \) in the inner case and mark it as impossible, which is then verified by the type checker. Again, type-checking verifies correctness of this implementation.

6.2 Binary Natural Numbers
Representing natural numbers in binary form is somewhat more complicated. We represent a number by a stream of bits \( b_0 \) and \( b_1 \), terminated by \( e \). The least significant bit comes first so that, for example, the number \( 6 = (110)_2 \) is represented by the sequence of labels \( b_0 ; b_1 ; b_1 ; e \).

\[
\text{type } \text{bin} = \{ b_0 : \text{bin}, b_1 : \text{bin}, e : 1 \}
\]

To capture the value of the number, we note that if \( c : \text{bin}[n] \) then after sending \( b_0 \) along \( c \), the channel should now have type \( \text{bin}[n/2] \) (and \( n \) would have to have been even). However, the integer division operator is not directly part of Presburger arithmetic, but can be expressed using an existential quantifier: if \( b_0 \) is sent along \( c : \text{bin}[n] \) then there exists a \( k \) such that \( n = 2 \ast k \) and the remaining stream has type \( \text{bin}[k] \). In addition, we would like to rule out leading zeros (which are actually "trailing" in this representation) and we achieve this by requiring that \( n > 0 \) in the case of \( b_0 \).

\[
\text{type } \text{bin}(n) = +(b_0 : ?(n > 0). 7k. ?(n = 2k). \text{bin}(k), \\
\text{ b1 : 7k. ?(n = 2k+1). \text{bin}(k), \\
\text{ e : ?(n = 0). 1})
\]

Recall that \( 7k \) is concrete syntax for \( \exists k \).

Now the successor process will have to implement the carry from binary addition. That’s done by a recursive call to the successor process on the remaining bit sequence. Again, the types guarantee the correctness of the code.

\[
\text{decl } \text{bzero} : . \mid \text{(x : bin(0))} \\
\text{decl } \text{bsucc}(n) : (x : \text{bin(n)}) \mid \text{(y : bin(n+1))}
\]

\[
\text{proc } x \leftarrow \text{bzero} = x.e ; \text{close } x \\
\text{proc } y \leftarrow \text{bsucc}(n) x = \\
\text{case } x (b_0 \Rightarrow (k) \Rightarrow \text{recv } x ; \\
 y.b1 ; \text{send } y (k) ; y \leftarrow x \\
| b_1 \Rightarrow (k) \Rightarrow \text{recv } x ; \\
 y.b0 ; \text{send } y (k+1) ; y \leftarrow \text{bsucc}(k) x \\
| e \Rightarrow y.b1 ; \text{send } y (0) ; \\
 y.e ; \text{wait } x ; \text{close } y)
\]

Because the quantifiers require explicit witnesses (rather than being reconstructed), this process has to send and receive a suitable \( k \) in each branch. If we know that the witness is computationally irrelevant (currently the case in Rast), no actual \( k \) has to be sent or received when the program executes.

6.3 Balanced Ternary Representation
We can represented integers (not just natural numbers) in balanced ternary form which is defined using three digits: \(-1, 0, \) and \(+1\). If we disallow leading zeros, this representation of integers is unique. Here, we face the difficulty that our index domain consists of natural numbers, not arbitrary integers, so we index each ternary number by two values \( a \) and \( b \) where \( \text{tern}(a, b) \) represents an integer with value \( a - b \). If we don’t bother preventing leading zeros, we get the following type

\[
\text{type } \text{tern}(a)(b) = \\
+(m_1 : ?c. ?d. ?(a+3d+1) = 3c+b). \text{ tern}(c)(d), \\
 z0 : ?c. ?d. ?(a+3d = 3c+b). \text{ tern}(c)(d), \\
 p1 : ?c. ?d. ?(a+3d = 3c+b+1). \text{ tern}(c)(d), \\
 e : ?(a = b). 1
\]

where \( m_1 \) represents digit \(-1\), \( z0 \) represents digit \( 0 \), and \( p1 \) represents digit \(+1\). Looking at the first line, for example, balanced ternary means the digit \(-1 \) \((m_1)\) implies \( a - b = 3 \ast(c - d - 1) \), which we normalize to the constraint \( a + 3 \ast d + 1 = 3 \ast c + b \) to avoid side
conditions on the natural numbers $a$ and $b$. Similar calculations apply for the other digits. The empty sequence $e$ represents the number 0, that is $a - b = 0$.

As an example, we define the \textit{predecessor} process, which is quite simple, except that we have to send and receive the witnesses $c$ and $d$. The carry occurs only in the case of $m1$.

\begin{verbatim}
dcl pred(a;b) : (x : tern(a;b)) |- (y : tern(a;b+1))
proc y <- pred(a;b) x =
  case x ( m1 => (c) <- recv x ; (d) <- recv x ;
             y.p1 ; send y (c) ; send y (d+1) ;
             y <- pred(c;d) x
          | z0 => (c) <- recv x ; (d) <- recv x ;
             y.m1 ; send y (c) ; send y (d) ;
             y <- x
          | p1 => (c) <- recv x ; (d) <- recv x ;
             y.z0 ; send y (c) ; send y (d) ;
             y <- x
          | e => y.m1 ; send y (0) ; send y (0) ;
             y.e ; wait x ; close y
}
\end{verbatim}

Note that once again, type checking verifies the correctness of this implementation because $a - (b + 1) = (a - b) - 1$.

We have the property that $\text{tern}[a,b] = \text{tern}[a+x,b+x]$, and our type equality algorithm [12] recognizes and exploits this equality while type checking. This is different from functional languages with indexed or dependent types, where recursively defined types are usually \textit{nominal}.

### 6.4 Linear $\lambda$-Calculus

An example along entirely different lines is an implementation of the linear $\lambda$-calculus and evaluation (weak head normalization) of terms. It illustrates a number of different techniques from the other examples in paper. We use higher-order abstract syntax, representing linear abstraction in the object language by a process receiving a message corresponding to its argument.

\begin{verbatim}
type exp = +{ lam : exp -o exp,
                     app : exp * exp }

We would like evaluation to return a value (a $\lambda$-abstraction), so we take advantage of the structural nature of types (allowing us to reuse the label \textit{lam}) to define the value type.

type val = +{ lam : exp -o exp }
\end{verbatim}

We have that \textit{val} is a subtype of \textit{exp}, but we actually not to take advantage of this fact (the current implementation of Rast does not support subtyping). We can derive straightforward constructors \textit{apply} for expressions and \textit{lambda} for values (we do not need the corresponding constructor for expressions).

\begin{verbatim}
dcl apply : (e1 : exp) (e2 : exp) |- (e : exp)
proc e <- apply e1 e2 =
  e.app ; send e e1 ; e <- e2

dcl lambda : (f : exp -o exp) |- (v : val)
proc v <- lambda f = v.lam ; v <-> f
\end{verbatim}

As a simple example, here is the representation of a combinator to swap the arguments to a function.

\begin{verbatim}
(* swap = \f. \x. \y. (f y) x *)
\end{verbatim}

\begin{verbatim}
decl swap : |- (e : exp)
proc e <- swap =
  e.lam ; f <- recv e ;
  e.lam ; x <- recv e ;
  e.lam ; y <- recv e ;
  fy <- apply f y ;
  e <- apply fy x
\end{verbatim}

Evaluation is now the following very simple process.

\begin{verbatim}
decl eval : (e : exp) |- (v : val)
proc v <- eval e =
  case e ( lam => v <- lambda e
              | app => e1 <- recv e ; % e = e2
                      v1 <- eval e1 ;
              case v1 ( lam => send v1 e ;
                        v <- eval v1 )
\end{verbatim}

If $e$ sends a \textit{lam} label, we just rebuild the expression as a value. If $e$ sends an \textit{app} label then $e$ represents a linear application $e_1 e_2$ and the continuation has type $\text{exp} \otimes \text{exp}$. This means we receive a channel representing $e_1$ and the continuation (still called $e_2$) behaves like $e_2$. We note this with a comment in the source. We then evaluate $e_1$, which exposes a $\lambda$-expression along the channel $v_1$. We send $e$ along $v_2$, carrying out the reduction via communication. The result of this (still called $v_1$) is evaluated to yield the final value $v$.

This particular call-by-name strategy has practically no parallelism; modeling parallel evaluation requires a small modification of the representation with $\text{lam} : \text{val} \otimes \text{exp}$ and an inclusion of values in expressions. We would now like to prove that the value of a linear $\lambda$-expression is smaller than or equal to the original expression. At the same time we would like to rule out a class of so-called exotic \textit{terms} in the representation, which are possible due to the presence of recursion in the metalanguage. We achieve this by indexing the types \textit{exp} and \textit{val} with their size. For an application, this is easy: the size is one more than the sum of the sizes of the subterms.

\begin{verbatim}
type exp(n) = +( lam : ...
              app : ?n1. ?n2. ?(n = n1+n2+1). exp(n1) * exp(n2) )
\end{verbatim}

The size $n_2 + 1$ of a $\lambda$-expression is one more than the size $n_2$ of its body, but what is that in our higher-order representation? The body is a linear function takes an expression of size $n_1$ and then behaves like an expression of size $n_1 + n_2$. Solving for $n_2$ then gives use the following type definitions and types for the constructor processes.

\begin{verbatim}
type exp(n) =
  +(lam : ?(n > 0). !n1.exp(n1) -o exp(n1+n-1),
      app : ?n1. ?n2. ?(n = n1+n2+1). exp(n1) * exp(n2))
\end{verbatim}

\begin{verbatim}
type val(n) =
  +( lam : ?(n > 0). !n1.exp(n1) -o exp(n1+n-1) )
\end{verbatim}

\begin{verbatim}
decl apply : !n1.exp(n1) (e2 : exp(n2)) |- (e : exp(n1+n2+1))
decl lambda(n2) : (f : !n1.exp(n1) -o exp(n1+n2)) |- (v : val(n2))
\end{verbatim}

The universal quantification over $n_1$ in the type of \textit{lam} is important, because a linear $\lambda$-expression may be applied to an argument of any size. We also cannot predict the size of the result of evaluation, so
we have to use existential quantification: The value of an expression of size \( n \) will have size \( k \) for some \( k \leq n \).

Because witnesses for quantifiers are not reconstructed, the evaluation process has to send and receive suitable sizes.

```haskell
proc v <- eval{n} e =
    case e (lam => send v (n);
              app => \( \{n1\} \) <- recv e;
                      \( \{n2\} \) <- recv e;
                      e1 <- recv e;
                      v1 <- eval(n1) e1;
                      \( k2 \) <- recv v1;
              case v1 (lam => send v1 v2;
                      send v1 e;
                      v2 <- eval(n2+k2-1) v2;
                      \( l \) <- recv v2;
                      send v l;
                      v <- v2))
```

Type-checking now verifies that if evaluation terminates, the resulting value is smaller than the expression (or of equal size). This comes down to deciding certain chains of linear inequalities.

For readers familiar with ergometric types [10], we show how we can bound the number of reductions using an amortized analysis of work. For this basic observation, the code for evaluation follows the previous pattern.

### 6.5 A Binary Counter

A binary counter has an internal value of \( n \) and an interface with two operations: increment (\( \text{inc} \) message) and obtain the value (\( \text{val} \) message). Due to linearity, obtaining the value turns the counter into a number in binary form as introduced in Section 6.2.

```haskell
type ctr{n} = \&{ inc : ctr{n+1},
               val : \text{bin}{n} }
```

We represent a counter as a chain of processes, each holding one bit, where the least significant bit faces the client. \( \text{bit0} \) represents a bit 0, where the whole counter has value \( 2^n \). To prevent leading zeros, we require \( n > 0 \). Similarly \( \text{bit1} \) represents a bit 1, where the whole counter has value \( 2^n + 1 \). Finally, \( \text{empty} \) represents the number 0 (an empty sequence of bits).

```
decl empty : . |- (c : ctr{0})
decl bit0{n|n > 0} : (d : ctr{n}) |- (c : ctr{2*n})
decl bit1{n} : (d : ctr{n}) |- (c : ctr{2*n+1})
```

The implementation of counters is entirely straightforward.

```haskell
proc c <- empty =
    case c (inc => c0 <- empty;
                c <- bit1{0} c0|
              val => c.e ; close c)
proc c <- bit0{n} d =
    case c (inc => c <- bit1{0} c0;
              c.e ; close c)
proc c <- bit1{n} d =
    case c (inc => c <- bit1{n} d;
              c.e ; close c)
```

As a final variation on the theme of the linear \( \lambda \)-calculus we show an implementation suitable for parallel evaluation of terms. Because we would like to evaluate the body of a \( \lambda \)-abstraction in parallel with the argument, we have to pass a channel promising a value to an abstraction. These considerations yield the type

```
type exp = +( app : exp * exp ,
               val : val )
```

Hence, the constructor \( \text{val} \) implements the inclusion of a value in an arbitrary expression. From this basic observation, the code for evaluation follows the previous pattern.

```haskell
decl eval : (e : exp) |- (v : ?k. \{k <= n\}. val{k})
```

Because witnesses for quantifiers are not reconstructed, the evaluation process has to send and receive suitable sizes.
We have two forms of processes: a
We now implement multisets of natural numbers (in binary form).
which is the value of the number inserted into or deleted from the
The code is somewhat repetitive, so we only show the code for
This type requires universal quantification over
One of the key questions is how to maintain linearity in the design
6.6 A Trie for Multisets of Natural Numbers
We now implement multisets of natural numbers (in binary form).
One of the key questions is how to maintain linearity in the design
of the data structure and interface. For example, should we be able
to delete an element from the trie, not knowing a priori if it is
even in the trie? To avoid exceedingly complex types to account for
these situations, the process maintaining a trie offers an interface
with two operations: insert (label ins) and delete (label del). We
index the type trie[n] with the number of elements in the trie,
so inserting an element always increases n by 1. If the element
is already present, we just add 1 to its multiplicity. Deleting an
element actually removes all copies of it and returns its multiplicity
m. If the element is not in the trie, we just return a multiplicity of
m = 0. In either case, the trie contains n − m elements afterwards.

type trie(n) =
&{ins : !k. bin{k} -o trie{n+1},
del : !k. bin{k} -o ?m. ?{m <= n}. bin{m} * trie{n-m}}
This type requires universal quantification over k, (written !k)
which is the value of the number inserted into or deleted from the
trie on each interaction (which is arbitrary).

The basic idea of the implementation is that each bit in the
number x : bin[k] addresses a subtrie: if it is b0 we descend into the
left subtrie, if it is b1 we descent into the right subtrie. If it is
false we have found (or constructed) the node corresponding to x
and we either increase its multiplicity (for insert), or respond with
its multiplicity and set the new multiplicity to zero (for delete).
We have two forms of processes: a leaf with zero elements and
an interior node with n0 + m + n1 elements (where n0 and n1
are the number of elements in the left and right subtrees, and m is the
multiplicity of the number corresponding to this node in the trie).

decl leaf : . |-(t : trie(0))
decl node(n0)(m)(n1) :
|-(t : trie(n0)) (c : ctr(m)) (r : trie(n1))
The code is somewhat repetitive, so we only show the code for
inserting an element into an interior node.

proc t <- node(n0)(m)(n1) l c r =
| case t ( ins => (k) <- recv t ;
| x <- recv t ;
| case x ( b0 =>
| {k'} <- recv x ;
| 1.ins ; send 1 {k'} ; send 1 x ;
| t <- node(n0+1)(m)(n1) l c r
| b1 =>
| {k'} <- recv x ;
| r.ins ; send r {k'} ; send r x
|)

What does type-checking verify in this case? It shows that the
number of elements in the trie increases and decreases as expected
for each insert and delete operation. On the other hand, it does not
verify that the correct multiplicities are incremented or decremented,
which is beyond the reach of the current type system.

7 FURTHER RELATED WORK

Languages with index refinements such as Zenger’s [37], DML [36]
or, more recently, Granule [23] (to name just three of them) were
developed in the realm of functional languages. Bidirectional type
checking was developed in part to tame the complexity of type
checking in DML, which, as a functional language, exhibited an
analogy to natural deduction. As this paper demonstrates, matters
are simpler in some respects when the underlying language is
based on the sequent calculus: type checking is very naturally
bidirectional and therefore robust under refinement. On the other
hand, session types are generally structural rather than nominal,
and that complicates matters to the extent that the underlying type
equality becomes undecidable [12], even if we restrict ourselves
to universal prefix quantifiers. Fortunately, our experience shows
that the algorithm for type equality we proposed in prior work and

Label-dependent session types [29] also integrate session types
indexed by natural numbers. However, they use a fixed schema of
iteration and specific unfolding equality on types, which seems to
apply only in a small number of our examples.

LiquidPi [19] also refines a language of session types, but limits it-
tself to refining basic data types rather than equirecursively defined
session types. As a result, in their language even full type
inference is decidable (under some assumptions on the constraint domain),
but it cannot express many of our motivating examples. A similar re-
finement system designed for dynamic monitoring rather than static
checking has been proposed by Gommerstadt et al. [17, 18]. Also re-
lated is a system by Wu and Xi [35], which only mentions recursive
session types as a possible extension, but does not investigate its
properties. Zhou et al. [38, 39] refine types with arithmetic expres-
sion in the context of multiparty session types. In this recursion-free
setting, they obtain a decidable notion of typing. Another session
typed language with refinements is SePi [3, 14], where refinements
represent capabilities and are therefore quite different from ours.

A step in a different direction is to integrate fully dependent
types, which has also been considered with different aims and
technical realizations [17, 24, 30, 32]. Generally, the theory of type
equality and type checking in these languages has not yet been
developed and, in any case, is likely to be quite different from an
algorithm rooted in the decidability of Presburger arithmetic. Also,
generally speaking, such languages require proof objects to be
communicated (with some specific exceptions [17, 24]).
8 CONCLUSION

In this paper we have shown how to construct a concurrent programming language over arithmetically indexed binary session types. The message-passing semantics of this language is based on the natural polarity of the quantifiers and associated constraints in linear logic, and thereby follows similar proof-theoretically motivated designs and admits an effective bidirectional type-checking algorithm. The language is quite verbose, which is addressed to some extent by our implicit syntax and reconstruction algorithm which is complete for a large class of types. We have probed the expressive power of our language with several examples, all of which easily check in our implementation.

While the general idea of reconstruction easily extends to ergometric types for expressing amortized complexity, our language for temporal types [9] for expressing parallel complexity has so far resisted a similar analysis, in essence because the next-time operator affects multiple channels at once and its proof-theoretic properties are not as uniform as those for the types treated here. We would like to explore if a similar reconstruction algorithm can nevertheless be devised.

Other natural generalizations we intend to pursue are richer constraint domains and mixed linear/nonlinear languages [4], perhaps all the way to adjoint session types [25, 26]. These would open up a whole new class of examples that are difficult or impossible to express in a purely linear language such as Rast is present at.

ACKNOWLEDGMENTS

This article is based on research supported by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514 and Grant No. 1718276.

REFERENCES


