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 λ-calculus, balanced ternary arithmetic, binary counters and tries.

CCS CONCEPTS

• **Theory of computation → Linear logic**; *Type theory*; • **Computing methodologies** → *Concurrent programming languages*.

KEYWORDS

Session types, Concurrency, Linear types, Verification

ACM Reference Format:
Ankush Das and Frank Pfenning. 2020. Verified Linear Session-Typed Concurrent Programming. In *Proceedings of the 22nd Symposium on Principles and Practice of Declarative Programming, PPDP 2020, Bologna, Italy, September 8–10, 2020*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3414080.3414087

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 *ergometric* [10] and *temporal* [9] types to measure and verify (amortized) work and

PPDP '20, September 8–10, 2020, Bologna, Italy

ACM ISBN 978-1-4503-8821-4/20/09…$15.00
https://doi.org/10.1145/3414080.3414087

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

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

This type describes a queue interface supporting insertion and deletion. The *external choice* operator $\&$ dictates that the process providing this data structure accepts either one of two messages: the labels **ins** or **del**. In the case of the label **ins**, it then receives an element of type A denoted by the $\multimap$ operator, and then the type recurses back to queue_A. On receiving a **del** request, the process can respond with one of two labels (**none** or **some**), indicated by the *internal choice* operator $\oplus$. It responds with **none** and then *terminates* (indicated by $\mathbf{1}$) if the queue is empty, or with **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 **none** and **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

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

uses the index refinement n to indicate the size of the queue. In addition, the refined type uses a *type constraint* $?\{\phi\}.\,A$ which can be read as "*there exists a proof of* ϕ". Here, the process providing the queue must (conceptually) send a proof of $n = 0$ after it sends **none**, and a proof of $n > 0$ after it sends **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 ϕ are drawn is Presburger arithmetic and hence decidable, no proof of ϕ 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* ϕ". 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 $\mathrm{queue}_A[n]$, we encounter the constraint $n > 0$ in the **some** branch, so we assume it and then verify that $n - 1 \geq 0$, ensuring the validity of $\mathrm{queue}_A[n-1]$.

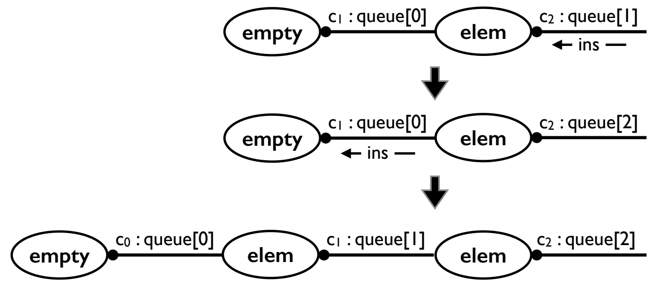

Figure 1: Inserting an element into the queue

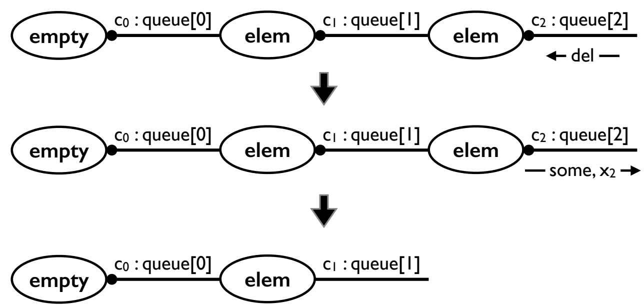

Figure 2: Deleting an element from the queue

Our language design is based on two *key dual principles*: the type $?\{\phi\}.\,A$ corresponds to an *assertion of* ϕ, whereas the type $!\{\phi\}.\,A$ corresponds to an *assumption of* ϕ. Consequently, we introduce dual process terms: *(i)* assert x $\{\phi\}$ to assert constraint ϕ on channel x, and dually, *(ii)* assume x $\{\phi\}$ to assume ϕ 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 provider to receive a natural number. Thus, we introduce *(i)* send x $\{e\}$ to send an arithmetic expression e on channel x and *(ii)* $\{n\} \leftarrow$ 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 *elem* processes, each storing an element of the queue, terminated by an *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 *empty* process that *provides* along c_1 : $\mathrm{queue}_A[0]$ (indicated by $\bullet$ between *empty* and c_1) and does not *use* any channels, and an *elem* process that *uses* c_1 : $\mathrm{queue}_A[0]$, and an element of type A (not shown) and *provides* c_2 : $\mathrm{queue}_A[1]$. The **ins** message is first received on c_2, then passed on to c_1, which spawns a new *empty* process offering on c_0 : $\mathrm{queue}_A[0]$. The original process offering on c_1 then transitions to *elem*. In Figure 2, the **del** message is received on c_2, which replies with the **some** label and an element x_2 stored inside (not shown) the offering *elem* process. The process then terminates by *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:  · ⊢ empty :: (s : queue_A[0])
2:    s ← empty =
3:      case s (
4:        ins ⇒ x ← recv s ;                         % (x : A) ⊢ (s : qu_A[1])
5:              e ← empty ;                  % (x : A), (e : qu_A[0]) ⊢ (s : qu_A[1])
6:              s ← elem[0] x e
7:       | del ⇒ s.none ;                            % · ⊢ (s : ?{0 = 0}. 1)
8:              assert s {0 = 0} ;                          % · ⊢ (s : 1)
9:              close s)
10: (x : A), (t : queue_A[n]) ⊢ elem[n] :: (s : queue_A[n + 1])
11:   s ← elem[n] x, t =
12:     case s (
13:       ins ⇒ y ← recv s ;
14:             t.ins ;
15:             send t y ;   % (x : A), (t : qu_A[n + 1]) ⊢ (s : qu_A[n + 2])
16:             s ← elem[n + 1] x t
17:      | del ⇒ s.some ;
18:             assert s {n + 1 > 0} ;
19:             send s x ;                     % (t : qu_A[n]) ⊢ (s : qu_A[n])
20:             s ↔ 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)

$$\begin{array}{rl} \cdot & \vdash empty :: (s : \text{queue}_A[0]) \\ (x : A)\ (t : \text{queue}_A[n]) & \vdash elem[n] :: (s : \text{queue}_A[n+1]) \end{array}$$

Figure 3 presents the implementation of *empty* and *elem* processes along with their derivations on the right (type $\text{queue}_A[n]$ abbreviated to $\text{qu}_A[n]$). 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 t (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 Δ.

In addition to the type constructors arising from the connectives of intuitionistic linear logic ($\oplus$, $\&$, $\otimes$, $\mathbf{1}$ $\multimap$), we have type names, indexed by a sequence of arithmetic expressions $V[\overline{e}]$, 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.

$$\begin{array}{llcl}
\text{Types} & A, B & ::= & \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \\
 & & \mid & A \otimes B \mid A \multimap B \mid \mathbf{1} \mid V[\overline{e}] \\
 & & \mid & ?\{\phi\}.\, A \mid !\{\phi\}.\, A \mid \exists n.\, A \mid \forall n.\, A \\
\text{Arith. Exps.} & e & ::= & i \mid e + e \mid e - e \mid i \times e \mid n \\
\text{Arith. Props.} & \phi & ::= & e = e \mid e > e \mid \top \mid \bot \mid \phi \wedge \phi \\
 & & \mid & \phi \vee \phi \mid \neg\phi \mid \exists n.\, \phi \mid \forall n.\, \phi \\
\text{Procs} & P, Q & ::= & x.k \;;\; P \mid \mathsf{case}\ x\ (l \Rightarrow P)_{l \in L} \\
 & & \mid & \mathsf{send}\ x\ y \;;\; P \mid y \leftarrow \mathsf{recv}\ x \;;\; P \\
 & & \mid & \mathsf{close}\ x \mid \mathsf{wait}\ x \;;\; P \\
 & & \mid & x \leftrightarrow y \mid x \leftarrow f\ y \;;\; P \\
 & & \mid & \mathsf{assert}\ x\ \{\phi\} \;;\; P \mid \mathsf{assume}\ x\ \{\phi\} \;;\; P \\
 & & \mid & \mathsf{send}\ x\ \{e\} \;;\; P \mid \{n\} \leftarrow \mathsf{recv}\ x \;;\; P
\end{array}$$

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

$$\mathcal{V} \;;\; C \;;\; \Delta \vdash_\Sigma P :: (x : A)$$

where $\mathcal{V}$ are index variables n, C are constraints over these variables expressed as a single proposition, Δ are the linear antecedents $x_i : A_i$, P is a process expression, and $x : A$ is the linear succedent. We propose and maintain that the x_i's and x are all distinct, and that all free index variables in C, Δ, P, and A are contained among

Table 1: Basic session types with operational description

Type	Continuation	Process Term	Continuation	Description
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$c.k \; ; \; P$	P	provider sends label k along c
		$\mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$	Q_k	client receives label k along c
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$\mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}$	P_k	provider receives label k along c
		$c.k \; ; \; Q$	Q	client sends label k along c
$c : A \otimes B$	$c : B$	$\mathsf{send}\ c\ w \; ; \; P$	P	provider sends channel $w : A$ along c
		$y \leftarrow \mathsf{recv}\ c \; ; \; Q_y$	$Q_y[w/y]$	client receives channel $w : A$ along c
$c : A \multimap B$	$c : B$	$y \leftarrow \mathsf{recv}\ c \; ; \; P_y$	$P_y[w/y]$	provider receives channel $w : A$ along c
		$\mathsf{send}\ c\ w \; ; \; Q$	Q	client sends channel $w : A$ along c
$c : \mathbf{1}$	—	$\mathsf{close}\ c$	—	provider sends *close* along c
		$\mathsf{wait}\ c \; ; \; Q$	Q	client receives *close* along c

$\mathcal{V}$. Finally, Σ is a fixed signature containing type and process definitions (explained in Section 3.1) Because it is fixed, we elide it from the presentation of the rules. In addition we write $\mathcal{V} \; ; \; C \vDash \phi$ for semantic entailment (proving ϕ assuming C) in the constraint domain where $\mathcal{V}$ contains all arithmetic variables in C and ϕ. Table 1 overviews the session types 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 $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$ 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 ($\&$, $\oplus$, $\otimes$, $\multimap$ and $\mathbf{1}$). A summary of the corresponding process terms and intuitive explanation for semantics is provided in Table 1.

External Choice. The *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 provider 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 $\mathsf{case}\ x\ (\ell \Rightarrow P)_{\ell \in L}$. Dually, the client must send one of the labels $k \in L$ using the process term $(x.k \; ; \; Q)$ where Q is the continuation.

$$\frac{(\forall \ell \in L) \quad \mathcal{V} \; ; \; C \; ; \; \Delta \vdash P_\ell :: (x : A_\ell)}{\mathcal{V} \; ; \; C \; ; \; \Delta \vdash \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})} \; \&R$$

$$\frac{(k \in L) \quad \mathcal{V} \; ; \; C \; ; \; \Delta, (x : A_k) \vdash Q :: (z : C)}{\mathcal{V} \; ; \; C \; ; \; \Delta, (x : \&\{\ell : A_\ell\}_{\ell \in L}) \vdash (x.k \; ; \; Q) :: (z : C)} \; \&L$$

Communication is asynchronous, so that the client $c.k \; ; \; Q$ 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 $(c.k \; ; \; c \leftrightarrow c')$ (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).

$$(\&S) : \mathsf{proc}(d, c.k \; ; \; Q) \;\mapsto\; \mathsf{msg}(c', c.k \; ; \; c' \leftarrow c), \mathsf{proc}(d, Q[c'/c])$$
$$(\&C) : \mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}),$$
$$\mathsf{msg}(c', c.k \; ; \; c' \leftarrow c) \;\mapsto\; \mathsf{proc}(c', Q_k[c'/c])$$

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

$$\frac{(k \in L) \quad \mathcal{V} \; ; \; C \; ; \; \Delta \vdash P :: (x : A_k)}{\mathcal{V} \; ; \; C \; ; \; \Delta \vdash (x.k \; ; \; P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})} \; \oplus R$$

$$\frac{(\forall \ell \in L) \quad \mathcal{V} \; ; \; C \; ; \; \Delta, (x : A_\ell) \vdash Q_\ell :: (z : C)}{\mathcal{V} \; ; \; C \; ; \; \Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vdash \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)} \; \oplus L$$

This dual constructor reverses the role of the provider and client. The provider $(x.k \; ; \; P)$ of $x : \oplus\{\ell : A_\ell\}_{\ell \in L})$ 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 ℓ with process term Q_ℓ. The rules of operational semantics $(\oplus S, \oplus C)$ are exact dual of $\&S$ and $\&C$ and omitted for brevity.

Channel Passing. The *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 $\mathsf{send}\ x\ y \; ; \; P$ where P is the continuation. Correspondingly, its client must receives a channel using the term $y \leftarrow \mathsf{recv}\ x \; ; \; Q$, binding it to variable y and continuing to execute Q.

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta \vdash P :: (x : B)}{\mathcal{V} \; ; \; C \; ; \; \Delta, (y : A) \vdash (\mathsf{send}\ x\ y \; ; \; P) :: (x : A \otimes B)} \; \otimes R$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta, (y : A), (x : B) \vdash Q :: (z : C)}{\mathcal{V} \; ; \; C \; ; \; \Delta, (x : A \otimes B) \vdash (y \leftarrow \mathsf{recv}\ x \; ; \; Q) :: (z : C)} \; \otimes L$$

Operationally, the provider $\mathsf{send}\ c\ d \; ; \; P$ 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.

$$(\otimes S) : \mathsf{proc}(c, \mathsf{send}\ c\ d\ ;\ P) \ \mapsto\ \mathsf{proc}(c', P[c'/c]),\ \mathsf{msg}(c, \mathsf{send}\ c\ d\ ;\ c \leftarrow c')$$

$$(\otimes C) : \mathsf{msg}(c, \mathsf{send}\ c\ d\ ;\ c \leftarrow c'),\ \mathsf{proc}(e, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \ \mapsto\ \mathsf{proc}(e, Q[c', d/c, x])$$

The dual operator $A \multimap B$ allows the provider to receive a channel of type A and continue to provide type B. The client of $A \multimap B$, on the other hand, sends the channel of type A and continues to use B.

$$\frac{\mathcal{V}\ ;\ C\ ;\ \Delta, (y : A) \vdash P :: (x : B)}{\mathcal{V}\ ;\ C\ ;\ \Delta \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P) :: (x : A \multimap B)}\ {\multimap}R$$

$$\frac{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : B) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : A \multimap B), (y : A) \vdash (\mathsf{send}\ x\ y\ ;\ Q) :: (z : C)}\ {\multimap}L$$

Termination. The type $\mathbf{1}$, 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.

$$\frac{}{\mathcal{V}\ ;\ C\ ;\ \cdot \vdash (\mathsf{close}\ x) :: (x : \mathbf{1})}\ \mathbf{1}R$$

$$\frac{\mathcal{V}\ ;\ C\ ;\ \Delta \vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : \mathbf{1}) \vdash (\mathsf{wait}\ x\ ;\ Q) :: (z : C)}\ \mathbf{1}L$$

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

$$(\mathbf{1}S) : \mathsf{proc}(c, \mathsf{close}\ c) \ \mapsto\ \mathsf{msg}(c, \mathsf{close}\ c)$$
$$(\mathbf{1}C) : \mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \ \mapsto\ \mathsf{proc}(d, Q)$$

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.

$$\frac{}{\mathcal{V}\ ;\ C\ ;\ y : A \vdash (x \leftrightarrow y) :: (x : A)}\ \mathsf{id}$$

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

$$(\mathsf{id}^+ C) : \mathsf{msg}(d, M), \mathsf{proc}(c, c \leftarrow d) \ \mapsto\ \mathsf{msg}(c, [c/d]M)$$
$$(\mathsf{id}^- C) : \mathsf{proc}(c, c \leftarrow d), \mathsf{msg}(e, M(c)) \ \mapsto\ \mathsf{msg}(e, [d/c]M(c))$$

We write $M(c)$ to indicate that c must occur in message M ensuring that M is the sole client of c.

Process Definitions. Process definitions have the form $\Delta \vdash f[\overline{n}] = P :: (x : A)$ where f is the name of the process and P its definition. In addition, $\overline{n}$ is a sequence of arithmetic variables that Δ, P and A can refer to. All definitions are collected in a fixed global signature Σ. For a *well-formed signature*, we require that $\overline{n}\ ;\ \top\ ;\ \Delta \vdash P :: (x : A)$ for every definition, thereby allowing definitions to be mutually recursive. A new instance of a defined process f can be spawned with the expression $x \leftarrow f[\overline{e}]\ \overline{y}\ ;\ Q$ where $\overline{y}$ is a sequence of channels matching the antecedents Δ and $[\overline{e}]$ is a sequence of arithmetic expression matching the variables $[\overline{n}]$. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q.

$$\frac{\overline{y' : B} \vdash f[\overline{n}] = P_f :: (x' : A) \in \Sigma \qquad \Delta' = \overline{(y : B)}[\overline{e}/\overline{n}] \qquad \mathcal{V}\ ;\ C\ ;\ \Delta, (x : A[\overline{e}/\overline{n}]) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, \Delta' \vdash (x \leftarrow f[\overline{e}]\ \overline{y}\ ;\ Q) :: (z : C)}\ \mathsf{def}$$

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

$$(\mathsf{def}C) : \mathsf{proc}(c, x \leftarrow f[\overline{e}]\ \overline{d}\ ;\ Q) \ \mapsto\ \mathsf{proc}(a, P_f[a/x, \overline{d}/\overline{y'}, \overline{e}/\overline{n}]),\ \mathsf{proc}(c, Q[a/x])$$

where $\overline{y' : B} \vdash f[\overline{n}] = P_f :: (x' : A) \in \Sigma$.

Sometimes a process invocation is a tail call, written without a continuation as $x \leftarrow f[\overline{e}]\ \overline{y}$. This is a short-hand for $x' \leftarrow f[\overline{e}]\ \overline{y}\ ;\ x \leftrightarrow x'$ for a fresh variable x', that is, we create a fresh channel and immediately identify it with x.

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 $V[\overline{n}] = A$ in the signature Σ. Here, $\overline{n}$ 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 $V[\overline{e}]$ indexed with arithmetic refinements by $A[\overline{e}/\overline{n}]$ during type checking, and no explicit rules for recursive types are needed.

All types in a signature must be *valid*, formally denoted with the judgment $\mathcal{V}\ ;\ C \vdash A$ *valid*, which requires that all free arithmetic variables of C and A are contained in $\mathcal{V}$, and that for each arithmetic expression e in A we can prove $\mathcal{V}'\ ;\ C' \vdash e : \mathsf{nat}$ 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 ($?\{\phi\}.\, A$, $!\{\phi\}.\, A$) [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 i along channel c and then continue as $A[i/n]$. 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 $\mathcal{V}$. This is represented with the auxiliary judgment $\mathcal{V}\ ;\ C \vdash e : \mathsf{nat}$ (implicitly proving that $e \geq 0$ under constraint C).

Table 2: Refined session types with operational description

Type	Continuation	Process Term	Continuation	Description
$c : \exists n.\, A$	$c : A[i/n]$	$\mathsf{send}\ c\ \{e\}\ ;\ P$	P	provider sends the value i of e along c
		$\{n\} \leftarrow \mathsf{recv}\ c\ ;\ Q$	$Q[i/n]$	client receives number i along c
$c : \forall n.\, A$	$c : A[i/n]$	$\{n\} \leftarrow \mathsf{recv}\ c\ ;\ P$	$P[i/n]$	provider receives number i along c
		$\mathsf{send}\ c\ \{e\}\ ;\ Q$	Q	client sends value i of e along c
$c : ?\{\phi\}.\, A$	$c : A$	$\mathsf{assert}\ c\ \{\phi\}\ ;\ P$	P	provider asserts ϕ on channel c
		$\mathsf{assume}\ c\ \{\phi\}\ ;\ Q$	Q	client assumes ϕ on c
$c : !\{\phi\}.\, A$	$c : A$	$\mathsf{assume}\ c\ \{\phi\}\ ;\ P$	P	provider assumes ϕ on channel c
		$\mathsf{assert}\ c\ \{\phi\}\ ;\ Q$	Q	client asserts ϕ on c

$$\frac{\mathcal{V}\ ;\ C \vdash e : \mathsf{nat} \qquad \mathcal{V}\ ;\ C\ ;\ \Delta \vdash P :: (x : A[e/n])}{\mathcal{V}\ ;\ C\ ;\ \Delta \vdash \mathsf{send}\ x\ \{e\}\ ;\ P :: (x : \exists n.\, A)}\ \exists R$$

$$\frac{\mathcal{V}, n\ ;\ C\ ;\ \Delta, (x : A) \vdash Q :: (z : C) \quad (n \text{ fresh})}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : \exists n.\, A) \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ Q :: (z : C)}\ \exists L$$

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

$(\exists S) : \mathsf{proc}(c, \mathsf{send}\ c\ \{e\}\ ;\ P) \ \mapsto$
$\quad \mathsf{proc}(c', P[c'/c]),\ \mathsf{msg}(c, \mathsf{send}\ c\ \{e\}\ ;\ c \leftrightarrow c')$
$(\exists C) : \mathsf{msg}(c, \mathsf{send}\ c\ \{e\}\ ;\ c \leftrightarrow c'),$
$\quad \mathsf{proc}(d, \{n\} \leftarrow \mathsf{recv}\ c\ ;\ Q) \ \mapsto\ \mathsf{proc}(d, Q[e/n][c'/c])$

The dual type $\forall n.\, A$ reverses the role of the provider and client. The client sends (the value of) an arithmetic expression e which the provider receives and binds to n.

$$\frac{\mathcal{V}, n\ ;\ C\ ;\ \Delta \vdash P_n :: (x : A)}{\mathcal{V}\ ;\ C\ ;\ \Delta \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ P_n :: (x : \forall n.\, A)}\ \forall R$$

$$\frac{\mathcal{V}\ ;\ C \vdash e : \mathsf{nat} \qquad \mathcal{V}\ ;\ \Delta, (x : A[e/n]) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : \forall n.\, A) \vdash \mathsf{send}\ x\ \{e\}\ ;\ Q :: (z : C)}\ \forall L$$

$(\forall S) : \mathsf{proc}(d, \mathsf{send}\ c\ \{e\}\ ;\ P) \ \mapsto$
$\quad \mathsf{msg}(c', \mathsf{send}\ c\ \{e\}\ ;\ c' \leftrightarrow c),\ \mathsf{proc}(d, [c'/c]P)$
$(\forall C) : \mathsf{proc}(d, \{n\} \leftarrow \mathsf{recv}\ c\ ;\ Q),$
$\quad \mathsf{msg}(c', \mathsf{send}\ c\ \{e\}\ ;\ c' \leftrightarrow c) \ \mapsto\ \mathsf{proc}(d, [e/n][c'/c]Q)$

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 $(c : ?\{\phi\}.\, A)$ should send a proof of ϕ 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 ϕ holds, while the client is permitted to assume that ϕ is true. Therefore, and in an analogy with imperative languages, we write $\mathsf{assert}\ c\ \{\phi\}\ ;\ P$ for a process that *asserts* ϕ for channel c and continues with P, while $\mathsf{assume}\ c\ \{\phi\}\ ;\ Q$ *assumes* ϕ and continues with Q.

Thus, the typing rules for this new type constructor are

$$\frac{\mathcal{V}\ ;\ C \vDash \phi \quad \mathcal{V}\ ;\ C\ ;\ \Delta \vdash P :: (x : A)}{\mathcal{V}\ ;\ C\ ;\ \Delta \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ P :: (x : ?\{\phi\}.\, A)}\ ?R$$

$$\frac{\mathcal{V}\ ;\ C \wedge \phi\ ;\ \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : ?\{\phi\}.\, A) \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ ?L$$

Notice how the provider must verify the truth of ϕ given the currently known constraints C (the premise $\mathcal{V}\ ;\ C \vDash \phi$), while the client assumes ϕ by adding it to C.

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

$(?S) : \mathsf{proc}(c, \mathsf{assert}\ c\ \{\phi\}\ ;\ P) \ \mapsto$
$\quad \mathsf{proc}(c', [c'/c]P),\ \mathsf{msg}(c, \mathsf{assert}\ c\ \{\phi\}\ ;\ c \leftrightarrow c')$
$(?C) : \mathsf{msg}(c, \mathsf{assert}\ c\ \{\phi\}\ ;\ c \leftrightarrow c'),$
$\quad \mathsf{proc}(d, \mathsf{assume}\ c\ \{\phi'\}\ ;\ Q) \ \mapsto\ \mathsf{proc}(d, [c'/c]Q)$

In well-typed configurations (which arise from executing well-typed processes) the constraint ϕ 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 ϕ, while the client must verify it. The dual rules are

$$\frac{\mathcal{V}\ ;\ C \wedge \phi\ ;\ \Delta \vdash P :: (x : A)}{\mathcal{V}\ ;\ C\ ;\ \Delta \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ P :: (x : !\{\phi\}.\, A)}\ !R$$

$$\frac{\mathcal{V}\ ;\ C \vDash \phi \quad \mathcal{V}\ ;\ C\ ;\ \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : !\{\phi\}.\, A) \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ !L$$

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 $c : \mathsf{queue}_A[0]$, the type then becomes

$$c : \oplus\{\mathbf{none} : ?\{0{=}0\}.\, \mathbf{1}, \mathbf{some} : ?\{0{>}0\}.\, A \otimes \mathsf{queue}_A[0{-}1]\}$$

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

$\mathsf{case}\ c\ \ (\,\mathbf{none} \Rightarrow \mathsf{assume}\ c\ \{0 = 0\}\ ;\ P_1$
$\qquad\quad \mid \mathbf{some} \Rightarrow \mathsf{assume}\ c\ \{0 > 0\}\ ;\ P_2)$

but what could we write for P_2 in the **some** branch? Intuitively, computation should never get there because the provider can not

$$\frac{}{\Delta \Vdash (\cdot) :: \Delta}\ \mathsf{emp} \qquad \frac{\Delta_1 \Vdash \mathcal{S}_1 :: \Delta_2 \qquad \Delta_2 \Vdash \mathcal{S}_2 :: \Delta_3}{\Delta_1 \Vdash (\mathcal{S}_1, \mathcal{S}_2) :: \Delta_3}\ \mathsf{comp}$$

$$\frac{\cdot\ ;\ \top\ ;\ \Delta \vdash P :: (x : A)}{\Delta \Vdash \mathsf{proc}(x, P) :: (x : A)}\ \mathsf{proc} \qquad \frac{\cdot\ ;\ \top\ ;\ \Delta \vdash M :: (x : A)}{\Delta \Vdash \mathsf{msg}(x, M) :: (x : A)}\ \mathsf{msg}$$

Figure 4: Typing rules for a configuration

assert $0 > 0$. Formally, we use the process expression 'impossible' to indicate that computation can never reach this spot:

case c (**none** $\Rightarrow$ assume c $\{0 = 0\}$; P_1
| **some** $\Rightarrow$ assume c $\{0 > 0\}$; 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

$$\frac{\mathcal{V}\ ;\ C \vDash \bot}{\mathcal{V}\ ;\ C\ ;\ \Delta \vdash \mathsf{impossible} :: (x : A)}\ \mathsf{unsat}$$

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, and attempts to create a *bisimulation* between them. Despite the incompleteness of the algorithm (since the problem is undecidable), we found the algorithm to be sufficient for all our examples.

3.3 Type Safety

The main theorems that establish the deep connection between our refined type system and operational semantics are the usual *type preservation* and *progress*, also referred as *session fidelity* and *deadlock freedom*. At runtime, a program is represented using a set of semantic objects, i.e. processes and messages together defined as a *configuration*.

$$\mathcal{S}\ ::=\ \cdot \mid \mathcal{S}, \mathcal{S}' \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, M)$$

We say that $\mathsf{proc}(c, P)$ (or $\mathsf{msg}(c, M)$) provide channel c. We stipulate that no two distinct semantic objects provide the same channel.

Type Preservation. A key question then is how to *type configurations*? We define a well-typed configuration using the judgment $\Delta_1 \Vdash_\Sigma \mathcal{S} :: \Delta_2$ denoting that configuration $\mathcal{S}$ uses channels Δ_1 and provides channels Δ_2. The rules for typing a configuration are defined in Figure 4. A configuration is always typed w.r.t. a *well-formed signature* Σ, requiring that all *(i)* all type definitions are valid and contractive, and *(ii)* all process definitions are well-typed. Since the signature Σ is fixed, we elide it from the presentation.

The rule emp defines that an empty configuration provides all the channels Δ that it uses. The comp rule composes two configurations $\mathcal{S}_1$ and $\mathcal{S}_2$; $\mathcal{S}_1$ provides channels Δ_2 while $\mathcal{S}_2$ uses channels Δ_2. The rule proc creates a configuration out of a single process.

$$\frac{\mathcal{V}\ ;\ C \vDash \phi \quad \mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash P :: (x : A)}{\mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash P :: (x : ?\{\phi\}.\,A)}\ ?R$$

$$\frac{\mathcal{V}\ ;\ C \wedge \phi\ ;\ \Delta, (x : A)\ {}_i\!\vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : ?\{\phi\}.\,A)\ {}_i\!\vdash Q :: (z : C)}\ ?L$$

$$\frac{\mathcal{V}\ ;\ C \wedge \phi\ ;\ \Delta\ {}_i\!\vdash P :: (x : A)}{\mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash P :: (x : !\{\phi\}.\,A)}\ !R$$

$$\frac{\mathcal{V}\ ;\ C \vDash \phi \quad \mathcal{V}\ ;\ C\ ;\ \Delta, (x : A)\ {}_i\!\vdash Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : !\{\phi\}.\,A)\ {}_i\!\vdash Q :: (z : C)}\ !L$$

Figure 5: Implicit Typing Rules

Configurations only exist at runtime where all arithmetic expressions in process terms are closed, i.e. they do not refer to any free variables. Hence, we use $\mathcal{V} = \cdot$ and $C = \top$ when typing process P (premise in proc rule). Similar to proc, the rule 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 $\mathsf{proc}(c, P)$ is poised if it is trying to receive a message on c. Dually, a message $\mathsf{msg}(c, M)$ 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 $\Delta_1 \Vdash_\Sigma \mathcal{S} :: \Delta_2$:*

(i) *(Preservation) If $\mathcal{S} \mapsto \mathcal{S}'$, then $\Delta_1 \Vdash_\Sigma \mathcal{S}' :: \Delta_2$*
(ii) *(Progress) Either $\mathcal{S}$ is poised, or $\mathcal{S} \mapsto \mathcal{S}'$.*

Proof. The proof of preservation proceeds by case analysis on the rules of operational semantics, applying inversion to the given typing derivation of $\mathcal{S}$, and then assembling a new derivation of $\mathcal{S}'$. Progress is proved by induction on the right-to-left typing of $\mathcal{S}$ so that either $\mathcal{S}$ is empty (and therefore poised) or $\mathcal{S} = (\mathcal{D}, \mathsf{proc}(c, P))$ or $\mathcal{S} = (\mathcal{D}, \mathsf{msg}(c, M))$. By induction hypothesis, $\mathcal{D}$ can either take a step (and then so can $\mathcal{S}$), or $\mathcal{D}$ 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 $\mathcal{S}$ 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 write a significant number of explicit process constructs pertaining to the refinement layer in their code. Relatedly, 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 $\lfloor n/2 \rfloor$) 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 ℓ is transformed into a branch $\ell \Rightarrow \mathsf{impossible}$ 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 $\mathcal{V} \; ; \; C \; ; \; \Delta \;{}_i\!\vdash P :: (x : A)$. The implicit system differs from the explicit system in only one way: for the implicit constructs related to constraints ($!R$, $!L$, $?R$, $?L$), 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 ($\exists R$, $\exists L$, $\forall R$, $\forall L$), 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 complete under the mild assumption that the types can be polarized (explained below). Logically, the $!R$, $?L$ rules are invertible, and are applied eagerly while their dual rules are applied lazily.

This strategy is formally realized in the forcing calculus using the judgment $\mathcal{V} \; ; \; C \; ; \; \Delta \; ; \; \Omega \vdash P :: (x : A)$. The context is split into two: the linear context Δ contains stable propositions on which the invertible left rules have been applied, while the ordered context Ω stores channels on which invertible rules can possibly still be applied to. First, we assign polarities to the type operators with implicit expressions, a notion borrowed from focusing [2] with a similar function here. Type definitions are unfolded in order to determine their polarity, which is always possible since type definitions are contractive. The types that involve communication are called *structural* and represented by S.

$$\begin{array}{lcl}
A^+ & ::= & S \mid ?\{\phi\}.\, A^+ \\
A^- & ::= & S \mid !\{\phi\}.\, A^- \\
A & ::= & A^+ \mid A^- \\
S & ::= & \oplus\{\ell : A\}_{\ell \in L} \mid \&\{\ell : A\}_{\ell \in L} \mid A \otimes A \mid \mathbf{1} \mid A \multimap A \\
 & & \mid \exists n.\, A \mid \forall n.\, A
\end{array}$$

Not all types can be polarized in this manner, particularly types containing alternating proof constraints e.g., $!\{\phi\}.\, ?\{\psi\}.\, A$. 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 ? operator is positive, while ! is negative. The structural types, denoted by S are considered neutral. In the forcing calculus, the invertible rules are applied first.

$$\frac{\mathcal{V} \; ; \; C \land \phi \; ; \; \Delta^- \; ; \; \Omega \vdash P :: (x : A^-)}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \vdash P :: (x : !\{\phi\}.\, A^-)} \; !R$$

$$\frac{\mathcal{V} \; ; \; C \land \phi \; ; \; \Delta^- \; ; \; \Omega \cdot (x : A^+) \vdash P :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \cdot (x : ?\{\phi\}.\, A^+) \vdash P :: (z : C^+)} \; ?L$$

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

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (x : A^-) \; ; \; \Omega \vdash P :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \cdot (x : A^-) \vdash P :: (z : C^+)} \; \mathsf{move}$$

The ordered context Ω 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 $\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash P :: (x : A^+)$, i.e., the ordered context is empty and the provided type A^+ 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.

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash x.k \; ; \; P :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash x.k \; ; \; P :: (x : A^+)} \; \oplus F_R$$

$$\frac{\mathcal{V} \; ; C \; ; \Delta, [x : A^-] \; ; \cdot \vdash \mathsf{case}\; x \; (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C^+)}{\mathcal{V} \; ; C \; ; \Delta, (x : A^-) \; ; \cdot \vdash \mathsf{case}\; x \; (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C^+)} \; \oplus F_L$$

The square brackets $[\cdot]$ indicates that the channel is forced, indicating that a communication is about to happen on it. If there are assert constructs pending on the forced channel, they are applied now.

$$\frac{\mathcal{V}\ ;\ C \vDash \phi \qquad \mathcal{V}\ ;\ C\ ;\ \Delta^-\ ;\ \cdot \vdash P :: [x : A^+]}{\mathcal{V}\ ;\ C\ ;\ \Delta^-\ ;\ \cdot \vdash P :: [x : ?\{\phi\}.\,A^+]}\ ?R$$

$$\frac{\mathcal{V}\ ;\ C \vDash \phi \qquad \mathcal{V}\ ;\ C\ ;\ \Delta^-, [x : A^-]\ ;\ \cdot \vdash P :: (z : C^+)}{\mathcal{V}\ ;\ C\ ;\ \Delta^-, [x : !\{\phi\}.\,A^-]\ ;\ \cdot \vdash P :: (z : C^+)}\ !L$$

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.

$$\frac{(k \in L) \quad \mathcal{V}\ ;\ C\ ;\ \Delta^-\ ;\ \cdot \vdash P :: (x : A_k)}{\mathcal{V}\ ;\ C\ ;\ \Delta^-\ ;\ \cdot \vdash (x.k\ ;\ P) :: [x : \oplus\{\ell : A_\ell\}_{\ell \in L}]}\ \oplus R_k$$

$$\frac{(\forall \ell \in L) \quad \mathcal{V}\ ;\ C\ ;\ \Delta\ ;\ (x : A_\ell) \vdash Q_\ell :: (z : C^+)}{\mathcal{V}\ ;\ C\ ;\ \Delta, [x{:} \oplus \{\ell : A_\ell\}]\ ;\ \cdot \vdash \text{case } x\ (\ell \Rightarrow Q_\ell){::}(z{:}C^+)}\ \oplus L$$

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 Δ we have:*

(1) *If* $\mathcal{V}\ ;\ C\ ;\ \Delta \ {}_i\!\vdash P :: (x : A)$*, then* $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta \vdash P :: (x : A)$.
(2) *If* $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta \vdash P :: (x : A)$*, then* $\mathcal{V}\ ;\ C\ ;\ \Delta \ {}_i\!\vdash P :: (x : A)$.

Proof. Part (1) of Theorem 2 corresponds to soundness. The proof of soundness follows by induction on the implicit typing judgment. Intuitively, soundness follows from the simple observation that every rule in the forcing calculus is also valid in the implicit typing judgment. Theorem 2 part (2) corresponds to completeness whose proof proceeds by induction on the forcing judgment. The proof relies on two *key lemmas*: *(i)* the rules $!R$ and $?L$ are invertible, and *(ii)* if $\mathcal{V}\ ;\ C\ ;\ \Delta^-\ ;\ \Omega \vdash (x : A^+)$ and $\mathcal{V}\ ;\ C \vDash \phi$, then $\mathcal{V}\ ;\ C\ ;\ \Delta^-\ ;\ \Omega \vdash (x : ?\{\phi\}.\,A^+)$, i.e. asserting a constraint ϕ on a channel can be done at any program point where ϕ holds assuming C, and thus, can be delayed. □

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.

Table 3: Abstract and Corresponding Concrete Syntax

Abstract Type	Concrete Type	Concrete Syntax
$\oplus\{l : A, \ldots\}$	`+{l : A, ...}`	`x.k`
$\&\{l : A, \ldots\}$	`&{l : A, ...}`	`case x (l => P | )`
$A \otimes B$	`A * B`	`send x w`
$A \multimap B$	`A -o B`	`y <- recv x`
$\mathbf{1}$	`1`	`close x`
$\exists n.\,A$	`?n. A`	`send x {e}`
$\forall n.\,A$	`!n. A`	`{n} <- recv x`
$?\{n = 0\}.\,A$	`?{n = 0}. A`	`assert x {n = 0}`
$!\{n = 0\}.\,A$	`!{n = 0}. A`	`assume x {n = 0}`
$V[\overline{e}]$	`V{e1}...{ek}`	

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 v{n1}...{nk} = A
decl f : (x1 : A1) ... (xn : An) |- (x : A)
proc x <- f x1 ... xn = P
```

The first line is a *type definition*, where v is the name with index variables n_i and A is its definition. The second line is a *process declaration*, where f is the process name, $(x_1 : A_1) \ldots (x_n : A_n)$ 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 Table 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 describe each case study: **arithmetic**: natural numbers in unary (Section 6.1) and binary (Section 6.2) representation; **integers**: standard

Table 4: Case Studies

Module	iLOC	eLOC	R (ms)
arithmetic	69	143	0.353
integers	90	114	0.200
linlam	54	67	0.734
list	244	441	1.534
primes	90	118	0.196
segments	48	65	0.239
ternary	156	235	0.550
theorems	79	141	0.361
tries	147	308	1.113
Total	**977**	**1632**	**5.280**

operations on an integer counter; **linlam**: the linear λ-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 reconstructed 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 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.

```
type nat = +{ zero : 1, succ : nat }
```

A process $P :: (c : \mathsf{nat})$ is required to send a stream of **succ** labels, possibly followed by **zero** and close. Except for the infinite stream of **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.

```
type nat{n} = +{ zero : ?{n = 0}. 1,
                 succ : ?{n > 0}. nat{n-1} }
```

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

We can use indexing to verify the correctness of some simple processes. We start with "constructor" processes *zero* and *succ* that correspond to the given labels.

```
decl zero : . |- (x : nat{0})
decl succ{n} : (y : nat{n}) |- (x : nat{n+1})

proc x <- zero = x.zero ; close x
proc x <- succ{n} y = x.succ ; x <-> y
```

The type of *succ* ensures that it definitely increments the value of the input. Slightly more interesting is a *half* process which is constrained to take an even number of value $2 * n$ and output a number of value n.

```
decl half{n} : (y : nat{2*n}) |- (x : nat{n})
proc x <- half{n} y =
  case y ( zero => wait y ; x.zero ; close x
         | succ => case y ( % no branch for zero
                     succ => x.succ ; x <- half{n-1} y) )
```

Since $y : \mathsf{nat}[2 * n]$ initially, in the **succ** branch, the type of y becomes $\mathsf{nat}[2*n-1]$, thus guaranteeing that the inner **zero** branch is now impossible since $2 * n - 1 \neq 0$. Reconstruction will fill in the branch for **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 **b0** and **b1**, 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 **b0** ; **b1** ; **b1** ; **e**.

```
type bin = +{ b0 : bin, b1 : bin, e : 1 }
```

To capture the value of the number, we note that if $c : \mathsf{bin}[n]$ then after sending **b0** along c, the channel should now have type $\mathsf{bin}[n/2]$ (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 b0 is sent along $c : \mathsf{bin}[n]$ then there exists a k such that $n = 2 * k$ and the remaining stream has type $\mathsf{bin}[k]$. 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 **b0**.

```
type bin{n} = +{b0 : ?{n > 0}. ?k. ?{n = 2*k}. bin{k},
                b1 :           ?k. ?{n = 2*k+1}. bin{k},
                e : ?{n = 0}. 1}
```

Recall that ?k is concrete syntax for $\exists k$.

Now the successor process will have to implement the carry familiar 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.

```
decl bzero : . |- (x : bin{0})
decl bsucc{n} : (x : bin{n}) |- (y : bin{n+1})

proc x <- bzero = x.e ; close x

proc y <- bsucc{n} x =
  case x ( b0 => {k} <- recv x ;
                 y.b1 ; send y {k} ;
                 y <-> x
         | b1 => {k} <- recv x ;
```

```
               y.b0 ; send y {k+1} ;
               y <- bsucc{k} x
       | e => y.b1 ; send y {0} ;
              y.e ; wait x ; 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 $\mathrm{tern}[a, b]$ represents an integer with value $a - b$. If we don't bother preventing leading zeros, we get the following type

```
type tern{a}{b} =
  +{ m1 : ?c. ?d. ?{a+3*d+1 = 3*c+b}.   tern{c}{d},
     z0 : ?c. ?d. ?{a+3*d   = 3*c+b}.   tern{c}{d},
     p1 : ?c. ?d. ?{a+3*d   = 3*c+b+1}. tern{c}{d},
      e : ?{a = b}. 1 }
```

where **m1** 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 (**m1**) implies $a-b = 3*(c-d)-1$, which we normalize to the constraint $a+3*d+1 = 3*c+b$ 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 *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**.

```
decl 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
         )
```

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

We have the property that $\mathrm{tern}[a, b] = \mathrm{tern}[a + x, b + x]$, 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 *nominal*.

6.4 Linear λ-Calculus

An example along entirely different lines is an implementation of the linear λ-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.

```
type exp = +{ lam : exp -o exp,
              app : exp * exp }
```

We would like evaluation to return a value (a λ-abstraction), so we take advantage of the structural nature of types (allowing us to reuse the label **lam**) to define the value type.

```
type val = +{ lam : exp -o exp }
```

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

```
decl apply : (e1 : exp) (e2 : exp) |- (e : exp)
proc e <- apply e1 e2 =
  e.app ; send e e1 ; e <-> e2

decl lambda : (f : exp -o exp) |- (v : val)
proc v <- lambda f = v.lam ; v <-> f
```

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

```
(* swap = \f. \x. \y. (f y) x *)
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
```

Evaluation is now the following very simple process.

```
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 ) )
```

If e sends a **lam** label, we just rebuild the expression as a value. If e sends an **app** label then e represents a linear application $e_1\, e_2$ and the continuation has type $\mathrm{exp} \otimes \mathrm{exp}$. This means we *receive* a channel representing e_1 and the continuation (still called e) behaves like e_2. We note this with a comment in the source. We then evaluate e_1 which exposes a λ-expression along the channel v_1. We send e along v_1, 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 **lam** : $\mathrm{val} \multimap \mathrm{exp}$ and an inclusion of values in expressions. We would now like to prove that the value of a linear λ-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 terms* in the representation, which are possible due to the presence of recursion in the metalanguage. We achieve this by indexing the types exp and val with their *size*. For an application, this is easy: the size is one more than the sum of the sizes of the subterms.

```
type exp{n} = +{ lam : ...
  app : ?n1. ?n2. ?{n = n1+n2+1}. exp{n1} * exp{n2} }
```

The size $n_2 + 1$ of a λ-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.

```
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}}

type val{n} =
 +{ lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1} }

decl apply{n1}{n2} :
 (e1 : exp{n1}) (e2 : exp{n2}) |- (e : exp{n1+n2+1})
decl lambda{n2} :
 (f : !n1. exp{n1} -o exp{n1+n2}) |- (v : val{n2+1})
```

The universal quantification over n_1 in the type of **lam** is important, because a linear λ-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 $k \leq n$.

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

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

```
proc v <- eval{n} e =
  case e ( lam => send v {n} ;
                  v <- lambda{n-1} e
         | app => {n1} <- recv e ;
                  {n2} <- recv e ;
                  e1 <- recv e ;
                  v1 <- eval{n1} e1 ;
                  {k2} <- recv v1 ;
             case v1 ( lam => send v1 {n2} ;
                              send v1 e ;
                              v2 <- eval{n2+k2-1} v1 ;
                              {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, we assign 1 erg (unit of potential) to each λ-expression. Our cost model is that all operations are free, except the equivalent of a β-reduction which costs 1 erg. Because transfer of potential is reconstructed, the program is very close to the original, size-free program.

```
type exp = +{ lam : |> exp -o exp,
              app : exp * exp }

type val = +{ lam : |> exp -o exp }

decl apply : (e1 : exp) (e2 : exp) |- (e : exp)
proc e <- apply e1 e2 =
  e.app ; send e e1 ; e <-> e2

decl lambda : (f : exp -o exp) |{1}- (v : val)
proc v <- lambda f =
  v.lam ; v <-> f

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 => work ;
                                   send v1 e ; % beta
                                   v <- eval v1 ) )
```

Type-checking here verifies that the reduction of a given expression with n λ-abstractions to a value performs at most $k < n$ β-reductions, with a potential of $n-k$ for further reductions remaining in the value. This means that there are exactly $n - k$ λ-abstractions remaining in the result.

As a final variation on the theme of the linear λ-calculus we show an implementation suitable for *parallel evaluation* of terms. Because we would like to evaluate the body of a λ-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 }

type val = +{ lam : val -o exp }
```

Here, the constructor **val** implements the inclusion of a value in an arbitrary expression. From this basic observation, the code for evaluation follows the previous pattern.

```
decl eval : (e : exp) |- (v : val)
proc v <- eval e =
  case e ( val => v <-> e
         | app => e1 <- recv e ; % e = e2
                  v1 <- eval e1 ;
                  v2 <- eval e ;
                  case v1 ( lam => send v1 v2 ;
                                   v <- eval v1 ) )
```

The key point is that the evaluation of e with destination v_2 (which represents the argument to the function) is started early and proceeds in parallel with the evaluation of e_1 and, once that is finished, the body of the function (v_1, in the final tail call). This version can also be annotated to track size and other information in a manner that is analogous to the more sequential versions of eval.

6.5 A Binary Counter

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

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

We represent a counter as a chain of processes, each holding one bit, where the least significant bit faces the client. *bit0*[n] represents a bit 0, where the whole counter has value $2 * n$. To prevent leading zeros, we require $n > 0$. Similarly *bit1*[n] represent a bit 1, where the whole counter has value $2 * n + 1$. Finally, *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.

```
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{n} d
         | val => c.b0 ; send c {n};
                  d.val ; c <-> d )

proc c <- bit1{n} d =
  case c ( inc => d.inc ;
                  c <- bit0{n+1} d
         | val => c.b1 ; send c {n} ;
                  d.val ; c <-> d )
```

The type checker verifies several properties, including that sending an **inc** message to the counter will indeed increment its value, and that requesting its value with the **val** message will return a binary number with the correct value.

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 **e** 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 $n_0 + m + n_1$ elements (where n_0 and n_1 and the number of elements in the left and right subtries, 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} :
    (l : trie{n0}) (c : ctr{m}) (r : trie{n1})
 |- (t : trie{n0+m+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 ;
                        l.ins ; send l {k'} ; send l x ;
                        t <- node{n0+1}{m}{n1} l c r
                      | b1 =>
                        {k'} <- recv x ;
                        r.ins ; send r {k'} ; send r x ;
                        t <- node{n0}{m}{n1+1} l c r
                      | e =>
                        wait x ;
                        c.inc ;
                        t <- node{n0}{m+1}{n1} l c r )
    | del => ...)
```

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 implemented in Rast [11] is quite robust.

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 itself 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 refinement system designed for *dynamic monitoring* rather than static checking has been proposed by Gommerstadt et al. [17, 18]. Also related 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 expression 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 at present.

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

[1] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. 2020. Deciding the Bisimilarity of Context-Free Session Types. In *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, A. Biere and D. Parker (Eds.). Springer LNCS 12079, Dublin, Ireland, 39–56.

[2] Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2 (1992), 297–347.

[3] Pedro Baltazar, Dimitris Mostrous, and Vasco T. Vasconcelos. 2012. Linearly Refined Session Types. In *International Workshop on Linearity (LINEARITY 2012)*, S. Alves and I. Mackie (Eds.). EPTCS 101, Tallinn, Estonia, 38–49.

[4] Nick Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models. In *Selected Papers from the 8th International Workshop on Computer Science Logic (CLS'94)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer LNCS 933, Kazimierz, Poland, 121–135. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

[5] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, P.Gastin and F.Laroussinie (Eds.). Springer LNCS 6269, Paris, France, 222–236.

[6] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 760 (11 2014).

[7] Iliano Cervesato and Andre Scedrov. 2009. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation* 207, 10 (2009), 1044 – 1077. https://doi.org/10.1016/j.ic.2008.11.006 Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

[8] David C Cooper. 1972. Theorem proving in arithmetic without multiplication. *Machine intelligence* 7, 91-99 (1972), 300.

[9] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel Complexity Analysis with Temporal Session Types. *Proc. ACM Program. Lang.* 2, ICFP, Article 91 (July 2018), 30 pages. https://doi.org/10.1145/3236786

[10] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. ACM, New York, NY, USA, 305–314. https://doi.org/10.1145/3209108.3209146

[11] Ankush Das and Frank Pfenning. 2020. Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 33:1–33:17. https://doi.org/10.4230/LIPIcs.FSCD.2020.33

[12] Ankush Das and Frank Pfenning. 2020. Session Types with Arithmetic Refinements. arXiv:2005.05970 [cs.PL]

[13] Farzaneh Derakhshan and Frank Pfenning. 2019. Circular Proofs as Session-Typed Processes: A Local Validity Condition. arXiv:1908.01909 [cs.LO]

[14] Juliana Franco and Vasco T. Vasconcelos. 2013. A Concurrent Programming Language with Refined Session Types. In *Software Engineering and Formal Methods (SEFM 2013)*, S. Counsell and M. Núñez (Eds.). Springer LNCS 8368, Madrid, Spain, 15–28.

[15] Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (01 Nov 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

[16] J. Y. Girard and Y. Lafont. 1987. Linear logic and lazy computation. In *TAPSOFT '87*, Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–66.

[17] Hannah Gommerstadt. 2019. *Session-Typed Concurrent Contracts*. Ph.D. Dissertation. Carnegie Mellon University. Available as Technical Report CMU-CS-19-119.

[18] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2018. Session-Typed Concurrent Contracts. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 771–798.

[19] Dennis Griffith and Elsa L. Gunter. 2013. LiquidPi: Inferrable Dependent Session Types. In *Proceedings of the NASA Formal Methods Symposium*. Springer LNCS 7871, 186–197.

[20] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

[21] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

[23] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades, III. 2019. Quantitative Program Reasoning with Graded Modal Types. In *International Conference on Functional Programming (ICFP 2019)*. ACM, Berlin, Germany, 110:1–110:30.

[24] Frank Pfenning, Luís Caires, and Bernardo Toninho. 2011. Proof-Carrying Code in a Session-Typed Process Calculus. In *1st International Conference on Certified Programs and Proofs (CPP 2011)*. Springer LNCS 7086, Kenting, Taiwan, 21–36.

[25] Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–22.

[26] Klaas Pruiksma and Frank Pfenning. 2019. A Message-Passing Interpretation of Adjoint Logic. In *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*, F. Martins and D. Orchard (Eds.). EPTCS 291, Prague, Czech Republic, 60–79.

[27] Rast 2020. Rast Language. https://bitbucket.org/fpfenning/rast/src/master/rast/ Version 1.01.

[28] Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-Free Session Types. In *Proceedings of the 21st International Conference on Functional Programming (ICFP 2016)*. ACM, Nara, Japan, 462–475.

[29] Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-Dependent Session Types. In *Proceedings of the Symposium on Programming Languages (POPL 2020)*, L. Birkedal (Ed.). ACM Proceedings on Programming Languages 4, New Orleans, Louisiana, USA, 67:1–67:29.

[30] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent Session Types via Intuitionistic Linear Type Theory. In *Proceedings of the 13th International Conference on Principles and Practice of Declarative Programming (PPDP 2011)*, P.Schneider-Kamp and M.Hanus (Eds.). ACM, Odense, Denmark, 161–172.

[31] Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Typed Processes. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 128–145.

[32] Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Types Processes. In *21st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2018)*, C. Baier and U. Dal Lago (Eds.). Springer LNCS 10803, Thessaloniki, Greece, 128–145.

[33] Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52 – 70. https://doi.org/10.1016/j.ic.2012.05.002

[34] Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012)*. ACM Press, Copenhagen, Denmark, 273–286.

[35] Hanwen Wu and Hongwei Xi. 2017. Dependent Session Types. arXiv:1704.07004 [cs.PL]

[36] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL 1999)*, A. Aiken (Ed.). ACM Press, San Antonio, Texas, USA, 214–227.

[37] Christoph Zenger. 1997. Indexed Types. *Theoretical Computer Science* 187 (1997), 147–165.

[38] Fangyi Zhou. 2019. *Refinement Session Types*. Master's thesis. Imperial College London.

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