Verified Linear Session-Typed Concurrent Programming

Ankush Das  
Carnegie Mellon University  
Pittsburgh, PA, USA  

Frank Pfenning  
Carnegie Mellon University  
Pittsburgh, PA, USA

ABSTRACT

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

CCS CONCEPTS

- Theory of computation \(ightarrow\) Linear logic; Type theory; Computing methodologies \(\rightarrow\) Concurrent programming languages.

KEYWORDS

Session types, Concurrency, Linear types, Verification

ACM Reference Format:

1 INTRODUCTION

Session types [19–21, 32] 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, 33] 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, 27], label-dependent session types [28], and general dependent session types [16, 23, 29, 30]. 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 [26]. 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
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 REFINEMENT SESSION TYPES

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

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

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 \text{ins} or \text{del}. In the case of the label \text{ins}, it then receives an element of type \(A\) denoted by the \(\rightarrow\) operator, and then the type recurses back to \text{queue}_A. On receiving a \text{del} request, the process can respond with one of two labels (\text{none} or \text{some}), indicated by the internal choice operator @. It responds with \text{none} and then terminates (indicated by 1) if the queue is empty, or with \text{some} followed by the element of type \(A\) (expressed with the @ operator) and recurses if the queue is nonempty. However, the simple session type does not express the conditions under which the \text{none} and \text{some} branches must be chosen, which requires tracking the length of the queue in the type.

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

\[
\text{queue}_A[n] = \& \{ \text{ins} : A \rightarrow \text{queue}_A[n + 1], \quad \text{del} : @ \{ \text{none} : ?[n = 0], 1, \quad \text{some} : ?[n > 0]. A \rightarrow \text{queue}_A[n - 1] \} \}
\]

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 \(\phi\).” Here, the process providing the queue must (concretely) send a proof of \(n = 0\) after it sends \text{none}, and a proof of \(n > 0\) after it sends \text{some}. It is therefore constrained in its choice between the two branches based on the value of the index \(n\). Because the the index domain from which the propositions \(\phi\) are drawn is Presburger arithmetic and hence decidable, no proof of \(\phi\) will actually be sent, but we can nevertheless verify the constraint statically. The dual to \(?\phi\). \(A\) is the type constraint \(!\phi\). \(A\) to be interpreted as “for all proofs of \(\phi\).” The refinement type system also supports explicit quantifiers \(\forall n. A\) 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.

\[\text{empty : queue[0]}\]
\[\text{elem : queue[1]}\]
\[\text{empty : queue[2]}\]

**Figure 1: Implementation of queue data structure**

1: \(\cdot \mapsto \text{empty} :: (s : \text{queue}_A[0])\)
2: \(s \leftarrow \text{empty} =\)
3: \(\text{case } s (\)
4: \(\quad \text{ins} \Rightarrow x \leftarrow \text{recv } s ; \quad \% (x : A) \rightarrow (s : \text{queue}_A[1])\)
5: \(\quad e \leftarrow \text{empty} ; \quad \% (x : A), (e : \text{queue}_A[0]) \rightarrow (s : \text{queue}_A[1])\)
6: \(\quad s \leftarrow \text{elem}[0] \mid x\ e\)
7: \(\mid \text{del} \Rightarrow s.\text{none} ; \quad \% \leftarrow (s : ?(n = 0), 1)\)
8: \(\quad \text{assert } s (0 = 0) ; \quad \% \leftarrow (s : 1)\)
9: \(\quad \text{close } s\)
10: \(\quad (x : A), (t : \text{queue}_A[n]) \mapsto \text{elem}[n] :: (s : \text{queue}_A[n + 1])\)
11: \(s \leftarrow \text{elem}[n] x, t =\)
12: \(\text{case } s (\)
13: \(\quad \text{ins} \Rightarrow y \leftarrow \text{recv } s ; \quad t,\text{ins} ;\)
14: \(\quad \text{send } t y ; \quad \% (x : A), (t : \text{queue}_A[n + 1]) \rightarrow (s : \text{queue}_A[n + 2])\)
15: \(\quad s \leftarrow \text{elem}[n + 1] \mid x t\)
16: \(\mid \text{del} \Rightarrow s.\text{some} ;\)
17: \(\quad \text{assert } s \{ n + 1 > 0 \} ;\)
18: \(\quad \text{send } s x ; \quad \% (t : \text{queue}_A[n]) \rightarrow (s : \text{queue}_A[n])\)
19: \(\quad s \leftarrow t\)

**Figure 2: Implementations for the empty and elem processes.**

This includes a static validity check for types to ensure that all index refinements are nonnegative. For example, while checking the validity of \text{queue}_A[n], we encounter the constraint \(n \geq 0\) in the \text{some} branch, so we assume it and then verify that \(n - 1 \geq 0\), ensuring the validity of \text{queue}_A[n - 1].

Our language design is based on two key dual principles: the type \(?\phi\). \(A\) corresponds to an assertion of \(\phi\), whereas the type \(!\phi\). \(A\) corresponds to an assumption of \(\phi\). Consequently, we introduce dual process terms: \((i)\) assert \(x \{ \phi \}\) to assert constraint \(\phi\) on channel \(x\), and dually, \((ii)\) assume \(x \{ \phi \}\) to assume \(\phi\) on \(x\). Following the same principle, we observe that \(\exists n. A\) requires the provider to send a natural number and \(\forall n. A\) mandates the provider to receive a natural number. Thus, we introduce \((i)\) send \(x \{ e \}\) to send an arithmetic expression \(e\) on channel \(x\), and \((ii)\) \(\{n\}\) \(\rightarrow\) 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 (Figure 1) is a sequence of \text{elem} processes, each storing an element of the queue, terminated by an \text{empty} process, representing the empty queue. The \text{empty} process provides along \(c_0 : \text{queue}_A[0]\) (indicated by \(\cdot\)) between \text{empty} and \(c_0\) and does not use any channels. Each \text{elem}[n] process uses \text{queue}_A[n], and an element of type \(A\) (not shown) and \text{provides}_A[n + 1]. In our notation, the process declarations will be written as (used channels on the left and provided channel on the right)

\[\cdot \mapsto \text{empty} :: (s : \text{queue}_A[0])\]
\[x : A, (t : \text{queue}_A[n]) \mapsto \text{elem}[n] :: (s : \text{queue}_A[n + 1])\]

Figure 2 presents the implementation of \text{empty} and \text{elem} processes along with their derivations on the right (type \text{queue}_A[n] abbreviated to \text{qu}_A[n]). Upon receiving the \text{ins} label and element \(x : A\) (line 4), the \text{empty} process spawns a new \text{empty} process (line 5),...
binder 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 $\text{elem}[n + 1]$ (line 16). The type expected by $\text{elem}[n + 1]$ 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.

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 [15]. The key idea is that an intuitionistic linear sequent $A_1, A_2, \ldots, A_n + A$ 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 $P$.

The resulting judgement 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. All these channels must be distinct. We abbreviate the antecedent of the sequent by $\Delta$.

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

We propose and maintain that the empty constructor $\&\{\ell : A_1 : \ell \in L\}$ is a fixed, 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.

3.1 Basic Session Types

External Choice. The external choice type constructor $\&\{\ell : A_1 : \ell \in L\}$ is an $n$-ary labeled generalization of the additive conjunction $A \& B$. Operationally, it requires the provider of $x : \&\{\ell : A_1 : \ell \in L\}$ to branch based on the label $k$ in $L$ it receives from the client and continue to provide type $A_k$. The corresponding process term is written as case $x (\ell \Rightarrow P_\ell) : L$. Dually, the client must send one of the labels $k \in L$ using the process term $(x,k : Q)$ where $Q$ is the continuation.

Communication is asynchronous, so that the client $\langle k \rangle : Q$ sends a message $k$ along channel $c$ and terminates 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 $\ell$ so that the message $k$ is actually represented as $(c,k : \ell \leftrightarrow c')$ (read: send $k$ along channel $c$ and continue along $c'$). When the message $k$ is received along $c$, we select branch $k$ and also substitute the continuation channel $\ell$ for $c$. Rules $\&S$ and $\&C$ below describe the operational behavior of the provider and client respectively ($c'$ fresh).

The internal choice constructor $\@\{\ell : A_1 : \ell \in L\}$ is the dual of external choice requiring the provider to send one of the labels.
\( k \in L \) that the client must branch on.

\[
(k \in L) \quad \forall \ell \in L \quad \forall V \quad C ; \Delta + P :: (x : A_k) \quad \rightarrow \quad \forall V \quad C ; \Delta + P :: (x : A_k) \quad \rightarrow \quad (\forall \ell \in L) \quad \forall V \quad C ; \Delta + P :: (x : A_k) \quad \rightarrow \quad \forall V \quad C ; \Delta + P :: (x : A_k) \quad \rightarrow
\]

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

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

\[
\forall V \quad C ; \Delta + P :: (x : B) \quad \rightarrow \quad \forall V \quad C ; \Delta + P :: (x : B) \quad \rightarrow \quad \forall V \quad C ; \Delta + P :: (x : B) \quad \rightarrow
\]

Operationally, the provider sends \(c \otimes d\) to \(P\) 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 \(c\).

\[
(\otimes) \quad \text{proc}(c, \text{send} c \otimes d) \quad \rightarrow \quad \text{proc}(c', p[c'/c]) \quad \rightarrow \quad \text{msg}(c, \text{send} c \otimes d, c \leftarrow c') \quad \rightarrow \quad \text{proc}(c, \text{send} c \otimes d, c \leftarrow c')
\]

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

\[
\forall V \quad C ; \Delta + (y \leftarrow \text{recv} x) \quad \rightarrow \quad \forall V \quad C ; \Delta + (y \leftarrow \text{recv} x) \quad \rightarrow \quad \forall V \quad C ; \Delta + (y \leftarrow \text{recv} x) \quad \rightarrow
\]

**Process Definitions.** Process definitions have the form \(\Delta \vdash f[\eta] = P :: (x : A)\) where \(f\) is the name of the process and \(P\) its definition. In addition, \(\Pi\) is a sequence of arithmetic variables that \(\Delta, P, A\) and \(\eta\) refer to. All definitions are collected in a fixed global signature \(\Sigma\). For a well-formed signature, we require that \(\Pi \vdash T ; \Delta \vdash P :: (x : A)\) for every definition, thereby allowing definitions to be mutually recursive. A new instance of a defined process \(f\) can be spawned with the expression \(x \leftarrow f[c] \eta \vdash Q\) where \(\eta\) is a sequence of channels matching the antecedents \(\Delta\) and \(\Pi\) is a sequence of arithmetic expression matching the variables \([\eta]\). The newly spawned process will use all variables in \(\eta\) and provide \(x\) to
We now describe quantifiers (\(\exists n. A\), \(\forall n. A\)) and constraints (\(\mathcal{?}(\phi). A\), \(\mathcal{!}(\phi). A\)) [12]. An overview of the types, process expressions, and their operational meaning can be found in Table 2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Continuation</th>
<th>Process Term</th>
<th>Continuation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c : \exists n. A)</td>
<td>(c : A[i/n])</td>
<td>send (c) {(e)} ; (P)</td>
<td>(P)</td>
<td>provider sends the value (i) of (e) along (c)</td>
</tr>
<tr>
<td>(c : \forall n. A)</td>
<td>(c : A[i/n])</td>
<td>({n} \leftarrow \text{recv } c) ; (Q)</td>
<td>(Q[i/n])</td>
<td>client receives number (i) along (c)</td>
</tr>
<tr>
<td>(c : \mathcal{?}(\phi). A)</td>
<td>(c : A)</td>
<td>assert (c) {(\phi)} ; (P)</td>
<td>(P)</td>
<td>provider asserts (\phi) on channel (c)</td>
</tr>
<tr>
<td>(c : \mathcal{!}(\phi). A)</td>
<td>(c : A)</td>
<td>assume (c) {(\phi)} ; (Q)</td>
<td>(Q)</td>
<td>client assumes (\phi) on (c)</td>
</tr>
</tbody>
</table>

The declaration of variables of type \(A\) requires \(A\) to be valid in the signature \(\Sigma\). The declaration of 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 intutionistic logic. For this purpose we allow (possibly mutually recursive) type definitions \(\mathcal{V}[\Gamma] = A\) in the signature \(\Sigma\). Here, \([\Gamma]\) denotes a sequence of arithmetic variables. Again, for a well-formed signature, we require \(A\) to be contractive [14] meaning \(A\) should not itself be a type name. Our type definitions are equirecursive so we can silently replace type names \(\mathcal{V}[\Gamma]\) indexed with arithmetic refinements by \(A[\Gamma/\Gamma]\) during type checking, and no explicit rules for recursive types are needed.

All types in a signature must be valid, formally defined 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 \(\mathcal{V} : C \vdash e : \text{nat}\) for the constraints \(C\) known at the occurrence of \(e\) (implicitly proving that \(e \geq 0\)).

### 3.2 The Refinement Layer

We now describe quantifiers (\(\exists n. A\), \(\forall n. A\)) and constraints (\(\mathcal{?}(\phi). A\), \(\mathcal{!}(\phi). A\)) [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 \(\mathcal{V} : C \vdash e : \text{nat}\) (implicitly proving that \(e \geq 0\) under constraint \(C\)).

### 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 \((c : \mathcal{?}(\phi). A)\) should send a proof of \(\phi\) along \(c\) and the client should receive such a proof. However, since the index domain is decidable and future computation cannot depend on the form of the proof (what is known in type theory as proof irrelevance) such messages are not actually exchanged. Instead, it is the provider’s responsibility to ensure that \(\phi\) holds, while the client is permitted to assume that \(\phi\) is true. Therefore, and in an analogy with imperative languages, we write assert \(c\) \{\(\phi\)\} ; \(P\) for a process that asserts \(\phi\) on channel \(c\) and continues with \(P\), while assume \(c\) \{\(\phi\)\} ; \(Q\) assumes \(\phi\) and continues with \(Q\).
Thus, the typing rules for this new type constructor are
\[
\frac{\nu : C \vdash \phi \quad \nu : C ; \Delta \vdash P :: (x : A)}{\nu : C ; \Delta \vdash \text{assert } x \{\phi\} :: P :: (x : ?\{\phi\}. A)} \quad \text{?R}
\]
\[
\frac{\nu : C \vdash \phi \quad \nu : C ; \Delta, (x : A) \vdash Q :: (z : C)}{\nu : C ; \Delta, (x : ?\{\phi\}. A) \vdash \text{assume } x \{\phi\} :: Q :: (z : C)} \quad \text{?L}
\]
Notice how the provider must verify the truth of \(\phi\) given the currently known constraints \(C\) (the premise \(\nu : C \vdash \phi\)), while the client assumes \(\phi\) by adding it to \(C\). In well-typed configurations (which arise from executing well-typed processes) the constraint \(\phi\) in these rules will always be closed and true so there is no need to check this explicitly.

The dual operator \(!\{\phi\}\). \(A\) reverses the role of provider and client. The provider of \(x : !\{\phi\}. A\) may assume the truth of \(\phi\), while the client must verify it. The dual rules are
\[
\frac{\nu : C \& \phi \quad \Delta \vdash P :: (x : A)}{\nu : C ; \Delta \vdash \text{assume } x \{\phi\} :: P :: (x : ?\{\phi\}. A)} \quad !\text{R}
\]
\[
\frac{\nu : C \vdash \phi \quad \nu : C ; \Delta, (x : A) \vdash Q :: (z : C)}{\nu : C ; \Delta, (x : ?\{\phi\}. A) \vdash \text{assume } x \{\phi\} :: Q :: (z : C)} \quad !\text{L}
\]

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 \(c : \text{queue}_{A[0]}\), the type then becomes
\[
c : \oplus\{\text{none} : \exists\{0=0\}\}. \text{\textbf{1}}, \text{\textbf{some}} : \exists\{0>0\}\}. A \& \text{queue}_{A[0-1]}
\]
The client would have to branch on the label received and then assume the constraint asserted by the provider
case \(c\) (\text{\textbf{none}} \Rightarrow \text{assume } c \{0 = 0\} ; P_1\
| \text{\textbf{some}} \Rightarrow \text{assume } c \{0 > 0\} ; P_2)
but what could we write for \(P_2\) in the \text{\textbf{some}} branch? Intuitively, computation should never get there because the provider can not assert \(0 > 0\). Formally, we use the process expression ‘impossible’ to indicate that computation can never reach this spot:
case \(c\) (\text{\textbf{none}} \Rightarrow \text{assume } c \{0 = 0\} ; P_1\
| \text{\textbf{some}} \Rightarrow \text{assume } c \{0 > 0\} ; \text{impossible})

In implicit syntax (see Section 4) we could omit the \text{\textbf{some}} branch altogether and it would be handled in the form shown above. Abstracting away from this example, the typing rule for impossibility simply checks that the constraints are indeed unsatisfiable
\[
\frac{\varepsilon : C \vdash \bot}{\nu : C ; \Delta \vdash \text{impossible} :: (x : A)} \quad \text{unsat}
\]

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

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

![Figure 3: Typing rules for a configuration](image-url)

### 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.

\[
S \::=\cdot | S, S' | \text{proc}(c, P) | \text{msg}(c, M)
\]

We say that \(\text{proc}(c, P)\) (or \(\text{msg}(c, M)\)) 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 \(\Delta_1 \models_\Sigma S :: \Delta_2\) denoting that configuration \(S\) uses channels \(\Delta_1\) and provides channels \(\Delta_2\). The rules for typing a configuration are defined in Figure 3. A configuration is always typed w.r.t. a well-formed signature \(\Sigma\), requiring that all (i) all type definitions are valid and contractive, and (ii) all process definitions are well-typed. Since the signature \(\Sigma\) is fixed, we elide it from the presentation.

The rule \(\text{emp}\) defines that an empty configuration provides all the channels \(\Delta\) that it uses. The comp rule composes two configurations \(S_1, S_2\); \(S_1\) provides channels \(\Delta_1\) while \(S_2\) uses channels \(\Delta_2\). The rule \(\text{proc}\) creates a configuration out of a single process. 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 \(\nu = \cdot\) and \(\nu = \top\) when typing process \(P\) (premise in proc rule). Similar to \(\text{proc}\), the rule \(\text{msg}\) creates a configuration out of a single message.

**Global Progress.** To state progress, we need the notion of a poisoned process [24]. A process \(\text{proc}(c, P)\) is poisoned if it is trying to receive a message on \(c\). Dually, a message \(\text{msg}(c, M)\) is poisoned if it is sending along \(c\). A configuration is poisoned if every message or process in the configuration is poisoned. 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 \(\Delta_1 \models_\Sigma S :: \Delta_2\):

(i) (Preservation) If \(S \rightarrow S'\), then \(\Delta_1 \models_\Sigma S' :: \Delta_2\)

(ii) (Progress) Either \(S\) is poisoned, or \(S \rightarrow S'\).

**Proof.** The proof of preservation proceeds by case analysis on the rules of operational semantics, applying inversion to the given typing derivation of \(S\), and then assembling a new derivation of \(S'\). Progress is proved by induction on the right-to-left typing of \(S\) so
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 4, 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 \( \text{\textbf{V}} \); \( \text{C} \); \( \text{\Delta} \) \( \vdash P \) :: \( x : A \) (explained below). The necessary backtracking would greatly complicate error messages and would also be exponential and severely inefficient.

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 \( \ell \) is transformed into a branch \( \ell \Rightarrow \) impossible so that type checking later verifies that the omitted branch is indeed impossible. Then assumes and asserts are inserted according to a reconstruction algorithm described in this section.

Following branch reconstruction, the resulting process expression is checked with the implicit type judgment \( \text{\textbf{V}} \); \( \text{C} \); \( \text{\Delta} \) \( \vdash P \) :: \( x : A \). 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 4 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 (\( \forall R, \exists L, \forall R, \forall L \)), no reconstruction is involved and the implicit rules exactly match the explicit rules.
If a negative type is encountered in the ordered context, it is considered stable (invertible rules applied) and moved to \( \Delta^- \).

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

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

Once all the invertible rules are applied, we reach a stable sequent of the form \( V; C; \Delta^-; \vdash P : (x : A^+) \), 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.

\[
\begin{align*}
V; C &; \Delta^-; \vdash x \cdot k \vdash P : [x : A^+] \quad \Delta^- \vdash P \vdash (x : A^+) \\
V; C &; \Delta^-; \vdash x \cdot k \vdash P : (x : A^+) \\
\end{align*}
\]

Once all the invertible rules are applied, we reach a stable sequent of the form \( V; C; \Delta^-; \vdash P ; (x : A^+) \), 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, we consider the internal choice operator.

\[
V; C; \Delta^-; \vdash \text{case } x (\ell \Rightarrow Q_{\ell} \epsilon L) \vdash P : (z : C^+) \quad \Delta^- \vdash P \vdash (z : C^+) \\
\]

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.

\[
\begin{align*}
V; C; \Delta^-; \vdash P : (x : A^+) &; \vdash \phi \quad \Delta^-; \vdash P : (x : A^+) \quad \Delta^-; \vdash P \vdash (x : A^+) \\
V; C; \Delta^-; \vdash P : (x : A^+) &; \vdash \phi \quad \Delta^-; \vdash P : (x : A^+) \\
\end{align*}
\]

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

\[
\begin{align*}
V; C; \Delta^-; \vdash P : (x : A_k) &; \forall (k \in L) \quad \forall (k \in L) \quad \forall (k \in L) \quad \forall (k \in L) \\
\forall (\ell \in L) &; \forall (\ell \in L) \quad \forall (\ell \in L) \quad \forall (\ell \in L) \\
\end{align*}
\]

In either case, applying the structural rules 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 eagerly, and assert lazily also turns out to be complete due to the monotonic property of constraints.

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

1. If \( V; C; \Delta \vdash P : (x : A) \), then \( V; C; \Delta \vdash P : (x : A) \).
2. If \( V; C; \Delta \vdash P : (x : A) \), then \( V; C; \Delta, \vdash P : (x : A) \).

**Proof.** Proof of 1. follows by induction on the implicit typing judgment. Proof of 2. follows by induction on the forcing judgment.

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. One takes advantage of the fact that we are working over natural numbers rather than integers which bounds possible solutions from below, and the other 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>
<thead>
<tr>
<th>Abstract Type</th>
<th>Concrete Type</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>@I : A, ...</td>
<td>+(1 : A, ... )</td>
<td>x.k</td>
</tr>
<tr>
<td>&amp;@I : A, ...</td>
<td>&amp;(1 : A, ... )</td>
<td>case x ((1 \Rightarrow P)</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>A &amp; B</td>
<td>send x w</td>
</tr>
<tr>
<td>A \rightarrow B</td>
<td>A \rightarrow B</td>
<td>y &lt;= recv x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>close x</td>
</tr>
<tr>
<td>?n, A</td>
<td>?n. A</td>
<td>send x (e)</td>
</tr>
<tr>
<td>\forall n. A</td>
<td>\forall n. A</td>
<td>(n &lt;= recv x</td>
</tr>
<tr>
<td>![n = 0]. A</td>
<td>![n = 0]. A</td>
<td>assert x ((n = 0)</td>
</tr>
<tr>
<td>![n = 0]. A</td>
<td>![n = 0]. A</td>
<td>assume x ((n = 0)</td>
</tr>
<tr>
<td>V[e]</td>
<td>V(e1 \ldots (ek)</td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Concrete Type</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>type (v[n] = A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decl f : (x1 : A1) ... (xn : An)</td>
<td>( \sim (x : A)</td>
<td></td>
</tr>
<tr>
<td>proc x &lt;= f x1 ... xn = P</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first line is a type definition, where \( v \) is the name with index variable \( a \) and \( A \) is its definition. The second line is a process declaration, where \( f \) is the process name, \( (x1 : A1) ... (xn : An) \) are the used channels and corresponding types, while the provided channel is \( x \) of type \( A \). Finally, the last line is a process definition for the same process \( f \) defined using the process expression \( P \). We use a hand-written lexer and shift-reduce parser to read an input file and
We implement a variety of examples in the Rast language, and which is then verified by the type checker. Again, type-checking a process P can be complicated. We represent a number by a stream of bits which is defined using three digits:

\[
\text{type bin} = +\{ \text{b0 : bin}, \text{b1 : bin}, \text{e : 1} \}
\]

To capture the value of the number, we note that if \( c : \text{bin}[n] \) then after sending \text{b0} along \( c \), the channel should now have type \text{bin}[n/2] (and \( n \) would have to have been even). However, the integer division operator is not directly part of Presburger arithmetic, but can be expressed using an existential quantifier: if \( \text{b0} \) is sent along \( c : \text{bin}[n] \) then there exists a \( k \) such that \( n = 2 \ast k \) and the remaining stream has type \text{bin}[k]. In addition, we would like to rule out leading zeros (which are actually “trailing” in this representation) and we achieve this by requiring that \( n > 0 \) in the case of \( \text{b0} \).

\[
\text{type bin}(n) = +\{ \text{b0 : ?}(n > 0). \text{?k}. \text{?}{n = 2\ast k}. \text{bin}{k},
\text{b1 : ?k}. \text{?}{n = 2\ast k+1}. \text{bin}{k},
\text{e : ?}(n = 0). 1 \}
\]

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.

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

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.

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

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

\[
\text{type tern(a)(b) =}
+\{ m1 : ?c. ?d. (?a+3*d+1 = 3*c+b). \text{tern}(c)(d),
\text{z0 : ?c. ?d. (?a+3*d = 3*c+b). \text{tern}(c)(d),
\text{p1 : ?c. ?d. (?a+3*d = 3*c+b+1). \text{tern}(c)(d),
e : ?(a = b). 1 \}
\]

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 \( a - b = 3 \ast (c-d) - 1 \), which we normalize to the constraint \( a + 3 \ast d + 1 = 3 \ast c + b \) to avoid side

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

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

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

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

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

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

\[
\text{decl zero : . |} (x : \text{nat}(0))
\text{decl succ(n) : (y : \text{nat}(n)) |} (x : \text{nat}(n+1))
\]

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

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

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

6.2 Binary Natural Numbers
Representing natural numbers in binary form is somewhat more complicated. We represent a number by a stream of bits \( \text{b0} \) and \( \text{b1} \), 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 \( \text{b0} : \text{b1} ; \text{b1} ; \text{e} \).

\[
\text{type bin = +\{ b0 : bin, b1 : bin, e : 1 \}}
\]
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 $m_1$.

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

### 6.4 Linear $\lambda$-Calculus

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

We would now like to prove that the value of a linear $\lambda$-expression is one more than the size of its body.

```plaintext
type exp = (+{ lam : exp -o exp, 
  app : ?n1. ?n2. ?{n = n1+n2+1}. exp{n1} * exp{n2}}
)

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

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

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

```plaintext
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.

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

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

\[
\text{decl } \text{eval}(n) : (e : \text{exp}(n)) \vdash (v : ?k. ?(k <= n). \text{val}(k))
\]

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

\[
\text{proc } v \leftarrow \text{eval}(n) e =
\]

\[
\begin{align*}
\text{case } e (\text{lamb} & \Rightarrow v \leftarrow \text{lambda}(n-1) e) \\
\text{app} & \Rightarrow \{n1\} \leftarrow \text{recv} e \; ; \\
\{n2\} & \leftarrow \text{recv} e \; ; \\
e1 & \leftarrow \text{recv} e \; ; \\
\{k2\} & \leftarrow \text{recv} v1 \; ; \\
\text{case } v1 (\text{lamb} & \Rightarrow \text{send} v1 \{n2\} \; ; \\
\text{send} v1 e ; \\
v2 & \leftarrow \text{eval}(n2+k2-1) v1 \; ; \\
\{l\} & \leftarrow \text{recv} v2 \; ; \\
\text{send } v \{l\} \; ; v & \leftarrow v2)
\end{align*}
\]

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 chain of linear inequalities.

We can also bound the number of reductions using an amortized analysis of work. For this, we assign 1 erg (unit of potential) to the original, size-free program, adding the potential in the type and transfer of potential is reconstructed, the program is very close to \( \beta \)-reduction which costs 1 erg. Because witnesses for quantifiers are not reconstructed, the evaluation process has to send and receive suitable sizes.

\[
\text{proc } v \leftarrow \text{eval}(n) e =
\]

\[
\begin{align*}
\text{case } e (\text{lamb} & \Rightarrow v \leftarrow \text{lambda}(n-1) e) \\
\text{app} & \Rightarrow \{n1\} \leftarrow \text{recv} e \; ; \\
\{n2\} & \leftarrow \text{recv} e \; ; \\
e1 & \leftarrow \text{recv} e \; ; \\
\{k2\} & \leftarrow \text{recv} v1 \; ; \\
\text{case } v1 (\text{lamb} & \Rightarrow \text{send} v1 \{n2\} \; ; \\
\text{send } v1 e ; \\
v2 & \leftarrow \text{eval}(n2+k2-1) v1 \; ; \\
\{l\} & \leftarrow \text{recv} v2 \; ; \\
\text{send } v \{l\} \; ; v & \leftarrow v2)
\end{align*}
\]

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 chain of linear inequalities.

We can also bound the number of reductions using an amortized analysis of work. For this, we assign 1 erg (unit of potential) to each \( \lambda \)-expression. Our cost model is that all operations are free, except the equivalent of a \( \beta \)-reduction which costs 1 erg. Because transfer of potential is reconstructed, the program is very close to the original, size-free program, adding the potential in the type and one occurrence of work in the evaluator.

\[
\text{type } \text{exp} = +\{ \text{lamb} : \{> \text{exp} \rightarrow \text{exp} \}, \\
\text{app} : \text{exp} \times \text{exp} \}
\]

\[
\text{type } \text{val} = +\{ \text{lamb} : \{> \text{exp} \rightarrow \text{exp} \}
\]

\[
\text{decl } \text{apply} : (e1 : \text{exp}) (e2 : \text{exp}) \vdash (e : \text{exp})
\]

\[
\text{proc } e \leftarrow \text{apply} e1 e2 =
\]

\[
\begin{align*}
e.\text{app} \; ; \text{send } e1 e2 \; ; e & \leftarrow e2
\end{align*}
\]

\[
\text{decl } \text{lambda} : (f : \text{exp} - \text{exp}) \vdash \{l\} (v : \text{val})
\]

\[
\text{proc } v \leftarrow \text{lambda} f =
\]

\[
\begin{align*}v.\text{lamb} \; ; v & \leftarrow f
\end{align*}
\]

\[
\text{decl } \text{eval} : (e : \text{exp}) \vdash (v : \text{val})
\]

\[
\text{proc } v \leftarrow \text{eval} e =
\]

\[
\begin{align*}
\text{case } e (\text{lamb} & \Rightarrow v \leftarrow \text{lambda} e) \\
\text{app} & \Rightarrow \{e1\} \leftarrow \text{recv} e \; ; \% e = e2 \\
v1 & \leftarrow \text{eval} e1 \; ; \\
\text{case } v1 (\text{lamb} & \Rightarrow \text{work} \; ; \\
\text{send } v1 e \; ; \% \beta \text{a} \\
v & \leftarrow v1)
\end{align*}
\]

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

### 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.

\[
\text{type } \text{ctr}(n) = \&\{ \text{inc} : \text{ctr}(n+1), \\
\text{val} : \text{bin}(n) \}
\]

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

\[
\text{decl } \text{empty} : . \vdash (c : \text{ctr}(0))
\]

\[
\text{decl } \text{bit0}(n | n > 0) : (d : \text{ctr}(n)) \vdash (c : \text{ctr}(2n))
\]

\[
\text{decl } \text{bit1}(n) : (d : \text{ctr}(n)) \vdash (c : \text{ctr}(2n+1))
\]

The implementation of counters is entirely straightforward.

\[
\text{proc } c \leftarrow \text{empty} =
\]

\[
\begin{align*}
\text{case } c (\text{inc} & \Rightarrow c0 \leftarrow \text{empty} \; ; \\
\% \leftarrow c.\text{e} \; ; \text{close } c)
\end{align*}
\]

\[
\begin{align*}
\text{proc } c \leftarrow \text{bit0}(n) d =
\end{align*}
\]

\[
\begin{align*}
\text{case } c (\text{inc} & \Rightarrow c \leftarrow \text{bit1}(n) d \; ; \\
\% \leftarrow c.\text{e} \; ; \text{send } c \{n\} ; \\
\text{d.}\text{val} ; c & \leftarrow d)
\end{align*}
\]

\[
\begin{align*}
\text{proc } c \leftarrow \text{bit1}(n) d =
\end{align*}
\]

\[
\begin{align*}
\text{case } c (\text{inc} & \Rightarrow d.\text{inc} \; ; \\
\% \leftarrow d.\text{e} \; ; \\
\text{c} & \leftarrow \text{bit0}(n+1) d \; ; \\
\% \leftarrow d.\text{val} \; ; \text{send } c \{n\} ; \\
\text{d.}\text{val} ; c & \leftarrow d)
\end{align*}
\]

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 \( \text{trie}[n] \) 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.

\[
\text{type } \text{trie}(n) =
\]

\[
\{\text{ins} : !k. \text{bin}(k) -o \text{trie}(n+1), \\
\text{del} : !k. \text{bin}(k) -o ?m. ?(m <= n). \text{bin}(m) \times \text{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).

\[
\text{proc } t \leftarrow \text{node}(n_0)(m)(n_1) \mid l \ c \ r =
\begin{align*}
\text{case } t (\text{ins} => & \{ k \} \leftarrow \text{recv} \ t ; \\
& x \leftarrow \text{recv} \ t ; \\
& \text{case } x (b0 => & \{ k' \} \leftarrow \text{recv} \ x ; \\
& \ 1.\text{ins} ; \text{send} \ l \ (k') ; \text{send} \ l \ x ; \\
& \ t \leftarrow \text{node}(n_0+1)(m)(n_1) \mid l \ c \ r \\
& b1 => & \{ k' \} \leftarrow \text{recv} \ x ; \\
& \ r.\text{ins} ; \text{send} \ r \ (k') ; \text{send} \ r \ x ; \\
& \ t \leftarrow \text{node}(n_0)(m+1)(n_1+1) \mid l \ c \ r \\
& e => & \text{wait} \ x ; \\
& \ c.\text{inc} ; \\
& \ t \leftarrow \text{node}(n_0)(m+1)(n_1) \mid l \ c \ r ) \\
& \text{del} => \ldots)
\end{align*}
\]

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 [36], DML [35] or, more recently, Granule [22] (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 [28] 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 [18] 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. [16, 17]. Also related is a system by Wu and Xi [34], which only mentions recursive session types as a possible extension, but does not investigate its properties. Zhou et al. [37, 38] 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, 13], 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 [16, 23, 29, 31]. 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 [16, 23]).

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 explored 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 ergonomic types for expressing amortized complexity, our language for temporal types for expressing parallel complexity has so far resisted a similar analysis, in essence because the next-time operator affects multiple channels at once.

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 [24, 25]. These would open up a whole new class of examples that are difficult or impossible to express in a purely linear logic such as Rast is at present.

Finally, prior work has explored a temporal linear type system for parallel complexity analysis [9] and we would like to explore if similar type-checking and reconstruction algorithms can be devised. However, its proof-theoretic properties are not as uniform as those for quantifiers, constraints, and ergonomic types.
REFERENCES


