Session Types with Arithmetic Refinements and Their Application to Work Analysis

Ankush Das  
Carnegie Mellon University  
USA  
ankushd@cs.cmu.edu

Frank Pfenning  
Carnegie Mellon University  
USA  
fp@cs.cmu.edu

Abstract
Session types statically prescribe bidirectional communication protocols for message-passing processes and are in a Curry-Howard correspondence with linear logic propositions. However, simple session types cannot specify properties beyond the type of exchanged messages. In this paper we extend the type system by using index refinements from linear arithmetic capturing intrinsic attributes of data structures and algorithms so that we can express and verify amortized cost of programs using ergonomic types. We show that, despite the decidability of Presburger arithmetic, type equality and therefore also type checking are now undecidable, which stands in contrast to analogous dependent refinement type systems from functional languages. We also present a practical incomplete algorithm for type equality and an explicit program, borrowing ideas from the proof-theoretic technique of focusing. We conclude by illustrating our systems and algorithms with a variety of examples that have been verified in our implementation.

Keywords  session types, resource analysis, refinement types

1 Introduction
Session types [29–31, 43] 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, 44] and are therefore of particular foundational significance. Moreover, type safety derives from properties of cut reduction and guarantees freedom from deadlocks (global progress) and session fidelity (type preservation) ensuring that at runtime the sender and receiver exchange messages conforming to the channel’s type.

However, basic session types have limited expressivity. As a simple example, consider the session type offered by a queue data structure storing elements of type $A$:

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

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 (none or some), indicated by the internal choice operator $\oplus$. It responds with none and then terminates (indicated by 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.

We propose extending session types with simple arithmetic refinements to express, for instance, the size of a queue. The more precise type

$\text{queue}_{A}[n] = \&\{\text{ins} : A \rightarrow \text{queue}_{A}[n + 1],$
\[
\text{del} : \oplus\{\text{none} : ?(n = 0), 1,$
\]
\[
\text{some} : ?(n > 0). A \otimes \text{queue}_{A}[n - 1]\}\}
\]

uses the index refinement $n$ to indicate the size of the queue. In addition, we introduce a type constraint $?(\phi)$. A which can be read as “there exists a proof of $\phi$” and is analogous to the assertion of $\phi$ in imperative languages. 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 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 (which is the subject of this paper) or dynamically (see [23, 24]). Although not used in this example, we also add the dual $!?(\phi)$. A (for all proofs of $\phi$, analogous to the assumption of $\phi$), and explicit quantifiers $\exists n. A$ and $\forall n. A$ that send and receive natural numbers, respectively.

Such arithmetic refinements are instrumental in expressing sequential and parallel complexity bounds. Prior work on ergonomic [15, 17] and temporal session types [16] rely on index refinements to express the size of lists, stacks and
queue data structures, or the height of trees and express work and time bounds as a function of these indices. However, they do not explore the metatheory or implementation of these arithmetic refinements or how they integrate with time and work analysis.

Of course, arithmetic type refinements are not new and have been explored extensively in functional languages, for example, by Zenger [48], in DML [46], or in the form of Liquid Types [38]. Variants have been adapted to session types as well [24, 25, 50], generally with the implicit assumption that index refinements are somehow “orthogonal” to session types. In this paper we show that, upon closer examination, this is not the case. In particular, unlike in the functional setting, session type equality and therefore type checking become undecidable. Remarkably, this is the case whether we treat session types equirecursively [21] or isorecursively [33], and even in the quantifier-free fragment. In response, we develop a new algorithm for type equality which, though incomplete, easily handles the wide variety of example programs we have tried (see Appendix A). Moreover, it is naturally extensible through the additional assertion of type invariants should the need arise.

With a practically effective type equality algorithm in hand, we then turn our attention to type checking. It turns out that assuming an oracle for type equality, type checking is decidable because it can be reduced to checking the validity of propositions in Presburger arithmetic. We define type checking over a language where constructs related to arithmetic constraints (∃n. A, ∀n. A, ?{ϕ}. A, and ![ϕ]. A) have explicit communication counterparts. Similarly, type constructs for receiving or sending potential (ϕ, A and !ϕ, A) for amortized work analysis have corresponding process constructs to (conceptually) send and receive potential. Revisiting the queue example, the type

\[
\text{queue}_A[n] = \& \{ \text{ins} : a^n (A \rightarrow \text{queue}_A[n + 1]), \text{del} : e^+ \oplus \{ \text{none} : ?(n = 0). 1, \text{some} : ?(n > 0). A @ \text{queue}_A[n - 1]) \}
\]

expresses that the client has to send 2n units of potential to enqueue an element, and 2 units of potential to dequeue. Despite the high theoretical complexity of deciding Presburger arithmetic, all our examples check very quickly using Cooper’s decision procedure [10] with two optimizations.

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 and amortized work analysis are omitted. The problem of reconstruction is then to map such an implicit program to an explicit one which is sound (the result type-checks) and complete (if there is a reconstruction, it can be found). Interestingly, the nature of Presburger arithmetic makes full reconstruction impossible. For example, the proposition ∀n. ∃k. (n = 2k ∨ n = 2k + 1) is true but the witness for k as a Skolem function of n (namely ⌊n/2⌋) cannot be expressed in Presburger arithmetic. Since witnesses are critical if we want to understand the work performed by a computation, we require that type quantifiers ∀n. A and ∃n. A have explicit witnesses in processes. We provide a sound and complete algorithm for the resulting reconstruction problem. This algorithm exploits proof-theoretic properties of the sequent calculus akin to focusing [1] to avoid backtracking and consequently provides precise error messages that we have found to be helpful.

We have implemented our language in SML, where a programmer can choose explicit or implicit syntax and the exact cost model for work analysis. The implementation consists of a lexer, parser, type checker, and reconstruction engine, with particular attention to providing precise error messages. To the best of our knowledge, this is the first implementation of ergometric session types with arithmetic refinements.

To summarize, we make the following contributions:

1. Design and implementation of a session-typed language with arithmetic refinements and ergometric types
2. Proof of undecidability of type equality for the small quantifier-free fragment of this language
3. A new type equality algorithm that works well in practice
4. A type checking algorithm that is sound and complete relative to type equality
5. A sound and complete reconstruction algorithm for a process language where most index and ergometric constructs remain implicit

2 Overview of Refinement Session Types

The underlying base system of session types is derived from a Curry-Howard interpretation [5, 6] of intuitionistic linear logic [22]. 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 \( P \).

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

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. All these channels must be distinct. We abbreviate the antecedent of the sequent by \( \Delta \).

We introduce a new type operator \( ?\{ϕ\}. A \) allowing processes to exchange constraints on these index refinements. The provider of \( ?\{ϕ\}. A \) sends a proof of \( ϕ \), which is received by the client. The dual of this operator is \( !(ϕ). A \) where the provider receives a proof of \( ϕ \) sent by the client. Because
Figure 1. Implementation of queue data structure

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

Figure 2. Implementations for the \text{empty} and \text{elem} processes.

label, the \text{empty} process takes the \text{none} branch (line 7) since it stores no elements. Therefore, it needs to send a proof of \( n = 0 \), and since it offers \text{queue}_A[0], it sends the trivial proof of \( 0 = 0 \) (line 8), and closes the channel terminating communication (line 9). The \text{elem} process receives the \text{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 \text{del} label, the \text{elem} process replies with the \text{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 expected by the id rule in Figure 3. 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 \text{none} to the \text{del} request. It would have to follow up with a proof that \( 1 = 0 \), which is of course impossible. Therefore, 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, postponing ergonomic types to Section 5. In addition to the type constructors arising from the connectives of intuitionistic linear logic (\( \otimes, \& \otimes, \& \neg \otimes \)), we have type names, indexed by a sequence of arithmetic expressions \( V[e] \), existential and universal quantification over natural numbers \( \exists n.A, \forall n.A \) and existential and universal constraints \( \langle \{\phi \}. A, \{!\phi \}. A \rangle \). We write \( i \) for constants.

#### Types

<table>
<thead>
<tr>
<th>( A )</th>
<th>( \otimes {\ell : A}_\ell \ell</th>
<th>\otimes {\ell : A}_\ell \ell</th>
<th>A \otimes \neg A</th>
<th>1</th>
<th>V[e]</th>
<th>\exists {\phi }. A</th>
<th>{!\phi }. A</th>
<th>\exists n.A</th>
<th>\forall n.A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( A \otimes \neg A</td>
<td>A \rightarrow A</td>
<td>1</td>
<td>V[e]</td>
<td>\exists {\phi }. A</td>
<td>{!\phi }. A</td>
<td>\exists n.A</td>
<td>\forall n.A</td>
<td></td>
</tr>
<tr>
<td>Arith. Exp.</td>
<td>( e )</td>
<td>( i )</td>
<td>( + )</td>
<td>( - )</td>
<td>( \times )</td>
<td>( e/n )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arith. Props.</td>
<td>( \phi )</td>
<td>( e = e</td>
<td>e &gt; e</td>
<td>\top</td>
<td>\bot</td>
<td>\phi \wedge \phi</td>
<td>\phi \vee \phi</td>
<td>\neg \phi</td>
<td>\exists n. \phi</td>
</tr>
</tbody>
</table>

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 \( A \).

We review a few basic session type operators before introducing the quantified type constructors. Table 1 overviews the session types, their associated process terms and operational description. Figure 3 describes selected typing rules (ignore the premises and annotation on the turnstile marked
where proving \( \Delta \) or message proc \( [8] \). We introduce semantic objects \( C \in V \in \Delta \) as a process expression, and \( c : A \to B \) is the client sends channel \( w : A \) along \( c \). When the message \( k \) is received along \( c \), we select branch \( k \) and return the continuation channel \( c' \) so that the message \( k \) is actually represented as \( (c \cdot k : c \to c') \) (read: send \( k \) along \( c \) and continue along \( c' \)).

The corresponding process term is written as \( (c \cdot k : c \to c) \). Operationally, it requires the provider to send one of the labels \( k \in L \) using the process term \( (x \cdot k : Q) \) where \( Q \) is the continuation. The typing for the provider and client are rules \&R and \&L in Figure 3 respectively. Communication is asynchronous, so the client \( c \cdot k : Q \) 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 \( (c \cdot k : c \to c') \) (read: send \( k \) along \( c \) and continue along \( c' \)).

The internal choice constructor \&\{ \ell : A_{\ell \in L} \} \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.

**3.1 Basic Session Types**

**External Choice** The external choice type constructor \&\{ \ell : A_{\ell \in L} \} \in L is an n-ary labeled generalization of the additive conjunction \( A \& B \). Operationally, it requires the provider of \( x : \&\{ \ell : A_{\ell \in L} \} \) 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 \( x (\ell \Rightarrow P_{\ell \in L}) \). Dually, the client must send one of the labels \( k \in L \) using the process term \( (x \cdot k : Q) \) where \( Q \) is the continuation. The typing for the provider and client are rules \&R and \&L in Figure 3 respectively. Communication is asynchronous, so that the client \( c \cdot k : Q \) 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 \( (c \cdot k : c \to c') \) (read: send \( k \) along \( c \) and continue along \( c' \)).

When the message \( k \) is received along \( c \), we select branch \( k \) and return the continuation channel \( c' \) so that the message \( k \) is actually represented as \( (c \cdot k : c \to c') \) (read: send \( k \) along \( c \) and continue along \( c' \)).

3.1 Basic Session Types

**External Choice** The external choice type constructor \&\{ \ell : A_{\ell \in L} \} \in L is an n-ary labeled generalization of the additive conjunction \( A \& B \). Operationally, it requires the provider of \( x : \&\{ \ell : A_{\ell \in L} \} \) 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 \( x (\ell \Rightarrow P_{\ell \in L}) \). Dually, the client must send one of the labels \( k \in L \) using the process term \( (x \cdot k : Q) \) where \( Q \) is the continuation. The typing for the provider and client are rules \&R and \&L in Figure 3 respectively. Communication is asynchronous, so that the client \( c \cdot k : Q \) 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 \( (c \cdot k : c \to c') \) (read: send \( k \) along \( c \) and continue along \( c' \)).

The internal choice constructor \&\{ \ell : A_{\ell \in L} \} \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.

**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 \( send \ y : P \) where \( P \) is the continuation. Correspondingly, its client must receives a channel using the term \( y \leftarrow recv \ x : Q \), binding it to variable \( y \) and continuing to execute \( Q \). The typing rules are \&R, \&L in Figure 3. The dual operator \( A \rightarrow B \) allows the provider to receive a channel of type \( A \) and continue to provide type \( B \).
Finally, the type 1 indicates termination operationally denoting that the provider send a close message followed by terminating the communication. The complete set of static and dynamic semantics is described in Appendix C.

Forwarding A process \( x \leftarrow y \) 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 (id in Figure 3). Its computation rules can be found in Appendix C.

Process Definitions Process definitions have the form \( \Delta \overset{f}[\pi] = P :: (x : A) \) where \( f \) is the name of the process and \( P \) its definition. In addition, \( \pi \) is a sequence of arithmetic variables that \( \Delta \), \( P \) and \( A \) can refer to. All definitions are collected in a fixed global signature \( \Sigma \). We require that \( \pi \) should not itself be a type name. Our type definitions are indexed with arithmetic refinements by \( \Sigma \) in the signature \( \Delta \) of \( P \). For this purpose we allow (possibly mutually recursive) type definitions \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) for every definition, thereby allowing definitions to be mutually recursive.

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 \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) in the signature \( \Sigma \). Here, \( [\pi] \) denotes a sequence of arithmetic variables. We also require \( A \) to be contractive [21] meaning \( A \) should not itself be a type name. Our type definitions are equirecursive so we can silently replace type names \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) indexed with arithmetic refinements by \( A[\theta] \) during type checking, and no explicit rules for recursive types are needed.

All types in a signature must be valid, formally written as \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) which requires that all free arithmetic variables of \( C \) and \( A \) are contained in \( \Sigma \), and that for each arithmetic expression \( e \) in \( \Sigma \) we can prove \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) (implicitly proving that \( e \geq 0 \)).

3.2 The Refinement Layer

We now introduce quantifiers (\( \exists \theta \in \mathbb{N} \)) and constraints (\( \exists e \in \mathbb{N} \)). An overview of the types, process expressions, and their operational meaning can be found in Table 2.

Quantification The provider of \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) 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 \( i \) (although we do not bother formally representing this evaluation). From the typing perspective, we just need to check that the expression \( e \) denotes a natural number, using only the permitted variables in \( \Sigma \). This is represented with the auxiliary judgment \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) (implicitly proving that \( e \geq 0 \) under constraint \( \Sigma \)).

The provider of \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) 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 \( i \) (although we do not bother formally representing this evaluation). From the typing perspective, we just need to check that the expression \( e \) denotes a natural number, using only the permitted variables in \( \Sigma \). This is represented with the auxiliary judgment \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) (implicitly proving that \( e \geq 0 \) under constraint \( \Sigma \)).

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 \( \forall \theta \in \mathbb{N} \quad \Sigma \vdash \tau \rightarrow P :: (x : A) \) should send a proof of \( e \) 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 \( e \) holds, while the client is permitted to assume that \( e \) is true. Therefore, and in an analogy with imperative languages, we write assert \( e \) for channel \( c \)
and continues with $P$, while assume $c\{\phi\};Q$ assumes $\phi$ and continues with $Q$.

Thus, the typing rules for this new type constructor are

$$
\begin{align*}
\Gamma; C \vdash \phi & \quad \Gamma; C, \Delta \nvdash P : (x : A) \quad \text{?}\text{R} \\
\Gamma; C, \Delta \nvdash \text{assert } x \{\phi\} & ; P : (x : \{\phi\}.A) \\
\Gamma; C, \Delta, (x : \{\phi\}.A) \nvdash Q : (z : C) & \text{?L}
\end{align*}
$$

Notice how the provider must verify the truth of $\phi$ given the currently known constraints $C (\Gamma; C \vdash \phi)$, while the client assumes $\phi$ by adding it to $C$. In well-typed configurations (which arise from executing well-typed processes, see Appendix C) 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

$$
\begin{align*}
\Gamma; C, \Delta, (x : \{\phi\}.A) \nvdash Q : (z : C) & \text{?L} \\
\Gamma; C, \Delta \nvdash \text{assert } x \{\phi\} & ; P : (x : \{\phi\}.A) \\
\Gamma; C \vdash \phi & \quad \Gamma; C, \Delta (x : A) \nvdash Q : (z : C) \\
\Gamma; C \vdash \phi & \quad \Gamma; C \nvdash \text{assert } x \{\phi\} & ; P : (x : (x : A))
\end{align*}
$$

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

$$
c : \{\text{none} : ?\{0=0\}, \text{some} : ?\{0>0\}\} \nvdash A \nvdash \text{queue}_A[0–1]
$$

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

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

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

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

In implicit syntax (see Section 6) 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

$$
\Gamma; C \vdash \perp \quad \text{unsat}
$$

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.

The extension of session types with index refinements is type safe, expressed using the usual proofs of preservation and progress on configurations. A well-typed configuration, represented using the judgment $\Delta_1 \vdash_\Sigma S :: \Delta_2$ denotes a set of semantic objects $S$ using channels in $\Delta_1$ and offering channels $\Delta_2$. The proof of preservation proceeds by induction on the operational semantics and inversion on the configuration and process typing judgment.

To state progress, we need the notion of a poised process [35]. A process $\text{proc}(c, w, P)$ is poised if it is trying to receive a message on $c$. Dually, a message $\text{msg}(c, w, M)$ is poised if it is sending along $c$. A configuration is poised if every message or process in the configuration is poised. Intuitively, this means that every process in the configuration is trying to interact with the outside world.

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

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

(ii) (Preservation) If $S \rightsquigarrow S'$, then $\Delta_1 \vdash_\Sigma S' :: \Delta_2$

## 4 Type Equality

At the core of an algorithm for type checking is type equality. It is necessary for the rule of identity (operationally: forwarding) as well as the channel-passing constructs for types $A \otimes B$ and $A \rightsquigarrow B$. Informally, two types are equal if they permit exactly the same communication behaviors. For example, if $\text{nat} = \{\text{zero} : 1, \text{succ} : \text{nat}\}$

$\text{nat}' = \{\text{zero} : 1, \text{succ} : \{\text{zero} : 1, \text{succ} : \text{nat}\}\}$

### Table 2. Refined session types with operational description

<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></td>
<td></td>
<td>${n} \gets \text{recv } c ; Q$</td>
<td>$Q[i/n]$</td>
<td>client receives number $i$ along $c$</td>
</tr>
<tr>
<td>$c : \forall n. A$</td>
<td>$c : A[i/n]$</td>
<td>${n} \gets \text{recv } c ; P$</td>
<td>$P[i/n]$</td>
<td>provider receives number $i$ along $c$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send $c{e} ; Q$</td>
<td>$Q$</td>
<td>client sends value $i$ of $e$ along $c$</td>
</tr>
<tr>
<td>$c : {\phi}. A$</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></td>
<td></td>
<td>assume $c{\phi} ; Q$</td>
<td>$Q$</td>
<td>client assumes $\phi$ on $c$</td>
</tr>
<tr>
<td>$c : {\phi}. A$</td>
<td>$c : A$</td>
<td>assert $c{\phi} ; P$</td>
<td>$P$</td>
<td>provider assumes $\phi$ on channel $c$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assume $c{\phi} ; Q$</td>
<td>$Q$</td>
<td>client assumes $\phi$ on $c$</td>
</tr>
</tbody>
</table>
we have that $\text{nat} \equiv \text{nat}'$ because each type allows any sequence of $\text{succ}$ labels, followed by $\text{zero}$ followed by $\text{close}$. This carries over to indexed types. If we define $\text{nat}[n] = \oplus \{? : \{n = 0\}. 1, \text{succ} : \{n > 0\}. \text{nat}[n - 1]\}$ then $\text{pos}[n + 1] = \text{nat}[n] + 1$ for all $n$.

Following seminal work by Gay and Hole [21] type equality is formally captured as a coinductive definition. Gay and Hole actually define subtyping ($A \leq B$ if every behavior permitted by $A$ is also permitted by $B$) and derive type equality from it; we simplify matters by directly defining equality. Subtyping follows the same pattern and presents no additional complications. Our definitions capture equirecursive type equality, but they can easily be adapted to an isorecursive equality with explicit fold messages [18, 33].

**Definition 1.** We define $\text{unfold}_R(A)$ as

$$\text{V}[n] = A \in \Sigma \quad \text{unfold}_R(\text{V}[\ell]) = \text{A}[\ell / \text{V}] \quad \text{def} \quad A \neq \text{V}[\varepsilon] \quad \text{unfold}_R(A) = A \quad \text{str}$$

Like Gay and Hole, we require type definitions to be contractive, so the result of unfolding is never a type variable.

**Definition 2.** A relation $R \subseteq \text{Type} \times \text{Type}$ is a type bisimulation if $(A, B) \in R$ implies (analogous cases omitted):

- If $\text{unfold}_R(A) = \oplus \{\ell : A_\ell\}_{\ell \in L}$, then $\text{unfold}_R(B) = \oplus \{\ell : B_\ell\}_{\ell \in L}$ and $(A_\ell, B_\ell) \in R$ for all $\ell \in L$.
- If $\text{unfold}_R(A) = A_1 \twoheadrightarrow A_2$, then $\text{unfold}_R(B) = B_1 \twoheadrightarrow B_2$ and $(A_1, B_1) \in R$ and $(A_2, B_2) \in R$.
- If $\text{unfold}_R(A) = \{\phi\}, A'$, then $\text{unfold}_R(B) = \{\psi\}, B'$ and either $\vdash \phi \leftrightarrow \psi$ and $(A', B') \in R$, or $\vdash \phi \land \psi$ and $\vdash \phi$.
- If $\text{unfold}_R(A) = \exists m. A'$, then $\text{unfold}_R(B) = \exists n. B'$ and for all $m \in \mathbb{N}$, $(A'[m/i], B'[n/i]) \in R$.

**Definition 3.** Two types $A$ and $B$ are equal ($A \equiv B$) iff there exists a type bisimulation $R$ such that $(A, B) \in R$.

This definition only applies to ground types with no free variables. Since we allow quantifiers and arithmetic constraints, we need to define equality in the presence of free variables and arbitrary constraints. To this end, we define the notion $\forall \forall V. C \Rightarrow A \equiv B$ under the presupposition that $A$ and $B$ are valid type under assumption $C$. Interestingly, this definition implies that if $C$ is unsatisfiable, then $A \equiv B$ for all valid types $A$ and $B$.

**Definition 4.** We define $\forall \forall V. C \Rightarrow A \equiv B$ iff for all ground substitutions $\sigma : V$ satisfying $C$ (that is, $\vdash C[\sigma]$), we have $A[\sigma] \equiv B[\sigma]$.

**Theorem 2.** Checking $\forall \forall V. C \Rightarrow A \equiv B$ is undecidable.

**Proof.** Given a two counter machine, we construct two types $A$ and $B$ such that the computation of the machine is infinite iff $A \equiv B$. Thus, we establish a reduction from non-halting problem to type equality.

The type system allows us to simulate the two counter machine. Intuitively, the quantified type constructors allow us to branch depending on arithmetic constraints. This coupled with arbitrary recursion in the language establishes undecidability. Remarkably, the small fragment of our language containing only type definitions, internal choice ($\oplus$) and assertions ($\{\phi\} . A$) constructors is sufficient to prove undecidability. Moreover, the proof still applies if we treat types isorecursively. Appendix B contains the machine construction and details of the undecidability proof.

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

We explain the algorithm with an illustrative example.

$$\text{ctr}[x, y] = \oplus \{\text{lt} : \{x < y\}. \text{ctr}[x + 1, y], \text{ge} : \{x \geq y\}. 1\}$$

The $\text{ctr}$ type outputs $\text{lt}$ or $\text{ge}$ based on comparing $x$ and $y$ and recurses with $\text{ctr}[x + 1, y]$ if $x < y$. Compare types $\text{ctr}[x, y]$ and $\text{ctr}[x + 1, y + 1]$. They will both output $\text{lt}$ exactly $\max(0, y - x)$ number of times terminating with $\text{ge}$, and thus are equal according to our definition. Suppose we wish to prove $\text{ctr}[x, y] \equiv \text{ctr}[x + 1, y + 1]$ for all $x, y \in \mathbb{N}$. We use the algorithmic equality judgment $\forall \forall V : C ; \Gamma \vdash A \equiv B$ to denote checking given variables $V$ satisfying constraint $C$, whether types $A$ and $B$ are equal. Since the algorithm is coinductive, $\Gamma$ stores equality constraints encountered so far. We initiate the algorithm with an empty $\Gamma$ and $\Gamma = \top, \text{thereby checking } x, y ; \top ; \vdash \text{ctr}[x, y] \equiv \text{ctr}[x + 1, y + 1]$.

$$V_1, V_2 \equiv \text{V} ; C ; \Gamma \vdash V_1, V_2 \equiv \text{V} ; \text{ctr}[x, y] \equiv \text{ctr}[x + 1, y + 1] \quad \text{expd}$$

The expd rule adds the equality constraint $V_1[\varepsilon_1] \equiv V_2[\varepsilon_2]$ to $\Gamma$ and expands the two sides by replacing each type name with its definition. In our example, we add $\forall X. Y . \text{ctr}[X, Y] \equiv \text{ctr}[X + 1, Y + 1]$ to $\Gamma$ (a-renamed to avoid confusion) and
where there are no index expressions, the algorithm behaves

Therefore, we conclude that

Thus, we substitute

such that

First, we check the lt branch: \( (x < y). \ c t r [x + 1, y] \equiv \exists (x + 1 < y + 1). \ c t r [x + 2, y + 1] \). We use the \( ? \) rule to check equivalence of the two constraints

If the assumption of \( \phi \) were to make the constraints contradictory we would succeed at this point, using the rule

in accordance with Definition 4. In our example, since \( x < y \rightarrow x + 1 < y + 1 \) and \( x < y \) is consistent, we compare \( c t r [x + 1, y] \equiv c t r [x + 2, y + 1] \). At this point, we check if this equality is entailed by one of the stored equality constraints stored. The simplest case of such an entailment is witnessed by a substitution, applied to one of the stored equality constraints. And yes, since we stored \( (\forall x, y. \ c t r [x, y] \equiv c t r [x + 1, y + 1]) \), we can substitute \( x + 1 \) for \( x \) and \( y \) for \( y \) to satisfy the desired equality constraint. This is the coinductive aspect of the algorithm formalized in the def rule.

For our example, this reduces to checking the validity of \( \forall x, y. \ c t r [x, y] \equiv c t r [x + 1, y + 1] \). Similarly, for the ge branch, we check if \( (\exists x \geq y). \ 1 \equiv \exists (x + 1 \geq y + 1). \ 1 \). Since the two constraints are equivalent and \( 1 \equiv 1 \), we infer that the types match in both branches. Thus, we conclude that \( c t r [x, y] \equiv c t r [x + 1, y + 1] \). The rest of the rules of the type equality algorithm are similar to the ones presented (full rules in Appendix B).

The system so far is potentially nonterminating because when encountering variable/variable equations, we can use \( \text{expd} \) indefinitely. To ensure termination we use two techniques. The first is to introduce internal names for every subexpression of type definitions. This means the algorithm alternates between comparing two type names and two type constructors. The second is to restrict the \( \text{expd} \) rule to the case where no assumption of the form \( \forall \forall \forall'. \ C' \Rightarrow V_1 \{ [E_1] \} \equiv V_2 \{ [E_2] \} \) is already present in \( \Gamma \). This means that for the case where there are no index expressions, the algorithm behaves exactly like Gay and Hole’s and is terminating: we close a branch when we find the equation \( V_1 \equiv V_2 \) in \( \Gamma \).

We prove that the type equality algorithm is sound with respect to the declarative equality definition. The soundness is proved by constructing a type bisimulation from a derivation of the algorithmic type equality judgment (Appendix B).

**Theorem 3.** If \( V; C; \Gamma \vdash A \equiv B \), then \( \forall \forall \forall'. \ C \Rightarrow A \equiv B \).

**Reflexivity** Because it is not always derivable with the rules so far, we found it necessary to add one more rule to the algorithm, namely reflexivity on type names when indices are provably equal. This is still sound since the reflexive closure of a type bisimulation is still a type bisimulation.

Traditional refinement languages such as DML [46] only use reflexivity as a criterion for equality of indexed type names. However, as exemplified in the \( c t r \) example, our algorithm extends beyond reflexivity.

### 5 Ergometric Session Types

An important application of refinement types is complexity analysis. To describe the resource contracts for interprocess communication, the type system is enhanced to support amortized resource analysis [39]. The key idea is that processes store potential and messages carry potential. This potential can either be consumed to perform work or exchanged using special messages. The type system provides the programmer with the flexibility to specify what constitutes work. Thus, the programmer can choose to count the resource they are interested in, and the type system provides the corresponding upper bound. Our current examples assign unit cost to message sending operations (\( c.k \), close \( c \), send \( c \ d \) (exempting those for index objects send \( c \{ e \} \), assert \( c \{ \phi \} \), and potential pay \( c \{ r \} \), see below) effectively counting the total number of "real" messages exchanged during a computation.

Two dual type constructors \( \triangleright A \) and \( \triangleleft A \) are used to exchange potential. The provider of \( x : \triangleright A \) must pay \( r \) units of potential along \( x \) using process term (pay \( x \{ \{ r \} \} \) \), and continue to provide \( A \) by executing \( P \). These \( r \) units are deducted from the potential stored inside the sender. Dually, the client must receive the \( r \) units of potential using the term (get \( x \{ r \} \); \( Q \)) and add this to its internal stored potential. Finally, since processes are allowed to store potential, the typing judgment is enhanced by adding a natural number on the turnstile denoting its internal potential.

We allow potential \( q \) to refer to index variables in \( V \). The typing rules for \( \triangleright A \) are

We allow potential \( q \) to refer to index variables in \( V \). The typing rules for \( \triangleright A \) are

We allow potential \( q \) to refer to index variables in \( V \). The typing rules for \( \triangleright A \) are

We allow potential \( q \) to refer to index variables in \( V \). The typing rules for \( \triangleright A \) are
In both cases, we check that the exchanged potential in the expression and type matches \((r_1 = r_2)\), and while paying, we ensure that the sender has sufficient potential to pay. Operationally, the provider creates a special message containing the potential that is received by the client.

\[(\triangleright \sigma) \quad \text{proc}(c, w, \text{pay} c \{r\} ; P) \mapsto \text{proc}(c', w, P[c'/c]), \text{msg}(c, 0, \text{pay} c \{r\} ; c \leftarrow c')\]

\[(\triangleright \sigma) \quad \text{msg}(c', w', \text{pay} c \{r\} ; c \leftarrow c'), \text{proc}(d, w), \text{get} c \{r\} ; Q) \mapsto \text{proc}(d, w + w', Q[c'/c])\]

The dual type \(\triangleright \sigma A\) enables the provider to receive potential that is sent by its client.

\[\mathcal{V} ; C \triangleright r_1 = r_2 \quad \mathcal{V} ; C ; A \triangleright r \quad P :: (x : A) \quad \Rightarrow \mathcal{V} ; C ; A \triangleright r_1 \quad \triangleright \sigma R\]

\[\mathcal{V} ; C \triangleright q \geq r_1 = r_2 \quad \mathcal{V} ; C ; (x : A) \triangleright r_2 \quad Q :: (z : C) \quad \Rightarrow \mathcal{V} ; C ; (x : A) \triangleright r \quad \triangleright \sigma L\]

The work counter \(w\) in the semantic objects \(\text{proc}(c, w, P)\) and \(\text{msg}(c, w, M)\) tracks the work done by the process. We use a special expression \(\text{work} \{r\} ; P\) to increment the work counter. None of the other rules affect this counter but simply preserve their total sum. The programmer can insert these flexibly in the program to count a specific resource. For example, to count the total number of messages sent, we insert \(\text{work} \{1\}\) just before sending every message.

\[\text{proc}(c, w, \text{work} \{r\} ; P) \mapsto \text{proc}(c, w + r, P)\]

Statically, type checking this construct requires potential. No other typing rule affects the potential.

\[\mathcal{V} ; C \triangleright q \geq r \quad \mathcal{V} ; C ; (x : A) \triangleright r \quad P :: (x : A) \quad \Rightarrow \mathcal{V} ; C ; (x : A) \triangleright r \quad \text{work} \{r\} ; P :: (x : A)\]

Since the amount of potential consumed to type check this expression is equal to the amount of work performed by it, the type safety theorem expresses that the total work done by a system can never exceed its initial potential.

The algorithms for type checking and equality extend easily to these new constructors since we already track variables and constraints over natural numbers in the judgments.

**Queue Example** Revisiting the ergometric session type of the queue data structure, we count the total number of messages exchanged in the system.

Queue function \(\text{queue}_A[n] = \& \{\text{ins} : 2^n (A \rightarrow \text{queue}_A[n + 1]),\)

\[\text{del} : \triangleright \sigma \oplus \{\text{none} : (?[n = 0], 1, \text{some} : (?[n > 0], A @ \text{queue}_A[n - 1])\}\]

\[\Delta \triangleright \emptyset :: (s : \text{queue}_A[0]) \quad (x : A), (t : \text{queue}_A[n]) \triangleright \text{elem}[n] :: (s : \text{queue}_A[n + 1])\]

Thus, the queue requires \(2n\) units of potential to insert and \(2n\) units of potential to delete an element. The \(2n\) units are used to carry the \(\text{ins}\) message and the element to insert to the end of the queue. While deletion, the process sends 2 messages, either \(\text{none}\) and close (lines 7, 9 in Figure 2) or \(\text{some}\) and the element stored (lines 17, 19). Operationally, when the \(\text{elem}\) process receives \(2(n + 1)\) units on \(s\), it consumes \(2n\) units to send 2 messages (lines 14, 15 in Figure 2) and sends the remaining \(2n\) units on \(t\) in accordance with the prescribed type. And on deletion, the \(2n\) units are consumed to send the two messages (lines 17, 19).

### 6 Constraint and Work 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 assume, assert, pay, get and work expressions in their code; constructs corresponding to proof constraints and potential (quantifiers are still explicit). 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, assert, pay, get and work constructs. Moreover, impossible branches may be omitted from case expressions. The missing branches and other constructs are restored by a type-directed process of reconstruction. In the first phase, a case expression with a missing branch for label \(\ell\) is extended by a branch \(\ell \Rightarrow \text{impossible}\) so that type checking later verifies that the omitted branch is indeed impossible. Then assume and asserts, and finally pay and gets are inserted according to a reconstruction algorithm described in this section. Finally, since the potential is treated as linear (must be \(0\) before termination), work constructs are inserted just before the terminating expression to consume the leftover potential.

Following branch reconstruction, the resulting process expression is typechecked with the implicit typing judgment \(\mathcal{V} ; C :: \Delta \triangleright P :: (x : A)\) using the rules in Figure 4 (analogous rules \(\triangleright R, \triangleright L, \triangleright \sigma R, \triangleright \sigma L, \triangleright L, \triangleright \sigma L, \triangleright \sigma R, \triangleright L, \triangleright L, \triangleright \sigma L, \triangleright L\) omitted). The only difference from the explicit system is that the process expressions do not change on application of these rules. The remaining rules exactly match the explicit system (see Appendix C).

The implicit rules are trivially 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. Given an implicit source program, there may be many different corresponding explicit programs depending...
on when the rules in Figure 4 are applied. The necessary backtracking would greatly complicate error messages and could also be 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 and get should be inserted as early as possible, and assert and pay 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). A similar reasoning holds for the potential: a sequence of potential exchanges can always be reordered as eagerly receiving all the potential, and then sending it only when required. Logically, the !R, ?L, âRvL rules are invertible, and are applied eagerly while their dual rules are applied lazily. The result is again complete under the assumption of polarizable types.

This strategy is formally realized in the forcing calculus using the judgment \( \nu; C; \Delta; P :: x : A \). The context is split into two: the linear context \( \Delta \) contains stable propositions on which the invertible left rules have been applied, while the ordered context \( \Omega \) 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 [1] 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 \( S \).

\[
\begin{align*}
A^+ &::= S | \{ \phi \} . A^* | \triangleright A^* \\
A^- &::= S | \{ \phi \} . A^- | \triangleright A^- \\
A &::= A^* | A^- \\
S &::= \ominus \{ \ell : A \ell \subseteq L \} \cup \{ \ell : A \ell \subseteq L \} | A \Theta A | 1 | A \rightarrow A \\
&| \exists n. A | \forall n. A
\end{align*}
\]

Not all types can be polarized in this manner. For example, \( \{ \phi \} . \{ \psi \} . A \) or \( \triangleright \triangleright \triangleright A \). When checking the validity of types before performing reconstruction we reject such types with alternating polarities since our deterministic algorithm would be incomplete and we have found no need for them. We also require that all process declarations contain only structural types at the top-level.

Thus, ? and > operators are positive, while ! and < are negative. The structural types, denoted by \( S \) are considered neutral. In the forcing calculus, the invertible rules are applied first (analogous to \( L \rightarrow \text{omitted} \)).

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

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

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 \( \nu; C; \Delta^-; \triangleright P :: (x : A^+) \), i.e., the ordered context is empty and the offered type \( A^+ \) is positive. A stable sequent implies that all constraints and potential have been received. We send a constraint or potential 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.

\[
\begin{align*}
\nu; C; \Delta^-; \triangleright P :: (x : A^+) &\text{ move} \\
\nu; C; \Delta^-; \triangleright P :: (x : A^+)
\end{align*}
\]

The square brackets \( [\cdot] \) indicates that the channel is forced, indicating that a communication is about to happen on it. If there are assert or pay constructs pending on the forced channel, they are applied now (analogous to \( L \rightarrow \text{omitted} \)).

\[
\begin{align*}
\nu; C; \Delta^-; \triangleright P :: (x : A^+) &\text{ move} \\
\nu; C; \Delta^-; \triangleright P :: (x : A^+)
\end{align*}
\]

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.

\[
\begin{align*}
\nu; C; \Delta^-; \triangleright P :: (x : A_k) &\text{ move} \\
\nu; C; \Delta^-; \triangleright P :: (x : A_k)
\end{align*}
\]

Not all types can be polarized in this manner. For example, \( \{ \phi \} . \{ \psi \} . A \) or \( \triangleright \triangleright \triangleright A \). When checking the validity of types before performing reconstruction we reject such types with alternating polarities since our deterministic algorithm would be incomplete and we have found no need for them. We also require that all process declarations contain only structural types at the top-level.

Thus, ? and > operators are positive, while ! and < are negative. The structural types, denoted by \( S \) are considered neutral. In the forcing calculus, the invertible rules are applied first (analogous to \( L \rightarrow \text{omitted} \)).
We use a straightforward implementation of Cooper’s algorithm [10] to decide Presburger arithmetic with two small but significant optimizations. One takes advantage of the fact that we are working over natural numbers rather than integers which bounds possible solutions from below, and the other is to eliminate constraints of the form $x - y$. We have implemented a prototype for the language in Stan-

work. Verifying constraints is postponed to the final pass of the reconstruction process. 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.

7 Implementation and Evaluation

We have a variety of 9 case studies implemented, totaling over 6500 lines of code) 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. The experiments were run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory. We briefly describe each case study.

1. **arithmetic**: natural numbers in unary and binary representation indexed by their value and processes implementing standard arithmetic operations.

2. **integers**: an integer counter represented using two indices $x$ and $y$ with value $x - y$.

3. **linam**: expressions in the linear $\lambda$-calculus indexed by their size with an eval process to evaluate them.

4. **list**: natural number lists indexed by their size, and their standard operations: append, reverse, map, fold, etc.

5. **primes**: implementation of the sieve of Eratosthenes to classify numbers as prime or composite.

6. **segments**: type $\mathsf{seg}[n] = \forall k. \mathsf{list}[k] \rightarrow \mathsf{list}[n + k]$ representing partial lists with constant-work append operation.

7. **ternary**: natural numbers represented in balanced ternary form with digits $0$, $1$, $-1$, indexed by their value, and some standard operations on them.

8. **theorems**: processes representing (circular [18]) proofs of simple theorems such as $n(k + 1) = nk + n$.

9. **tries**: a trie data structure to store multisets of binary numbers, with constant amortized work insertion and deletion verified with ergonomic types.

We give details for some of these examples in Appendix A.

7.1 Queue as Two Lists

The running queue $\mathcal{A}[n]$ example has a linear insertion cost, since the element travels to the end of the queue where it is inserted. However, a more efficient implementation of a queue using two lists (or stacks) [17] has a constant amortized cost for insertion and deletion.

Elements are inserted into an input list, and removed from an output list. If the output list is empty at removal, the input list is reversed and made the output list. The input list stores extra potential (4 units) which is consumed during reversal. Since ergonomic types support amortized analysis, we obtain the constant cost type $\mathcal{A}[n] = \mathfrak{s} \{ \mathfrak{n} : \mathfrak{e} (\mathfrak{A} \rightarrow \mathcal{A}[n+1]) \}$.

**Binary Numbers**

We can represent binary numbers indexed by their value as sequences of bits $\mathfrak{b}0$ and $\mathfrak{b}1$ followed by $\mathfrak{e}$ to indicate the end of the sequence.

$\mathfrak{bin}[n] = \mathfrak{b}0 (? \{ n > 0 \} \cdot \mathfrak{e} \cdot \mathfrak{b}1 (? \{ n > 0 \} \cdot \mathfrak{e} \cdot \mathfrak{e} (? \{ n = 0 \} \cdot \mathfrak{b}1 (? \{ n = 2 \cdot k \} \cdot \mathfrak{b}0) \cdot \mathfrak{b}1) \cdot \mathfrak{b}1) \cdot \mathfrak{b}0) \cdot \mathfrak{b}1 \cdot \mathfrak{e} \cdot ? \{ n = 0 \} \cdot 1 \}$

A binary number on outputting $\mathfrak{b}0$ (or $\mathfrak{b}1$) must send a proof of $n \gg 0$ (no leading zeros) and a witness $k$ such that $n = 2k$ (resp. $n = 2k + 1$) before continuing with $\mathfrak{b}0[k]$. While outputting $\mathfrak{e}$, it must send a proof that $n = 0$ and then terminate, as described in the type. Moreover, since we reconstruct impossible branches, when a programmer implements a predecessor process declared as

$(m : \mathfrak{bin}[n + 1]) \rightarrow \mathfrak{pred}[n] :: (n : \mathfrak{bin}[n])$
they can skip the impossible case of label e (since \( n + 1 \neq 0 \)).

**Linear \( \lambda \)-calculus**  We demonstrate an implementation of the (untyped) linear \( \lambda \)-calculus in which the index objects track the size of the expression.

\[
\text{exp}[n] = \oplus \{ \text{lam} : \{ n > 0 \}, \forall n_1, \text{exp}[n_1] \to \text{exp}[n_1 + n - 1], \\
\text{app} : \exists n_1, \exists n_2, \{ n = n_1 + n_2 + 1 \}, \text{exp}[n_1] \otimes \text{exp}[n_2] \}
\]

An expression is either a \( \lambda \) (label lam) or an application (label app). In case of lam, it expects a number \( n_1 \) and an argument of size \( n_1 \) and then behaves like the body of the \( \lambda \)-abstraction of size \( n_1 + n - 1 \). In case of app, it will send \( n_1 \) and \( n_2 \) such that \( n = n_1 + n_2 + 1 \), then an expression of size \( n_1 \) and then behaves as an expression of size \( n_2 \).

A value can only be a \( \lambda \) expression \n\[
\text{val}[n] = \oplus \{ \text{lam} : \{ n > 0 \}, \forall n_1, \text{exp}[n_1] \to \text{exp}[n_1 + n - 1] \}
\]

so the app label is not permitted. Type checking verifies that the result of evaluating a linear \( \lambda \)-term is no larger than the original term.

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

### 8 Further Related Work

Refinement types were introduced to allow specification and verification of recursively defined subtypes of user-defined types [20, 46], but have since been applied for complexity analysis [11, 32]. Dal Lago and Gaboardi [12] designed a system of linear dependent types for the \( \lambda \)-calculus with higher-order recursion and use it for complexity analysis [13]. Refinement [26] and dependent types [14] have also been employed to reason about efficiency of lazy functional programs. Refinement type and effect systems have been proposed for incremental computational complexity [9] and relational cost analysis of functional [7] and functional-imperative programs [37]. Automatic techniques for complexity analysis of sequential [27] and parallel programs [28] that do not rely on refinements have also been studied. In contrast to these articles that use nominal types and apply to functional programs, the structural type system of session types poses additional theoretical and practical challenges for deciding type equality, type checking, and reconstruction.

Label-dependent session types [40] use a limited form of dependent types where values can depend on labels drawn from a finite set. They use this to encode general binary session types and also extend the types with primitive recursion over natural numbers, although unlike our work, they do not support general recursive types. Toninho and Yoshida [42] propose a dependent type theory combining functions and session types through a contextual monad allowing processes to depend on functions and vice-versa. Unlike our equality algorithm that involves no type-level computation, they rely on term (or process) equality to define type equality. Toninho et al. [41] develop an interpretation of linear type theory as dependent session types for a term passing extension of the \( \pi \)-calculus to express interface contracts and proof-carrying certification. However, they do not discuss a type equality algorithm nor provide an implementation. Wu and Xi [45] propose a dependent session type system of DML style based on ATS [47] formalizing type equality in terms of subtyping and regular constraint relations. In contrast to our refinement layer, none of these dependent type systems are applied for complexity analysis.

LiquidPi [25] applies the idea of refinements to session types to describe and validate safety properties of distributed systems. They also present an algorithm for inferring these refinements when they are restricted to a finite set of templates. However, they do not specifically explore the fragment of arithmetic refinements, nor apply them to study resource analysis. Linearly refined session types [2] extend the \( \pi \)-calculus with capabilities from a fragment of multiplicative linear logic. These capabilities encode authorization logic enabling fine-grained specifications, i.e., a process can take an action only if it contains certain capabilities. Franco and Vasconcelos [19] implement these linearly refined session types in a language called SePi. In this work, we explore arithmetic refinements that are more general than a multi-set of uninterpreted formulae [2]. Bocchi et al. [4] present asynchronous timed session types to model timed protocols, ensuring processes perform actions in the time frame prescribed by their protocol. Zhou et al. [50] refine base types with arithmetic propositions [49] in the context of multiparty session types without recursive types. In this restricted setting, subtyping and therefore type equality is decidable and much simpler than in our setting. Finally, session types with limited arithmetic refinements (only base types could be refined) have been proposed for the purpose of runtime monitoring [23, 24], which is complementary to our uses for static verification. They have also been proposed to capture work [15, 17] and parallel time [16], but parameterization over index objects was left to an informal meta-level and not part of the object language. Consequently, these languages contain neither constraints nor quantifiers, and the metatheory of type equality, type checking, and reconstruction in the presence of index variables was not developed.

### 9 Conclusion

This paper explored the metatheory of session types extended with arithmetic refinements. The type system was enhanced with quantifiers and type constraints and applied to verify sequential complexity bounds (characterizing the total work) of session-typed programs.

In the future we plan to pursue several natural generalizations. In multiple examples we have noted that even nonlinear arithmetic constraints that arise have simple proofs, despite their general undecidability, so we want to develop a heuristic nonlinear solver. Secondly, much of the theory in this paper is modular relying on a few simple properties of quantified linear arithmetic and could easily be generalized to other domains such as quantifier-free index domains with SMT solvers, arbitrary integers, modular arithmetic,
and fractional potentials. We would also like to generalize our approach to a mixed linear/nonlinear language [3] or all the way to adjoint session types [35, 36].

We also plan to explore automatic inference of potential annotations. Currently, programmers have to compute work bounds, express them in the type, and let the type-checker verify them. With inference some of this work may be automated, although the tradeoff between automation and precision of error messages will have to be carefully weighed. Finally, prior work has explored a temporal linear type system for parallel complexity analysis [16] 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.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. mmmmmm and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References


We present several different kinds of example from varying domains illustrating different features of the type system and algorithms. To transition from abstract syntax to the concrete syntax from the implementation, we use Table 4 as a guide. In addition, types and processes are declared and defined as follows.

\[
\begin{align*}
\text{Abstract Syntax} & \quad \text{Concrete Syntax} \\
\emptyset \{l_1 : A_1, l_2 : A_2\} & \quad \{\mathit{l_1} : A_1, \mathit{l_2} : A_2\} \\
\& \{l_1 : A_1, l_2 : A_2\} & \quad \{\mathit{l_1} : A_1, \mathit{l_2} : A_2\} \\
A \odot B & \quad A \ast B \\
A \rightarrow B & \quad A \rightarrow B \\
\exists n. A & \quad \mathit{\exists n. A} \\
\forall n. A & \quad \mathit{\forall n. A} \\
\mathit{?\{n = 0\}. A} & \quad \mathit{\{n = 0\}. A} \\
\mathit{!\{n = 0\}. A} & \quad \mathit{\{n = 0\}. A} \\
\mathit{\rightarrow A} & \quad \mathit{\rightarrow A} \\
V[n_1, n_2] & \quad V(n_1)(n_2) \\
x.k & \quad \mathit{x.k} \\
case x (l \Rightarrow P) & \quad \mathit{case x (l => P)} \\
send x w & \quad \mathit{send x w} \\
y \leftarrow \mathit{recv x} & \quad \mathit{y \leftarrow recv x} \\
close x & \quad \mathit{close x} \\
wait x & \quad \mathit{wait x} \\
send x \{e\} & \quad \mathit{send x \{e\}} \\
\{n\} \leftarrow \mathit{recv x} & \quad \{n\} \leftarrow \mathit{recv x} \\
\mathit{assert x \{n = 0\}} & \quad \mathit{assert x \{n = 0\}} \\
\mathit{assume x \{n = 0\}} & \quad \mathit{assume x \{n = 0\}} \\
\mathit{pay x \{r\}} & \quad \mathit{pay x \{r\}} \\
\mathit{get x \{r\}} & \quad \mathit{get x \{r\}} \\
\end{align*}
\]

Table 4. Abstract and Corresponding Concrete Syntax

### A Further Examples

We present several different kinds of example from varying domains illustrating different features of the type system and algorithms. To transition from abstract syntax to the concrete syntax from the implementation, we use Table 4 as a guide. In addition, types and processes are declared and defined as follows.

\[
\begin{align*}
\text{type } V(n) & = A \\
\text{type } V[n] & = \{\mathit{cons} : \mathit{nat} \ast \mathit{list}, \\
& \quad \mathit{nil} : 1\} \\
\end{align*}
\]

List is a purely positive type, so the provider of a list just sends messages. For example, ignoring constraints and potentials for now, the list 3, 4, 5 would be the following sequence of messages:

\[
\text{cons, } a_3, \text{ cons, } a_4, \text{ cons, } a_5, \text{ nil, close}
\]

Here, \(a_3, a_4, a_5\) are channels along which representations of the numbers 3, 4, and 5 are sent, respectively.

Next we index lists by their length \(n\) and assign a uniform potential \(p\) to every element. This potential is transferred to the client of the list before each element.

\[
\begin{align*}
\text{type list}(n)(p) & = \\
& \quad \{\mathit{cons} : ?\{n > 0\}. \{p\} > \mathit{nat} \ast \mathit{list}(n-1)(p), \\
& \quad \mathit{nil} : ?\{n = 0\}. 1\}
\end{align*}
\]

Now our example list where each element has potential 2, provided along an \(l : \mathit{list}[3,2]\) would send the following messages, if all implicit information were actually transmitted.

\[
\begin{align*}
\text{cons, assert } \{3 > 0\}, \text{ pay } 2, a_3, \text{ cons, assert } \{2 > 0\}, \text{ pay } 2, a_4, \\
\text{cons, assert } \{1 > 0\}, \text{ pay } 2, a_5, \text{ nil, assert } \{0 = 0\}, \text{ close}
\end{align*}
\]

### Nil and Cons Processes

Next we would like to program a process \(\mathit{nil}\) that sends the messages for an empty list. \(\mathit{nil}\) does not have to send any potential, according to the type, but it must send two messages: \(\mathit{nil}\) and close. Since our cost model assigns unit cost to every send operation, the \(\mathit{nil}\) process must carry a potential of 2. Nevertheless, it can be at the end of a list of any potential \(p\).

\[
\begin{align*}
\text{proc } l & \leftarrow \mathit{nil}(p) \leftarrow = \mathit{l.nil} ; \text{ close } l \\
\text{Reconstruction will add the assertion of } \theta = 0 \text{ to this implicit code. It will also add the necessary work before every send operation, according to our cost model. We will show the reconstruction only for this example.}
\end{align*}
\]

\[
\begin{align*}
\text{proc } l & \leftarrow \mathit{nil}(p) \leftarrow = \mathit{work} ; \\
& \quad \mathit{l.nil} ; \\
& \quad \mathit{assert } l \{\theta = 0\} ; \\
& \quad \mathit{work} ; \\
& \quad \text{close } l
\end{align*}
\]

A \(\mathit{cons}\) process should take an element \(x\) and a list \(t\) and provide the list that sends \(\mathit{cons}\), then \(x\), and then behaves like \(t\). This time, let’s first examine the code before the type.

\[
\begin{align*}
\text{proc } l & \leftarrow \mathit{cons}(n)(p) \leftarrow = x \leftarrow t = \\
& \quad l.\mathit{cons} ; \\
& \quad \mathit{send } l \{x\} ;
\end{align*}
\]
1 <- t

Regarding typing, we see that if \( t \) is a list of length \( n \), then \( l \) should be a list of length \( n + 1 \). How much potential does cons need? We need 2 units to send cons and the element \( x \), but we also need \( p \) units because we are constructing a list where each element has potential \( p \). So, overall, cons requires potential \( p + 2 \). Putting these together, we get

\[
\text{decl cons}(n)(p) : \quad (x : \text{nat}) \ (t : \text{list}(n)(p)) \ |(p+2)- (1 : \text{list}(n+1)(p))
\]

If we make a mistake, for example, state the potential as \( p \) instead of \( p + 1 \), we get an error message:

\[
\text{error: insufficient potential: true} \\ | /= p+1-1 >= 1
\]

\[
\text{proc} \ l \ \leftarrow \ \text{cons}(n)(p) \ \leftarrow \ x \ \leftarrow \ t = \\
\begin{array}{l}
1.\text{cons} ; \text{send} \ l \ x ; \ l \ \leftarrow \ t \\
\end{array}
\]

which pinpoints the source of the error: we have insufficient potential to send the element \( x \).

### Appending Two Lists

For the append process, let’s start again with the code. We receive the lists along \( l_1 \) and \( l_2 \) and send the result of appending them along \( l \). We branch on \( l_1 \). If it is the label cons, we receive the element \( x \), then we send on cons and \( x \) along \( l \) and recurse. When recursing, the length of \( l_1 \) (which is \( n \)) is reduced by one, while the length of \( l_2 \) (which is \( k \)) stays the same. When \( l_1 \) is empty (we receive nil), we wait for \( l_1 \) to finish and then forward \( l_2 \) to \( l \).

\[
\text{proc} \ l \ \leftarrow \ \text{append}(n)(k)(p) \ \leftarrow \ l_1 \ l_2 = \\
\begin{array}{l}
\begin{cases}
\text{case} \ l_1 \\
\begin{cases}
\text{cons} \Rightarrow \ x \ \leftarrow \ \text{recv} \ l_1 ; \\
1.\text{cons} ; \text{send} \ l \ x ; \ l \ \leftarrow \ t \\
\end{cases}
\end{cases}
\end{array}
\]

Considering the parallelism inherent in this process, we see that it implements a pipeline from \( l_1 \) to \( l \) with a constant delay. The list \( l_2 \) can be computed in parallel with this pipeline and eventually is connected directly to the end of \( l_1 \).

It remains to reason about the potential. We need two units of potential to send cons and \( x \), and this for each element of \( l_1 \). Forwarding as needed in the second branch is cost-free, so no additional potential is needed. Therefore we obtain the type

\[
\text{decl append}(n)(k)(p) : \quad (l_1 : \text{list}(n)(p+2)) \ (l_2 : \text{list}(k)(p)) \ |- \\
(1 : \text{list}(n+k)(p))
\]

that is, each element of \( l_1 \) must have a potential \( p + 2 \) and each element of \( l_2 \) only potential \( p \). Also, the length of the output list is \( n + k \).

If we want a more symmetric form of append where all three lists carry potential \( p \) per element, we can “pre-pay” the cost of sending the cons and \( x \) messages for each element of \( l_1 \). Since \( l_1 \) has length \( n \), this means the append process requires \( 2n \) units of potential.

\[
\text{decl append}(n)(k)(p) : \\
(l_1 : \text{list}(n)(p+2)) \ (l_2 : \text{list}(k)(p)) \ |(2*n)- \\
(1 : \text{list}(n+k)(p))
\]

Note that this type can be assigned to exactly the same program as the first one: it is our choice if we want to require that \( l_1 \) carry the potential or that the process invoking append pre-pay the total cost when invoking append.

### Reversing a List

The process rev for reversing a list is quite similar to append. It uses an accumulator \( a \) to which it sends all the elements from the incoming list \( l \). When \( l \) is empty, the reversed list \( r \) is just the accumulator. The work analysis is analogous to append, so we only show the result.

\[
\text{decl rev}(n)(k)(p) : \\
(1 : \text{list}(n)(p+2)) \ (a : \text{list}(k)(p)) \ |- \\
(r : \text{list}(n+k)(p))
\]

\[
\text{proc} \ r \ \leftarrow \ \text{rev}(n)(k)(p) \ \leftarrow \ a = \\
\begin{array}{l}
\begin{cases}
\text{case} \ l \ (\text{cons} \Rightarrow \ x \ \leftarrow \ \text{recv} \ l ) \\
\begin{cases}
\text{a} \ \leftarrow \ \text{cons}(k)(p) \ \leftarrow \ x \ a ; \\
r \ \leftarrow \ \text{rev}(n-1)(k+1)(p) \ \leftarrow \ a ; \\
\text{nil} \ \rightarrow \ \text{wait} \ l ; \ l \ \leftarrow \ t \\
\end{cases}
\end{cases}
\end{array}
\]

Even though similar in work to append, its concurrent behavior is quite different. I cannot send an element along the output \( r \) until the whole input list \( l \) has been processed.

To just reverse a list we have to create an empty accumulator to start, which requires 2 units of potential, just once for the whole list.

\[
\text{decl reverse}(n)(p) : \ (1 : \text{list}(n)(p+2)) \ |(2)- \\
(r : \text{list}(n)(p))
\]

\[
\text{proc} \ r \ \leftarrow \ \text{reverse}(n)(p) \ \leftarrow \ l = \\
a \ \leftarrow \ \text{nil}(p) \ \leftarrow ; \\
r \ \leftarrow \ \text{rev}(n)(0)(p) \ \leftarrow \ l a
\]

Again, we could assign a different potential if we would be willing to prepay for operations instead of expecting the necessary potential to be stored with the elements of the input list.

### Recharging Potential

We can also “recharge” the potential of a list by adding 2 units to every element. For a list of length \( n \), this requires up-front potential of \( 4n + 2 \): a portion \( 2n \) goes to boost the potential of each element, another portion \( 2n \) goes to actually send each cons label and element \( x \) from the input list, and 2 units go to sending nil and close.

\[
\text{decl charge2}(n)(p) : \\
(k : \text{list}(n)(p)) \ |(4*n+2)- (1 : \text{list}(n)(p+2))
\]

\[
\text{proc} \ l \ \leftarrow \ \text{charge2}(n)(p) \ \leftarrow \ l = \\
\begin{array}{l}
\begin{cases}
\text{case} \ k \ (\text{cons} \Rightarrow \ x \ \leftarrow \ \text{recv} \ k ) = \\
1.\text{cons} ; \text{send} \ l \ x ; \\
\end{cases}
\end{array}
\]
We see there that for each element, map

While this requires a lot of work, its parallel complexity is good since it is a pipeline with a constant delay between input and output.

It would also be correct to recharge every element of the list with $q$ units of potential.

decl charge2(n){p}(q) :

This, unfortunately, requires the nonlinear arithmetic expression $(q + 2)n + 2$. While a simple solver for polynomial constraint could handle this, we currently reject this as nonlinear.

**Map** In a functional setting, mapping a function $f$ over a list of length $n$ requires $n$ uses of $f$ and is therefore not linear. However, reuse can be replaced by recursion. In this example, a mapper from type $A$ to type $B$ is a process that provides the choice between two labels, next and done. When receiving next it then receives an element of type $A$, responds with an element of type $B$ and recurses to wait for the next label. If it receives done, it terminates. We have

mapper$_{AB} = \& \{ \text{next} : A \to B \otimes \text{mapper}_{AB}, \text{done} : 1 \}$

Here, we use nat for both $A$ and $B$. We further assume for simplicity that the mapper has enough internal potential so it does not require any potential from the map process. We then have

proc l <- map(n)(p) <- k m =

We see there that for each element, map needs to send 4 messages (next, x, cons, and y) so the input list should have potential $p + 4$. There is also a constant overhead of 3 for the empty list (2 for nil and close, and 1 to notify the mapper that we are done).

We see there that for each element, map needs to send 4 messages (next, x, cons, and y) so the input list should have potential $p + 4$. There is also a constant overhead of 3 for the empty list (2 for nil and close, and 1 to notify the mapper that we are done).

decl map(n){p} :

Folding a list can be done in a similar fashion.

**Filter** Filtering elements from an input list is interesting because we cannot statically predict the length of the output list. So for the first time in the list examples we require a quantifier. However, we know that the output list is not longer than the input list so we define a new type bdd_list[n,p] = $\exists m. \ ?(m \leq n). \ list[m,p]$.

type bdd_list(n){p} = ?m. ?(m <= n). \ list[m]{p}

Even in implicit form, this type requires communication of the witness $m$. While not strictly needed, it is helpful to define bounded versions of nil and cons which have the same ergometric properties (definitions elided). More interesting is the cost-free bdd_resize which we can use to inform the type checker that a list bounded by $n$ is also bounded by $n + 1$: the type checker just has to verify that $m \leq n$ implies $m \leq n + 1$.

decl bdd_nil{p} : . |{2}- (l : bdd_list(){}p))
decl bdd_cons{n}{p} :

A selector responds false for elements to be excluded from the result, and it responds true and then returns the element itself so it can be included in the output list. Either way, it recurses so the next element can be tested.

A selector responds false for elements to be excluded from the result, and it responds true and then returns the element itself so it can be included in the output list. Either way, it recurses so the next element can be tested.

selector$_A = \& \{ \text{next} : A \wedge (\{ \text{false} : \text{selector}_A,\

done : 1 \})$.

Concretely (using nat for $A$):

type selector =

Then the filter process has the type and definition as described in Figure 5.

**A Queue as Two Lists** Finally, we return to the queue we used earlier as a running example. An efficient functional implementation uses two lists to implement the queue. Actually, its efficiency is debatable if used non-linearly [34], but here the type checker establishes constant-time amortized cost for enqueueing and dequeueing messages.

The algorithm is straightforward: the queue process maintains two lists, in and out. When elements are enqueued, they are put into the in queue, where each element has a potential of 4. When elements are dequeued, we take them from the out list where each element has a potential of 2, which is enough to send back the element to the client. When the output list is empty when a dequeue request is received, we first reverse the input list and make it the output list. This reversal costs 2 units of potential for each element, but we have stored 4 so this is sufficient.
decl filter(n)(p) : (s : selector) (k : list(n)(p+4)) |{3}- (l : bdd_list(n)(p))
proc l <- filter(n)(p) <- s k =
case k ( cons => x <- recv k ; s.next ; send s x ;
  case s ( false => l' <- filter(n-1)(p) <- s k ;
    l <- bdd_resize(n-1)(p) <- l') |
  true => x' <- recv s ; l' <- filter(n-1)(p) <- s k ;
    l <- bdd_cons(n-1)(p) <- x' l' ) |
nil => wait k ; s.done ; wait s ;
l <- bdd_nil(p) <- )

Figure 5. Type and Definition of filter process

The amortized cost of an enqueue is therefore 6:2 to construct the new element of the input list, plus 4 because each element of the input list must have potential 4 to account for its later reversal. When calculating the cost of a dequeue we see it should be 4, leading us to the type

type queue(n) = &{ enq : <{6}| nat -> queue(n+1),
  deq : <{4}| deq_reply(n) }

type deq_reply(n) =
  { none : ?{n = 0}. 1,
    some : ?{n > 0}. nat * queue(n-1) }

Note that uses of $\triangleleft$ and $\triangleright$ which mean that the client has to transfer this potential, while in lists we used $\triangleright_p$ so the list provider paid the potential $p$.

We have two processes definitions, one with an output list, and one where we have noted that the output list is empty. Figure 6 shows the declarations and definitions.

To create a new queue from scratch we need 4 units of potential in order to create two empty lists.

decl queue_new : |{(4)}- (q : queue(0))
proc q <- queue_new <- =
in0 <- nil(4) <- ;
out0 <- nil(2) <- ;
q <- queue_lists(0)(0) <- in0 out0

A.2 Linear $\lambda$-Calculus

We demonstrate an implementation of the (untyped) linear $\lambda$-calculus, including evaluation, in which the index objects track the size of the expression. Type-checking verifies that the result of evaluating a linear $\lambda$-term is no larger than the original term. Our representation uses linear higher-order abstract syntax.

Ignoring issues of size for the moment, $\lambda$-calculus expressions are represented with the type $\text{exp}$, values are of type $\text{val}$.

\[
\begin{align*}
\text{exp} &= \oplus\{ \text{lam : exp -> exp}, \\
& \quad \text{app : exp \otimes exp} \}
\end{align*}
\]

\[
\text{val} = \oplus\{ \text{lam : exp -> exp} \}
\]

An expression is a process sending either the label lam or app. In case of lam it then expects a channel along which it receives the argument to the function and then behaves like the body of the $\lambda$-abstraction. In case of app it will send a channel along which the function part of an application will be sent and then behaves like the argument. While this kind of encoding may seem strange, it works miraculously.

For example, the representation of the identity function $\lambda x. x$ is

\[
\begin{align*}
decl \text{id} : \_ |\text{-} (e : \text{exp})
proc e <- id <- =
e.lam ; \\
x <- recv e ; \\
e <- x
\end{align*}
\]

The application of the identity function to itself:

\[
\begin{align*}
decl \text{idid} : \_ |\text{-} (e : \text{exp})
proc e <- idid <- =
i1 <- id <- ; \\
i2 <- id <- ; \\
e.app ; send e i1 ; e <- i2
\end{align*}
\]

Similar to the nil and cons processes for lists, we have two processes for constructing expressions and values.

\[
\begin{align*}
decl \text{apply} : (e1 : \text{exp}) (e2 : \text{exp}) |\text{-} (e : \text{exp})
proc e <- apply <- e1 e2 = 
e.app ; send e e1 ; e <- e2
\end{align*}
\]

\[
\begin{align*}
decl \text{lambda} : (f : \text{exp \circ exp}) |\text{-} (v : \text{val})
proc e <- lambda <- f = 
e.lam ; e <- f
\end{align*}
\]

Evaluation of a $\lambda$-abstraction immediately returns it as a value. For an application $e_1 e_2$ we first evaluate the function. By typing, this value is a process that will send the lam label and then expects argument expression as a message. So we send $e_2$. 

Session Types with Arithmetic Refinements

PL’18, January 01–03, 2018, New York, NY, USA

decl queue_lists(n1){n2} : (in : list{n1}{4}) (out : list{n2}{2}) |- (q : queue{n1+n2})

decl queue_rev{n1} : (in : list{n1}{4}) |{4}|- (q : deq_reply{n1})

proc q <- queue_lists{n1}{n2} <- in out =
  case q ( enq => x <- recv q ;
       in' <- cons{n1}{4} <- x in ;
       q <- queue_lists(n1+1){n2} <- in' out
   | deq => case out ( cons => x <- recv out ;
       q.some ; send q x ;
       q <- queue_lists{n1}{n2-1} <- in out
   | nil => wait out ;
       q <- queue_rev{n1} <- in ) )

proc q <- queue_rev{n1} <- in =
  out <- reverse{n1}{2} <- in ;
  case out ( cons => x <- recv out ;
       q.some ; send q x ;
       in0 <- nil{4} <- ;
       q <- queue_lists{0}{n1-1} <- in0 out
   | nil => wait out ;
       q.none ; close q )

Figure 6. Queue as 2 Lists

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 )))

The trickiest part here is the line marked with the comment % e = e2. An e representing e1 e2 will first send e1 : exp and then behave like e2. So what we send to v1 two lines below in fact represents e2.

To track the sizes of λ-expressions we need quantifiers. For example, an application e1 e2 has size n if there exist sizes n1 and n2 such that n = n1 + n2 + 1 and of e1 and e2 have size n1 and n2, respectively.

The case of λ-abstractions is more complicated. The issue is that we cannot assign a fixed size (say 1) to the variable in a λ-abstraction because we may apply it to an argument of any size. Instead, the body can receive an arbitrary size n1 and then an expression of size n1.

These considerations lead us to the types

\[
\text{exp}[n] = \\
\forall \lambda \forall n > 0. \forall n1. \text{exp}[n1] \Rightarrow \text{exp}[n1 + n - 1].
\]

\[
\text{app} : \forall n1, \forall n2. \{ n = n1 + n2 + 1 \}. \text{exp}[n1] \otimes \text{exp}[n2] \}
\]

\[
\text{val}[n] = \\
\forall \lambda \forall n > 0. \forall n1. \text{exp}[n1] \Rightarrow \text{exp}[n1 + n - 1]}
\]

The types of apply, lambda, and eval change accordingly
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})
decl eval{n} :
  (e : exp{n}) |- (v : ?k. ?{k \leq n}. val{k})

Interesting here is only that the result type of evaluation contains an existential quantifier since we do not know the precise size of the value—we just know it is bounded by n. Side calculation shows that every reduction that takes place during evaluation reduces the size of the output value by 2 since both the λ-abstraction and the application disappear. The eval process is implemented in Figure 7.

A.3 Binary Numbers

We can represent binary numbers as sequences of bits b0 and b1 followed by e to indicate the end of the list. The representation function \( \overline{n} \) is given by

\[
\overline{0} = e, \text{close}
\]

\[
\overline{2n} = b0, \overline{n} \quad n > 0
\]

\[
\overline{2n+1} = b1, \overline{n}
\]

The restriction on \( n > 0 \) in the second line ensures uniqueness of representation (no leading zeros). For the operations on them it is important that the representation be in little endian for, that is, the least significant bit comes first.

Ignoring indexing for now, we would represent this as the type
proc x <- zero <- = x.e ; close x
proc y <- succ(n) <- x =

Figure 7. Evaluating expressions in linear $\lambda$-calculus

As can be seen from its type, type checking verifies that, for example, succ implements the successor function. It is plausible that we could omit the sending a receiving of $n'$, collect constraints in a manner analogous to the other implicit constructs, and check their validity, making this program even more compact. However, two problems arise. The first is that the quality of the error messages degrades because first we need to collect all constraints and then solve them together and it is no longer clear were the error might have arisen. And secondly we can no longer perform reconstruction in general because in $\forall n. \exists k. \phi$ propositions of Presburger arithmetic the Skolem function that computes $k$ given $n$ may not be expressible. This means we would have to add something like Hilbert’s $\epsilon$ operator or definite description $\iota$, which is an interesting item for future work. So far in our examples the benefits of good error messages have outweighed the few additional messages that are sometimes required for quantified types.

Domain Restrictions  We would like to apply the predecessor process only to strictly positive numbers so we don’t have to worry about computing 0 − 1. We can express this with an explicit constraint in process typing, which we have omitted in the body of the paper for the sake of brevity. This is quite similar to standard idioms in Liquid types [25, 38].
decl pred\(\{\text{n\mid n > 0}\}\) : (\(x : \text{bin\{\text{n}\}}\)) |- (\(y : \text{bin\{\text{n-1}\}}\))
proc \(y \leftarrow \text{pred\{\text{n}\}} \leftarrow x\) =
case \(x\)
  ( \(\text{b0 \Rightarrow \{\text{n'}\}} \leftarrow \text{recv} \ x\) ; \(\% 2^{\text{k-1}} = 2^{\text{(k-1)+1}}\)
    \(y.\text{b1} ; \text{send} \ y \ \{\text{n'-1}\}\) ;
    \(y \leftarrow \text{pred\{\text{n'}\}} \leftarrow x\)
  | \(\text{b1 \Rightarrow \{\text{n'}\}} \leftarrow \text{recv} \ x\) ;
    \(y \leftarrow \text{pred\{\text{n'}\}} \leftarrow x\) ; \(\% 2^{\text{k+1}-1} = 2^{\text{k}}\)

This is an example where the case for \(e\) is impossible (since \(n > 0\) is incompatible with \(n = 0\)) so we omit this branch and it is reconstructed as shown in Section 6.

Here \(\text{dbl0}\) is a process with type

decl \(\text{dbl0}\{\text{n}\}\) : (\(x : \text{bin\{\text{n}\}}\)) |- (\(y : \text{bin\{2\text{n}\}}\))
which is not as trivial as one might think. In fact, if we try

proc \(y \leftarrow \text{dbl0\{\text{n}\}} \leftarrow x\) =
  \(y.\text{b0} ; \text{send} \ y \ \{\text{n}\}\) ;
we get the error message

error: assertion not entailed: true \(\%/= 2^\text{n} > 0\)
  \(y.\text{b0} ; \text{send} \ y \ \{\text{n}\}\) ;

which means that this function is not correct because we cannot prove that \(2n > 0\) because we don’t know if \(n > 0\), possibly introduce leading zeros! The correct code requires one bit of look-ahead on the input channel \(x\). Computing \(2n + 1\) is much simpler.

proc \(y \leftarrow \text{dbl0\{\text{n}\}} \leftarrow x\) =
case \(x\)
  ( \(\text{b0 \Rightarrow \{\text{n'}\}} \leftarrow \text{recv} \ x\) ;
    \(y.\text{b0} ; \text{send} \ y \ \{\text{n}\}\) ;
    \(y \leftarrow \text{dbl0\{\text{n'}\}} \leftarrow x\)
  | \(\text{b1 \Rightarrow \{\text{n'}\}} \leftarrow \text{recv} \ x\) ;
    \(y.\text{b1} ; \text{send} \ y \ \{\text{n}\}\) ;
    \(y \leftarrow \text{dbl0\{\text{n'}\}} \leftarrow x\)
  | \(\text{e \Rightarrow y.e} ; \text{wait} \ x \ ; \text{close} \ y\) )

Discarding and Copying Numbers Even though we are in a purely linear language, message sequences of purely positive type can be dropped or duplicated by explicit programs. It is plausible we could build this into the language as a derivable idiom, maybe along the lines of equality types or type classes in languages like Standard ML or Haskell. However, this has not been a high priority, so here are the processes that consume a binary number without producing output, or duplicating a binary number.

decl \(\text{drop}\{\text{n}\}\) : (\(x : \text{bin\{\text{n}\}}\)) |- (\(\text{u : 1}\))
proc \(u \leftarrow \text{drop\{\text{n}\}} \leftarrow x\) =
case \(x\)
  ( \(\text{b0 \Rightarrow \{\text{n'}\}} \leftarrow \text{recv} \ x\) ; \(u \leftarrow \text{drop\{\text{n'}\}} \leftarrow x\)
    \(\% \text{no case for e}\) )

decl \(\text{dup}\{\text{n}\}\) :
(x : bin\{n\}) |- (xx : bin\{2*n\} * bin\{n\} * 1)
proc xx <- \(\text{dup\{\text{n}\}} \leftarrow x\) =
case xx
  ( \(\text{b0 \Rightarrow \{\text{n'}\}} \leftarrow \text{recv} \ x\) ;
    xx \(\leftarrow \text{dup\{\text{n'}\}} \leftarrow x\) ;
    xx \(\leftarrow \text{recv} xx\) ;
    xx \(\leftarrow \text{recv} xx\) ; \text{wait} xx\) ;
    xx \(\leftarrow \text{dbl0\{\text{n'}\}} \leftarrow x1\) ; \text{send} xx x1 ;
    xx \(\leftarrow \text{dbl0\{\text{n'}\}} \leftarrow x2\) ; \text{send} xx x2 ;
    \text{close} xx
  | \(\text{b1 \Rightarrow \{\text{n'}\}} \leftarrow \text{recv} \ x\) ;
    xx \(\leftarrow \text{dup\{\text{n'}\}} \leftarrow x\) ;
    xx \(\leftarrow \text{recv} xx\) ; \text{wait} xx\) ;
    xx \(\leftarrow \text{dbl1\{\text{n'}\}} \leftarrow x1\) ; \text{send} xx x1 ;
    xx \(\leftarrow \text{dbl1\{\text{n'}\}} \leftarrow x2\) ; \text{send} xx x2 ;
    \text{close} xx
  | \(\text{e \Rightarrow x} \leftarrow \text{zero} \leftarrow ; \text{send} xx x1\) ;
    xx \(\leftarrow \text{zero} \leftarrow ; \text{send} xx x2\) ;
    \text{close} xx
)

Note that the correctness of duplication (we get two copies of the number \(n\)) is expressed in the indexed type and verified by the type checker.

Comparisons Comparison is one place where we need to discard part of a binary number because once the shorter number has been exhausted, we need to consume the remaining bits of the longer number. For comparison, it is very helpful to have a representation that enforces the absence of leading zeros. But how do we express the outcome and verify the correctness? For this purpose we define a new type, \(\text{ord}\{m, n\}\) to represent the outcome of the comparison of the representations of \(m\) and \(n\).

\[
\text{type ord}\{m, n\} = \{ \text{lt} : \{m < n\}, \text{eq} : \{m = n\}, \text{gt} : \{m > n\}\}
\]

Then the type

decl \(\text{compare}\{m\}{n}\) :\n(x : bin\{m\}) (y : bin\{n\}) |- (\(o : \text{ord}\{m, n\}\))
guarantees the correctness of compare because it can produce \(o\) only if \(m < n\), and similarly for \text{eq} and \text{gt}. The code is somewhat tedious, because we do not have nested pattern
matching, but we show the beginning of the process definition because it brings up an interesting fact regarding the type checker.

\[
\text{proc } o \leftarrow \text{compare}(m) \leftarrow x \ y = \\
\begin{cases}
\text{case } x & \leftarrow \text{recv } x ; \\
\leftarrow \text{recv } y ;
\end{cases}
\]

For this recursive call to type-check we have to make sure that the outcome of \(\text{compare}(m, n)\) (of type \(\text{ord}(m, n)\)) is the same as the outcome of \(\text{compare}(m', n')\) (which is of type \(\text{ord}(m', n')\)). This requires that

\[
m, n, m', n';  \\
m' > 0 \land m = 2 \ast m' \land n' > 0 \land n = 2 \ast n';  \\
\text{ord}(m, n) \equiv \text{ord}(m', n')
\]

This is certainly true declaratively (according to the definition of type bisimulation) because \(2 \ast m' < 2 \ast n'\) if \(m' < n'\) and similarly for equality and greater-than comparison. However, it goes beyond reflexivity of indexed types and the generality of the def rule in the type equality algorithm (see Section 4) is needed in this and other examples.

We have other processes, such as multiplication (which require nonlinear constraints and utilizes the dup process) as well as conversion to and from the unary representation of numbers. Unfortunately we cannot easily characterize the work of many of these operations because they depend on the number of digits in the binary representation, which is logarithmic in its value. This source of nonlinearity makes it difficult to track work unless we consider numbers of a fixed width (such as 32-bit or 64-bit numbers). However, then we would be working in modular arithmetic and have to overcome the lack of div and mod function in Presburger arithmetic.

### A.4 Prime Sieve

As a slightly more complex example we use an implementation prime sieve. It turns out to be convenient to use an implementation of unary natural numbers, which we elide, which satisfies the following signature.

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

\[
\text{decl } \text{zero} : . |- (x : \text{nat}(0))  \\
\text{decl } \text{suc}c(n) : (x : \text{nat}(n)) |- (y : \text{nat}(n+1))  \\
\text{decl } \text{drop}(n) : (x : \text{nat}(n)) |- (u : \text{1})  \\
\text{decl } \text{dup}(n) :  \\
\begin{cases}
(x : \text{nat}(n)) |- (xx : \text{nat}(n) \ast \text{nat}(n) \ast \text{1})
\end{cases}
\]

The output of the prime sieve is a stream of bits (which we call \text{prime} and \text{composite}) of length \(k\) giving the status of the numbers \(2, 3, 4, 5, 6, \ldots, \ k + 2\). It would start with \text{prime}, \text{prime}, \text{composite}, \text{prime}, \text{composite}, \ldots

\[
\text{type } \text{stream}(k) = +(\text{prime} : ?(k > 0). \text{stream}(k-1),  \\
\text{composite} : ?(k > 0). \text{stream}(k-1),  \\
\text{end} : ?(k = 0). \text{1})
\]

The underlying algorithmic idea is to set up a chain of filters that mark multiples of all the prime number we produce as composite. So when the output stream has proclaimed \(p\) numbers to be prime, there will be \(ap + b\) active processes as small constants \(a\) and \(b\). This algorithm and its implementation has a good amount of parallelism since all the filters can operate concurrently. Moreover, we do not need to perform any multiplication or division, we just set up some cyclic counters along the way.

At one end of the chain is a process candidates that produces a sequence of bits \text{prime}, because initially all natural numbers starting at 2 are candidates to be primes.

\[
\text{decl } \text{candidates}(n) : (x : \text{nat}(n)) |- (s : \text{stream}(n))  \\
\text{proc } s \leftarrow \text{candidates}(n) \leftarrow x = \\
\begin{cases}
\text{case } x & (\text{succ}c => s.\text{prime} ;  \\
\text{zero} => \text{wait } x ;  \\
\text{s.end} ; \text{close } s)
\end{cases}
\]

The candidates are fed into a process head that remains at the other end of the stream, after all multiples of primes so far have been marked as \text{composite}. The bit \text{prime} it sees must indeed be a prime number, so it sets up a filter for multiples of the current list index (which is \(x : \text{nat}\)). If the number is composite, that information is simply passed on.

Two interesting aspects of this process are that the current list index must be duplicated to set up the filter which contains a cyclic counter. The other is that the sum \(n \ast k\) of the current list index \(n\) and the length of the remaining stream \(k\) must be invariant. We express this by passing in a redundant argument, constrained to be equal to the sum which never changes. This would catch errors if we incorrectly forgot to increment the list index.

\[
\text{decl } \text{head}(k)(n)\{\text{kn}|\text{kn} = k+n\} : \\
\begin{cases}
\text{t} : \text{stream}(k) & (x : \text{nat}(n)) |- (s : \text{stream}(k))  \\
\text{proc } s \leftarrow \text{head}(k)(n) \leftarrow t \ x = \\
\text{case } t & (\text{prime} => s.\text{prime} ;  \\
\text{z} \leftarrow \text{zero} <= ;  \\
\text{xx} \leftarrow \text{dup}(n) \leftarrow x ;  \\
\text{x}_1 \leftarrow \text{recv } xx ;  \\
\text{x}_2 \leftarrow \text{recv } xx ;  \\
\text{wait } xx ;  \\
f \leftarrow \text{filter}(k-1)(n)\{\text{kn}|\text{kn} = k+n\} \leftarrow t \ x_1 \ z ;  \\
x' \leftarrow \text{suc}c(n) \leftarrow x_2 ;
\end{cases}
\]

The bits output of the prime sieve are as follows:

\[
\begin{array}{cccccccc}
2 & 3 & 5 & 7 & 11 & 13 & 17 & 19 \\
\text{0} & 1 & 1 & 1 & 0 & 0 & 1 & 1
\end{array}
\]
Finally, the filter process filter itself. It implements a cyclic counter as two natural numbers and whose sum remains invariant. When c reaches 0 we know we have reach a multiple of the original d + 1, we mark the position in the stream as composite and reinitialize the counter c with the value of d and set d to 0. When the stream is empty (implying k = 0) we have to dealloc the counters. Figure 8 shows the process declaration and definition.

A.5 Trie

We illustrate the data structure of a trie to maintain multisets of natural numbers. Remarkably, the data structure is linear even in one case we have to deallocate an integer. There is a fair amount of parallelism since consecutive requests to insert numbers into the trie can be carried out concurrently. We also obtain a good characterization of the necessary work—the data structure is quite efficient (in theoretical terms).

A Binary Counter We start with a binary counter that can receive inc messages to increment the counter and val to retrieve the current value of the counter. For the latter, we use binary numbers where each bit carries potential p. When retrieving the number each bit carries potential 0, for simplicity.

type bin(n)(p) =
  +{ b0 : ?(n > 0). ?k. ?(n = 2\times k). (p)\times bin(k)(p),
  b1 : ?(n > 0). ?k. ?(n = 2\times k+1). (p)\times bin(k)(p),
  e : ?(n = 0). 1 }

type ctr(n) =
  &[ inc : <{3}| ctr(n+1),
  val : <{2}| bin(n)(0) ]
  % potential 0 here for simplicity

An increment of the binary counter has 3 units of amortized cost. 1 of these units accounts for a future carry, and 2 units account for the cost of converting the counter to a sequence of bits should the value be required. Note that amortized analysis is necessary here, because in the worst case a single increment message could require \log(n) carry bits to be propagated through the number.

The implementation of the counter is similar to the implementation of a queue. It is a chain of processes bit0 and bit1, each holding a single bit, terminated by an empty process. When converted to a message sequence of bits of type bin, bit0 corresponds to b0, bit1 to b1, and empty to e.

\newpage

\begin{figure}
\begin{center}
\begin{verbatim}
s <- head(k-1){n+1}{k} <- f x'
  | composite => s.composite ;
  x' <- succ(n) <- x ;
  s <- head(k-1){n+1}{k} <- t x'
  | end => wait t ;
  u <- drop(n) <- x ;
  wait u ;
s.end ; close s )
\end{verbatim}
\end{center}
\caption{A Binary Counter}
\end{figure}

\begin{figure}
\begin{center}
\begin{verbatim}
\textbf{proc} c <- bit0{n} <- d =
  case c ( inc => c <- bit0{n} <- d ;
    | val => c.b0 ; send c {n} ;
      d.val ; c <- d )
\textbf{proc} c <- bit1{n} <- d =
  case c ( inc => d <- c ;
    | val => c.b1 ; send c {n} ;
      d.val ; c <- d )
\end{verbatim}
\end{center}
\caption{The Trie Interface}
\end{figure}

The type-checker was useful in this example because it discovered a bug in an earlier, unindexed version of the code: bit0[n] must require n > 0, otherwise its response to a value request might have leading zeros. We also see that each bit0 process carries a potential of 2 (to process a future val request) and each bit1 process carries a potential of 3 (to also be able to pass on a carry as an inc message when incremented).

The Trie Interface A trie is represented by the type trie[n] where n is the number of elements in the current multiset. When inserting a number it becomes a trie trie[n+1]. When we delete a number x from the trie we delete all copies of x and return its multiplicity. For example, if x was not in the trie at all, we respond with 0; if x was added to the trie 3 times we respond with 3. If m is the multiplicity of the number, then after deletion the trie will have trie[n−m] elements. This requires the constraint that m ≤ n: the multiplicity of an element cannot be greater than the total number of elements in the multiset.

When inserting a binary number into the trie that number can be of any value. Therefore, we must pass the index k representing that value, which is represented by a universal quantifier in the type. Conversely, when responding we need to return the unique binary number m which is of course not known statically and therefore is an existential quantifier.

The way we insert the binary number is starting at the root with the least significant bit and recursively insert the number into the left or right subtree, depending on whether the bit is b0 or b1. When we reach the end of the sequence of bits (e) we increase the multiplicity at the leaf we have
decl filter{k}{n1}{n2}{n} : (t : stream{k}) (c : nat{n1}) (d : nat{n2}) |- (s : stream{k})
proc s <- filter{k}{n1}{n2}{n} <- t c d =
case t ( prime => case c ( succ => s.prime ; % not divisible by n
    d' <- succ{n2} <- d ;
    s <- filter{k-1}{n1-1}{n2+1}{n} <- t c d' ) % cyclic loop
| composite => s.composite ; % already composite
    case c ( succ => d'
        <- succ{n2} <- d ;
        s <- filter{k-1}{n1-1}{n2+1}{n} <- t c d' )
    | zero => wait c ; % divisible by n: not prime
    s.composite ;
    z <- zero <- ;
    s <- filter{k-1}{n2}{0}{n} <- t d z ) % cyclic loop
| composite => s.composite ; % already composite
    case c ( succ => d'
        <- succ{n2} <- d ;
        s <- filter{k-1}{n1-1}{n2+1}{n} <- t c d' )
    | zero => wait c ;
    z <- zero <- ;
    s <- filter{k-1}{n2}{0}{n} <- t d z )
| end => wait t ;
u <- drop{n1} <- c ; wait u ;
u <- drop{n2} <- d ; wait u ;
s.end ; close s )

Figure 8. Filtering prime numbers

reached. This relies on the uniques of representation, that is,
our binary numbers may not have any leading zeros (which
is enforced by the indices, as explained in an earlier exam-
ple in Appendix A.3). As we traverse the trie, we need to
construct new intermediate nodes in case we encounter a
leaf. It turns out these operations require 4 messages per bit,
so the input number should have potential of 4 per bit. For
deletion, it turns out we need one more because we need to
communicate the answer back to the client, so 5 units per
bit. For simplicity, we therefore uniformly require 5 units
of potential per bit when adding a number to the trie and
"burn" the extra unit during insertion.

trie{n} =
&{ ins : 4\forall k. bin[k,5] -o trie{n+1},
    del : 4\forall k. bin[k,5] -o \exists m. ?{m <= n},
    bin[m,0] \otimes trie{n-m} }

Trie Implementation We have two kinds of nodes: leaf
nodes (process leaf{0}) not holding any elements and element
nodes (process node{n0, m, n1}) representing an element of
multiplicity m with n0 and n1 elements in the left and right
subtries, respectively. A node therefore has type trie{n0 +
m + n1}. Neither process carries any potential.

decl leaf : . |-( t : trie{0})
decl node(n0)(m)(n1) :
(1 : trie{n0}) (c : ctr{m}) (r : trie{n1}) |-

(t : trie{n0+m+n1})
The code then is a bit tricky in the details (nice to have a
type checker verifying both indices and potential!) but not
conceptually difficult. We present it here without further
comments.

proc t <- leaf <- =
case t
    ( ins => {k} <- recv t ;
        x <- recv t ;
        case x
            ( b0 => {k'} <- recv x ;
                l <- leaf <- ;
                c0 <- empty <- ;
                r <- leaf <- ;
                l.ins ; send l {k'} ; send l x ;
                t <- node(1){0}{0} <- l c0 r
            | b1 => {k'} <- recv x ;
                l <- leaf <- ;
                c0 <- empty <- ;
                r <- leaf <- ;
                r.ins ; send r {k'} ; send r x ;
                t <- node(0){0}{1} <- l c0 r
            | e => wait x ;
                l <- leaf <- ;
                c0 <- empty <- ;
                c0.inc ;
                r <- leaf <- ;
                t <- node(0){0}{1} <- l c0 r
        )
    | del => {k} <- recv t ;
        x <- recv t ;
        case x
            ( b0 => {k'} <- recv x ;
                l <- leaf <- ;
                c0 <- empty <- ;
                r <- leaf <- ;
                l.ins ; send l {k'} ; send l x ;
                t <- node(1){0}{0} <- l c0 r
            | b1 => {k'} <- recv x ;
                l <- leaf <- ;
                c0 <- empty <- ;
                r <- leaf <- ;
                r.ins ; send r {k'} ; send r x ;
                t <- node(0){0}{1} <- l c0 r
            | e => wait x ;
                l <- leaf <- ;
                c0 <- empty <- ;
                c0.inc ;
                r <- leaf <- ;
                t <- node(0){0}{1} <- l c0 r
        )
    | end => wait t ;
u <- drop{n1} <- c ; wait u ;
u <- drop{n2} <- d ; wait u ;
s.end ; close s )
u <- drop{k}{5} <- x ; wait u ;
send t {0} ;
c0 <- empty <- ;
c0.val ;
send t c0 ;
t <- leaf <-
)

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+1}{n1} <- l c r
      | e => wait x ;
        c.inc ;
        t <- node{n0}{m}{n1} <- l c r)
    | del => {k} <- recv t ;
      x <- recv t ;
      case x
        ( b0 => {k'} <- recv x ;
          l.del ; send l {k'} ; send l x ;
          {m1} <- recv l ;
          a <- recv l ; send t {m1} ;
          send t a ;
          t <- node{n0-m1}{m}{n1} <- l c r
        | b1 => {k'} <- recv x ;
          r.del ; send r {k'} ; send r x ;
          {m2} <- recv r ;
          a <- recv r ;
          send t {m2} ; send t a ;
          t <- node{n0}{m}{n1-m2} <- l c r
        | e => wait x ;
          send t {m} ;
          c.val ; send t c ;
          c0 <- empty <- ;
          t <- node{n0}{0}{n1} <- l c0 r
      )
  )
B Type Equality

Syntax The types follow the syntax below; \( n \) is a variable, and \( i \) is a constant.

Types
\[
A ::= \oplus \ell : A_\ell \mid \& \ell : A_\ell \mid A \otimes A \mid A \to A \mid 1 \mid V[e] \\
\text{Arith. Exp.} \quad e ::= i \mid e + e \mid e - e \mid i \times e \mid n \\
\text{Arith. Prop.} \quad \phi ::= e = e \mid e > e \mid \top \mid \bot \mid \phi \land \phi \\
\quad \phi \lor \phi \mid \neg \phi \mid \exists n. \phi \mid \forall n. \phi
\]

B.1 Type Equality Definition

Definition 5. For a type \( A \), define unfold\(_2\)(\( A \)) in the presence of signature \( \Sigma \):

\[
\text{unfold\(_2\)(V[\ell]) = A \in \Sigma} \quad \text{def} \quad A \neq V[e] \quad \text{unfold\(_2\)(A) = A} \quad \text{str}
\]

Definition 6. Let Type be the set of all ground session types. A relation \( R \subseteq \text{Type} \times \text{Type} \) is a type simulation if \((A, B) \in R\) implies the following conditions:
- If \( \text{unfold\(_2\)(A)} = \oplus \ell : A_\ell \), then \( \text{unfold\(_2\)(B)} = \oplus \ell : B_\ell \) and \((A_\ell, B_\ell) \in R\) for all \( \ell \) in \( L \).
- If \( \text{unfold\(_2\)(A)} = \& \ell : A_\ell \), then \( \text{unfold\(_2\)(B)} = \& \ell : B_\ell \) and \((A_\ell, B_\ell) \in R\) for all \( \ell \) in \( L \).
- If \( \text{unfold\(_2\)(A)} = A \to A \) then \( \text{unfold\(_2\)(B)} = B_1 \to B_2 \) and \((A_1, B_1) \in R\) and \((A_2, B_2) \in R\).
- If \( \text{unfold\(_2\)(A)} = A_1 \otimes A_2 \) then \( \text{unfold\(_2\)(B)} = B_1 \otimes B_2 \) and \((A_1, B_1) \in R\) and \((A_2, B_2) \in R\).
- If \( \text{unfold\(_2\)(A)} = 1 \), then \( \text{unfold\(_2\)(B)} = 1 \).
- If \( \text{unfold\(_2\)(A)} = ? \phi \). A', then \( \text{unfold\(_2\)(B)} = ? \psi \). B' and either \( \models \phi \to \psi \) and \( \models A' \to B' \in R \) or \( \models \neg \phi \) and \( \models \neg \psi \).
- If \( \text{unfold\(_2\)(A)} = ! \phi \) then \( \text{unfold\(_2\)(B)} = ! \psi \). and either \( \models \phi \to \psi \) and \( \models A' \to B' \in R \), or \( \models \neg \phi \) and \( \models \neg \psi \).
- If \( \text{unfold\(_2\)(A)} = \exists n. A' \) then \( \text{unfold\(_2\)(B)} = \exists n. B' \) and for all \( i \in \mathbb{N} \), \((A'[i/n], B'[i/n]) \in R\).
- If \( \text{unfold\(_2\)(A)} = \forall n. A' \) then \( \text{unfold\(_2\)(B)} = \forall n. B' \) and for all \( i \in \mathbb{N} \), \((A'[i/n], B'[i/n]) \in R\).

Definition 7. The type equality relation \( \equiv\_R \) is defined by \( A \equiv\_R B \) iff \( R \) is a type simulation and \((A, B) \in R\).

Definition 8. The coinductive type equality definition \( \equiv \) is defined by \( A \equiv B \) iff there exists a relation \( R \) such that \( A \equiv\_R B \).

Lemma 1. The relation \( \equiv \) is reflexive.

Proof. Consider the reflexive relation \( R_e \), such that \((A, A) \in R_e\) for any \( A \). To establish \( R_e \), as a type simulation, consider a representative case. If \( \text{unfold\(_2\)(A)} = ? \phi \). A', then the same holds and either \( \models \phi \to \phi \) and \((A'[i/n], A'[i/n]) \in R_e\), by definition of \( R_e \), or \( \models \neg \phi \to \neg \psi \). If \( \text{unfold\(_2\)(A)} = \exists n. A' \), then the same holds and \((A'[i/n], A'[i/n]) \in R_e\) by definition.

Lemma 2. The relation \( \equiv \) is symmetric.

Proof. Consider \( A \equiv B \). There exists a type simulation \( R_1 \) such that \((A, B) \in R_1\). Define \( R_2 \) as follows.

\[
R_2 = \{(B, A) \mid (A, B) \in R_1\}
\]

It is easy to see that \( R_2 \) is a type simulation since all invariants in Definition 6 are symmetric.

Lemma 3. The relation \( \equiv \) is transitive.

Proof. Suppose \( A_1 \equiv A_2 \) and \( A_2 \equiv A_3 \). Thus, there exist type simulations \( R_1 \) and \( R_2 \) such that \((A_1, A_2) \in R_1\) and \((A_2, A_3) \in R_2\). Denoting relational composition by \( R_1 \cdot R_2 \),

\[
R = (R_1 \cdot R_2)
\]

Clearly, \((A_1, A_3) \in R\). To establish \( R \) as a type simulation, consider \((A, C) \in R\) and the case \( A = ? \phi \). A'. Since \((A, B) \in R_1\) for some \( B \), we get that \( B = ? \phi \). B'. As a first case, assume \( \models \phi \) and \((A', B') \in R_1\). Similarly, since \((B, C) \in R_2\), we get that \( C = ? \phi \). C'. And since \( \models \phi \), we get that \( ? \phi \) and \((B', C') \in R_2\). Using the definition of \( R \), we get that \((A', C') \in R\). Thus, we have \( A = ? \phi \). A' and \( C = ? \phi \). C' and and \( \models \phi \) and \( \models \phi \) and \((A', C') \in R\). Therefore, \( R \) is a type simulation. A similar argument holds if \( \models \neg \phi \).

B.2 Type Equality Algorithm

We start with defining a judgment for defining type equality. The judgment is defined as \( \forall V ; C ; T \vdash A \equiv B. \Sigma \) defines the signature containing the type definitions. \( \forall V \) stores the list of free variables in \( A \) and \( B \), and \( \Sigma \) stores the constraints that the variables in \( \forall V \) satisfy. Finally, \( \Gamma \) stores the equality constraints that we are collecting while following the algorithm recursively. The algorithm is initiated with an empty \( \Gamma \).

We perform a simple type transformation that assigns a name to each intermediate type expression. After the transformation, the type grammar becomes:

\[
A ::= \oplus \ell : T \mid \& \ell : T \mid A \to T \mid T \otimes T \mid 1 \\
\quad \mid \top \mid \bot \mid \exists n. T \mid \forall n. T
\]

The type equality algorithm is performed on transformed type expressions. This simplifies the algorithm and improves its completeness. Figure 9 presents the algorithmic rules for type equality.

Lemma 4. Consider a goal \( \forall V ; C ; T \vdash A \equiv B. \Sigma \) Each \( A \) and \( B \) is both structural or both type names.

Proof. By induction on the algorithmic type equality judgment, and the fact that type definitions are contractive, and the intermediate transformation where every continuation is replaced by a type name.
\[ \forall V. C \Rightarrow V_1[e_1] \equiv_R V_2[e_2] \]

To prove \( \forall V. C \Rightarrow V_1[e_1] \equiv_R V_2[e_2] \), it is sufficient to show that \( V_1[e_1[\sigma]] \equiv_R V_2[e_2[\sigma]] \) for any substitution \( \sigma \) such that \( \vdash C[\sigma] \). Applying this substitution to \( V \); \( C \in \exists V'. C' \land \overrightarrow{e_1} \equiv \overrightarrow{e_1} \land \overrightarrow{e_2} \equiv \overrightarrow{e_2} \); we get that \( \exists V'. C' \land \overrightarrow{e_1} \equiv \overrightarrow{e_1} \land \overrightarrow{e_2} \equiv \overrightarrow{e_2} \) since \( C[\sigma] \) is true. Thus, there exists \( \exists V'. C' \) such that \( \vdash C'[\sigma]' \) holds, and \( e_1'[\sigma]' = e_1[\sigma] \) and \( e_2'[\sigma]' = e_2[\sigma] \).

This implies that \( V_1[e_1[\sigma]] \equiv_R V_2[e_2[\sigma]] \) since \( e_1'[\sigma]' = e_1[\sigma] \) and \( e_2'[\sigma]' = e_2[\sigma] \).

\[ \square \]

**Definition 11.** Given \( V \); \( C \vdash T \equiv_B \), define the set of conclusions \( S \) by the judgment \( \text{conc}(V; C; \Gamma \vdash A \equiv_B) = S \). This judgment simply collects the set of conclusions in a derivation tree. It is defined using the rules in Figure 10.

**Theorem 5.** If \( \forall V. C \vdash T \equiv_B \), then \( \forall V. C \Rightarrow A \equiv_B \).

\begin{align*}
\text{Figure 9. Algorithmic Rules for Type Equality} \\
V & \vdash C. \Gamma \vdash T \equiv U \quad \vdash V. C \vdash \psi. \Gamma \vdash T \equiv U \\
V & \vdash C. \Gamma \vdash T \equiv U \quad \vdash V. C \vdash \neg \psi. \Gamma \vdash T \equiv U \\
V & \vdash C. \Gamma \vdash T \equiv U \quad \vdash V. C \vdash \psi \land \psi'. \Gamma \vdash T \equiv U";
\end{align*}

**B.3 Soundness of Type Equality**

**Definition 9.** The notation \( \forall V. C \Rightarrow A \equiv_R B \) denotes that two types \( A \) and \( B \) quantified over \( V \) are related in \( \mathcal{R} \) under the constraint \( C \). This holds when for all ground substitutions \( \sigma \) over \( V \) satisfying \( C \), \( A[\sigma] \equiv_R B[\sigma] \).

**Definition 10.** The declarative definition for equality, denoted by \( \forall V. C \Rightarrow A \equiv B \) holds when there exists a type simulation \( \mathcal{R} \) such that \( \forall V. C \Rightarrow A \equiv_R B \).

**Lemma 5.** Suppose \( \forall V'. C' \Rightarrow V_1[e_1] \equiv_R V_2[e_2] \) holds. And suppose, \( V \vdash C \vdash \exists V'. C' \land \overrightarrow{e_1} \equiv \overrightarrow{e_1} \land \overrightarrow{e_2} \equiv \overrightarrow{e_2} \) for some \( V, \overrightarrow{e_1}, \overrightarrow{e_2} \). Then, \( \forall V. C \Rightarrow V_1[e_1] \equiv_R V_2[e_2] \) holds.
A two counter machine $M$ is defined as a triple $(i, c_1, c_2)$, where $i$ denotes the number of the current instruction and $c_j$‘s denote the value of the counters. A configuration $C'$ is defined as the successor configuration of $C$, written as $C \Rightarrow C'$ if $C'$ is the result of executing the $i$-th instruction on $C$. If $i_t = 1$, then $C = (i, c_1, c_2)$ has no successor configuration.

The computation of $M$ is the unique maximal sequence $\rho = p(0)p(1)\ldots$ such that $\rho(i) \mapsto \rho(i+1)$ and $\rho(0) = (1, 0, 0)$. Either $\rho$ is infinite, or ends in $(i, c_1, c_2)$ such that $i_t = \text{halt}$.

The halting problem refers to determining whether the computation of a two counter machine $M$ is finite. Both the halting problem and its dual, the non-halting problem are undecidable.

We prove the undecidability of the type equality problem by reducing the non-halting problem of an instance of the
two counter machine $M$ to an instance of the type equality problem.

Consider the instruction $i_l$. It can have one of the three forms shown above. We will define a recursive type based on the type of instruction. First, we define a type $T_{\inf} = \oplus\{\ell : T_{\inf}\}$ and $T'_{\inf} = \oplus\{\ell' : T'_{\inf}\}$.

- Case ($i_l = \text{inc}(c_1)$; goto $k$) : In this case, we define the type $T[c_1, c_2] = \oplus\{\text{inc}_1 : T_k[c_1 + 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'[c_1, c_2] = \oplus\{\text{inc}_1 : T_k'[c_1 + 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$.
- Case ($i_l = \text{inc}(c_2)$; goto $k$) : In this case, we define the type $T[c_1, c_2] = \oplus\{\text{inc}_2 : T_k[c_1, c_2 + 1]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'[c_1, c_2] = \oplus\{\text{inc}_2 : T_k'[c_1, c_2 + 1]\}$ for all $c_1, c_2 \in \mathbb{N}$.
- Case ($i_l = \text{zero}(c_1)$; goto $k$ : dec($c_j$); goto $l$) : In this case, we define the type $T[c_1, c_2] = \oplus\{\text{zero}_1 : \{c_1 = 0\}, T_k[c_1, c_2], \text{dec}_1 : \{c_1 > 0\}, T_l[c_1 - 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'[c_1, c_2] = \oplus\{\text{zero}_1 : \{c_1 = 0\}, T_k'[c_1, c_2], \text{dec}_1 : \{c_1 > 0\}, T_l'[c_1 - 1, c_2]\}$.
- Case ($i_l = \text{zero}(c_2)$; goto $k$ : dec($c_j$); goto $l$) : In this case, we define the type $T[c_1, c_2] = \oplus\{\text{zero}_2 : \{c_2 = 0\}, T_k[c_1, c_2], \text{dec}_2 : \{c_2 > 0\}, T_l[c_1, c_2 - 1]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'[c_1, c_2] = \oplus\{\text{zero}_2 : \{c_2 = 0\}, T_k'[c_1, c_2], \text{dec}_2 : \{c_2 > 0\}, T_l'[c_1, c_2 - 1]\}$ for all $c_1, c_2 \in \mathbb{N}$.
- Case ($i_l = \text{halt}$) : In this case, we define $T[c_1, c_2] = T_{\inf}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'[c_1, c_2] = T'_{\inf}$ for all $c_1, c_2 \in \mathbb{N}$.

Suppose, the counter $M$ is initialized in the state $(1, m, n)$. The type equality question we ask is $T[m, n] = T'[m, n]$. The two types only differ at the halting instruction. If and only if $M$ halts, the first type sends infinitely many $\ell$ labels while the second type sends infinitely many $\ell'$ labels. While if $M$ does not halt, the two types produce exactly the same communication behavior, since the halting instruction is never reached. Hence, the two types are equal iff $M$ does not halt.

Hence, the type equality problem can be reduced to the non-halting problem.
C Formal Typing Rules

C.1 Explicit Type System

This section formalizes the algorithmic explicit type system, providing a compact collection of all the rules. Figure 13 presents the algorithmic typing rules, Figure 14 presents the operational semantics and Figure 12 presents the configuration typing rules.

C.2 Implicit Type System

The rules of the implicit system are similar, except they do not change the process terms for the rules involving proof constraints and potential. The rest of the rules exactly match the explicit system. Figure 11 contains the implicit typing rules that differ from the explicit system.

\[
\begin{align*}
V; C \vdash \phi & \quad \quad \quad V; C \vdash \Delta, i^\beta P :: (x : A) \quad \text{?R} \\
V; C \vdash \Delta, i^\beta Q :: (z : C) & \quad \quad \quad V; C \vdash \Delta, (x : ?\{\phi}\{\}, A), i^\beta Q :: (z : C) \quad \text{?R} \\
V; C \vdash \Delta, i^\beta P :: (x : A) & \quad \quad \quad V; C \vdash \Delta, (x : ?\{\phi}\{\}, A), i^\beta P :: (x : A) \quad \text{!R} \\
V; C \vdash q \geq r & \quad \quad \quad V; C \vdash \Delta, i^\beta P :: (x : a) \\
V; C \vdash \Delta, i^\beta Q :: (z : C) & \quad \quad \quad V; C \vdash \Delta, (x : ?a), i^\beta Q :: (z : C) \quad \text{!R} \\
V; C \vdash \Delta, i^\beta Q :: (z : C) & \quad \quad \quad V; C \vdash \Delta, (x : ?a), i^\beta Q :: (z : C) \quad \text{!R} \\
\end{align*}
\]

Figure 11. Implicit Typing Rules

\[
\begin{align*}
\Delta \vdash E :: \Delta' \quad \quad \quad \Sigma \vdash E' :: \Delta'' \quad \text{compose} \\
\Delta \vdash (S, S') :: \Delta'' \quad \quad \quad \rho_1 \vdash (x : A) \quad \text{proc} \\
\Delta, \Delta_1 \vdash (\text{proc}(x, w, P)) :: (\Delta, (x : A)) \quad \quad \quad \rho_2 \vdash (x : A) \quad \text{msg} \\
\Delta, \Delta_1 \vdash (\text{msg}(x, w, P)) :: (\Delta, (x : A)) \\
\end{align*}
\]

Figure 12. Configuration Typing Rules
Figure 13. Explicit Typing Rules
\((\text{id}\,c)\) \(\text{msg}(d, w, M), \text{proc}(c, w', s, c \leftarrow d) \rightarrow \text{msg}(c, w + w', M[c/d])\)

\((\text{id}^{-}c)\) \(\text{proc}(c, w, s, c \leftarrow d), \text{msg}(e, w', M(c)) \rightarrow \text{msg}(e, w + w', M(c)[d/c])\)

\((\text{def}C)\) \(\text{proc}(c, w, x \leftarrow f[e] \leftarrow \bar{y} ; Q) \rightarrow \text{proc}(a, 0, P_f[a/x, \bar{y} / \bar{A}_f, \bar{e} / \bar{e}], \text{proc}(c, w, Q[a/x]))\)

\((\text{S})\) \(\text{proc}(c, w, c.k ; P) \rightarrow \text{proc}(c', w, P[c'/c]), \text{msg}(c, 0, c.k ; c \leftarrow c')\)

\((\text{C})\) \(\text{msg}(c', w', c.k ; c \leftarrow c'), \text{proc}(d, w, \text{case} c (\ell \Rightarrow Q_\ell)[\ell] \Rightarrow \text{proc}(d, w + w', Q_k[c'/c])\)

\((\delta, S)\) \(\text{proc}(d, w, c.k ; Q) \rightarrow \text{msg}(c', 0, c.k ; c' \leftarrow c), \text{proc}(d, w, Q[c'/c])\)

\((\delta, C)\) \(\text{proc}(c, \text{case} c (\ell \Rightarrow Q_\ell)[\ell] \Rightarrow \text{msg}(c', w', c.k ; c' \leftarrow c) \rightarrow \text{proc}(c', w + w', Q_k[c'/c])\)

\((1S)\) \(\text{proc}(c, w, \text{close} c) \rightarrow \text{msg}(c, w, \text{close} c)\)

\((1C)\) \(\text{msg}(c, w', \text{close} c), \text{proc}(d, w, \text{wait} c ; Q) \rightarrow \text{proc}(d, w + w', Q)\)

\((\oplus S)\) \(\text{proc}(c, w, \text{send} c d ; P) \rightarrow \text{proc}(c', w, P[c'/c]), \text{msg}(c, 0, \text{send} c d ; c \leftarrow c')\)

\((\oplus C)\) \(\text{msg}(c', w', \text{send} c d ; c \leftarrow c'), \text{proc}(e, w, x \leftarrow \text{recv} c ; Q) \rightarrow \text{proc}(e, w + w', Q[c'/c])\)

\((\rightarrow S)\) \(\text{proc}(e, w, \text{send} c d ; Q) \rightarrow \text{msg}(c', 0, \text{send} c d ; c' \leftarrow c), \text{proc}(e, w, Q[c'/c])\)

\((\rightarrow C)\) \(\text{proc}(c, w, x \leftarrow \text{recv} x ; P), \text{msg}(c', w', \text{send} c d ; c' \leftarrow c) \rightarrow \text{proc}(c', w + w', P[c'/d/c, x])\)

\((\rightarrow S)\) \(\text{proc}(c, w, \text{send} c \{e\} ; P) \rightarrow \text{proc}(c', w, P[c'/c]), \text{msg}(c, 0, \text{send} c \{e\} ; c \leftarrow c')\)

\((\rightarrow C)\) \(\text{msg}(c', w', \text{send} c \{e\} ; c \leftarrow c'), \text{proc}(d, w, \{n\} \leftarrow \text{recv} c ; Q) \rightarrow \text{proc}(d, w + w', Q[e/n][c'/c])\)

\((\hat{\text{S}})\) \(\text{proc}(d, w, \text{send} c \{e\} ; P) \rightarrow \text{msg}(c', 0, \text{send} c \{e\} ; c' \leftarrow c), \text{proc}(d, w, P[c'/c])\)

\((\hat{\text{C}})\) \(\text{proc}(d, w, \{n\} \leftarrow \text{recv} c ; Q), \text{msg}(c', w', \text{send} c \{e\} ; c' \leftarrow c) \rightarrow \text{proc}(d, w + w', Q[e/n][c'/c])\)

\((\hat{?}S)\) \(\text{proc}(c, w, \text{assert} c \{\phi\} ; P) \rightarrow \text{proc}(c', w, P[c'/c]), \text{msg}(c, 0, \text{assert} c \{\phi\} ; c \leftarrow c')\)

\((\hat{?}C)\) \(\text{msg}(c', w', \text{assert} c \{\phi\} ; c \leftarrow c'), \text{proc}(d, w, \text{assume} c \{\phi_2\} ; Q) \rightarrow \text{proc}(d, w + w', Q[c'/c])\)

\((\oplus S)\) \(\text{proc}(d, w, \text{assert} c \{\phi\} ; P) \rightarrow \text{msg}(c', 0, \text{assert} c \{\phi\} ; c' \leftarrow c), \text{proc}(d, w, P[c'/c])\)

\((\oplus C)\) \(\text{proc}(d, w, \text{assume} c \{\phi_1\} ; Q), \text{msg}(c', w', \text{assert} c \{\phi_2\} ; c' \leftarrow c) \rightarrow \text{proc}(d, w + w', Q[c'/c])\)

\((\odot S)\) \(\text{proc}(c, w, \text{pay} c \{r\} ; P) \rightarrow \text{proc}(c', w, P[c'/c]), \text{msg}(c, 0, \text{pay} c \{r\} ; c \leftarrow c')\)

\((\odot C)\) \(\text{msg}(c', w', \text{pay} c \{r\} ; c \leftarrow c'), \text{proc}(d, w, \text{get} c \{r\} ; Q) \rightarrow \text{proc}(d, w + w', Q[c'/c])\)

\((\odot S)\) \(\text{proc}(d, w, \text{pay} c \{r\} ; P) \rightarrow \text{msg}(c', 0, \text{pay} c \{r\} ; c' \leftarrow c), \text{proc}(d, w, P[c'/c])\)

\((\odot C)\) \(\text{proc}(c, w, \text{get} c \{r\} ; Q), \text{msg}(c', w', \text{pay} c \{r\} ; c' \leftarrow c) \rightarrow \text{proc}(c', w + w', Q[c'/c])\)

\((\text{work})\) \(\text{proc}(c, w, \text{work} \{r\} ; P) \rightarrow \text{proc}(c, w + r, P)\)

---

**Figure 14.** Basic and Refined Operational Semantics
D Forcing Calculus

We define the forcing calculus, with the following judgment \( \mathcal{V} : \mathcal{C} ; \Delta^- ; \Omega \Downarrow P :: (x : A) \). Here, \( \mathcal{C} \) denotes the arithmetic constraints that are currently valid, \( \Delta^- \) stores the negative propositions, while \( \Omega \) is an ordered context storing the remaining propositions. This calculus applies the \(!R, ?L, \ll R, \gg L\) rules eagerly, and the \(?R, !L, \gg R, \ll L\) rules lazily. Later, we will establish that this forcing calculus is equivalent to the implicit type system. First, we define the grammar of types.

\[
A_+ ::= S | \{\phi\}.A_+ | \phi^rA^r \\
A_- ::= S | \{\phi\}.A_- | \phi^rA^r \\
A ::= A_+ | A_- \\
S ::= \Theta(\ell : A)_{\ell \in L} | \&\{\ell : A\}_{\ell \in L} | A \otimes A | A \rightarrow A \\
\exists n. A | \forall n. A
\]

Here, \( S \) denotes a structural type. Next, we define the rules for eagerly applying the \(!R, ?L, \ll R, \gg L\) rules.

\[
\begin{align*}
\mathcal{V} : C \land \phi ; \Delta^- ; \Omega \Downarrow P :: (x : A^-) & \quad \mathcal{V} : C ; \Delta^- ; \Omega \Downarrow P :: (x : \{\phi\}.A^-) & \quad \text{!R} \\
\mathcal{V} : C ; \Delta^- ; \Omega \Downarrow P :: (x : \{\phi\}.A^-) & \quad \mathcal{V} : C ; \Delta^- ; \Omega \Downarrow P :: (x : \phi^rA^r) & \quad \ll R \\
\mathcal{V} : C ; \Delta^- ; \Omega \Downarrow P :: (x : A^-) & \quad \mathcal{V} : C ; \Delta^- ; \Omega \Downarrow P :: (x : \phi^rA^r) & \quad \gg L \\
\end{align*}
\]

If a negative type is encountered in the ordered context, it is considered as stable and moved to \( \Delta^- \).

\[
\mathcal{V} ; C ; \Delta^-, (x : A^+); \Omega \Downarrow P :: (z : C^+) & \quad \text{move} \\
\mathcal{V} ; C ; \Delta^-; \Omega \Downarrow P :: (z : C^+)
\]

The invertible rules are applied until we reach a stable sequent defined using the normal form \( \mathcal{V} ; C ; \Delta^-; \cdot \Downarrow P :: (x : A^+) \). At this point, we case analyze on the program \( P \) to decide which channel to force. If \( P \) is a forward, we first force right.

\[
\mathcal{V} ; C ; (y : B^-) ; \cdot \Downarrow x \leftarrow y :: [x : A^+] & \quad \text{id-}F_R \\
\mathcal{V} ; C ; (y : B^-) ; \cdot \Downarrow x \leftarrow y :: (x : A^+)
\]

then force left.

\[
\mathcal{V} ; C ; (y : B^-) ; \cdot \Downarrow x \leftarrow y :: (x : S) & \quad \text{id-}F_L \\
\mathcal{V} ; C ; (y : B^-) ; \cdot \Downarrow x \leftarrow y :: [x : S]
\]

Finally, the forward is typechecked after the forcing.

\[
\mathcal{V} ; C ; C = q = 0 & \quad \mathcal{V} ; C ; [y : S] ; \cdot \Downarrow x \leftarrow y :: (x : S) & \quad \text{id}
\]

On encountering a spawn, we force each channel used in the context in order. If a forced channel becomes structural, we lose force on that channel, and force the next. These are rules \( \text{spawn}-F_n \) and \( \text{spawn}-F_l \) in Figure 15. Once the first channel is forced, we typecheck the continuation, which could possibly be an unstable sequent, as described in spawn rule in Figure 15. For the structural types, we force the channel that is being communicated on.

\[
\mathcal{V} ; C ; \Delta^-; \cdot \Downarrow (x.k : P) :: [x : A^+] & \quad \mathcal{V} ; C ; \Delta^-; \cdot \Downarrow (x.k : P) :: (x : A^+) & \quad \text{\&F_R} \\
\mathcal{V} ; C ; \Delta^-; \cdot \Downarrow (x.k : P) :: (x : A^+) & \quad \text{\&F_L} \\
\mathcal{V} ; C ; \Delta^-; \cdot \Downarrow (x.k : P) :: (x : A^+) & \quad \text{\&F_R} \\
\mathcal{V} ; C ; \Delta^-; \cdot \Downarrow (x.k : P) :: (x : A^+) & \quad \text{\&F_L}
\]

For the tensor and lolli rules, we force both the channel involved in the send while sending, and while receiving, we
\[
\frac{}{\mathcal{V}; C; \Delta^-, [y_n : A^*] ; \cdot \overline{b} \ (x \leftarrow \lambda [e_c] \leftarrow y_1 \ldots y_n; Q_x) :: (z : C^*) \ \text{spawn-F}_n}
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, [y_n : A^*] ; \cdot \overline{b} \ (x \leftarrow \lambda [e_c] \leftarrow y_1 \ldots y_n; Q_x) :: (z : C^*) \ \text{spawn-F}_i}
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, [y_l : A^-], [y_{l+1} : S] ; \cdot \overline{b} \ (x \leftarrow \lambda [e_c] \leftarrow y_1 \ldots y_n; Q_x) :: (z : C^*) \ \text{spawn-F}_i}
\]

**Figure 15.** Spawn rules in forcing calculus

place the new channel in the ordered context.

\[
\frac{}{\mathcal{V}; C; \Delta^-, [w : A^-] ; \cdot \overline{b} \ y \leftarrow \text{send} x w; P :: (x : B^*)} \ \Theta F^1_R
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, (w : A^-) ; \cdot \overline{b} \ y \leftarrow \text{send} x w; P :: (x : B^*)} \ \Theta F^2_R
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, (w : S) ; \cdot \overline{b} \ y \leftarrow \text{send} x w; P :: (x : A^*)} \ \Theta F^3_R
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, [w : A^-] ; \cdot \overline{b} \ y \leftarrow \text{recv} x; Q_y :: (z : C^*)} \ \Theta F_L
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, (w : A^-) ; \cdot \overline{b} \ y \leftarrow \text{recv} x; Q_y :: (z : C^*)} \ \Theta F_{L'}^1
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, (w : S), [x : A^-] ; \cdot \overline{b} \ y \leftarrow \text{send} x w; Q :: (z : C^*)} \ \Theta F_{L'}^2
\]

The rules for 1 are analogous.

\[
\frac{}{\mathcal{V}; C; \cdot ; \cdot \overline{b} \ \text{close} x :: [x : A^*]} \ 1F_R
\]
\[
\frac{}{\mathcal{V}; C; \cdot ; \cdot \overline{b} \ \text{close} x :: (x : A^*)} \ 1F_R
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, [x : A^-] ; \cdot \overline{b} \ \text{wait} x :: (z : C^*)} \ 1F_L
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, (x : A^-) ; \cdot \overline{b} \ \text{wait} x :: (z : C^*)} \ 1F_L
\]

Since the $\forall$ and $\exists$ operators are structural, they follow the same structure.

\[
\frac{}{\mathcal{V}; C; \Delta^- ; \cdot \overline{b} \ \text{send} x \{t\} :: [x : A^*]} \ \exists F_R
\]
\[
\frac{}{\mathcal{V}; C; \Delta^- ; \cdot \overline{b} \ \text{send} x \{t\} :: (x : A^*)} \ \exists F_R
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, [x : A^-] ; \cdot \overline{b} \ \text{recv} x :: (z : C^*)} \ \exists F_L
\]
\[
\frac{}{\mathcal{V}; C; \Delta^-, (x : A^-) ; \cdot \overline{b} \ \text{recv} x :: (z : C^*)} \ \exists F_L
\]

Finally, we provide the structural rules of forcing calculus in Figure 16.


**Definition 12.** We define the 4 notations below.

\[
[!\phi].A^!_R = \phi \land [A]_R \quad [\phi].A^!_L = [A]_L \quad [\phi]_R = [A]_R \quad [\phi]_L = [A]_L
\]

**Lemma 7** (Move Left). If $!\mathcal{V} C \Delta^- \Omega (x : A) \overline{b} P :: (z : C^*)$, then $\mathcal{V} C \Delta^- \Omega [A]_L \Delta \Delta^- (x : [A]_L) \overline{b} P :: (z : C^*)$.

Proof. If $[A]_L = S$ and $[A]_L = \phi$, then $\mathcal{V} C \Delta^- \Omega \phi \Delta^- (x : S) :: [A]_R \overline{b} P :: (z : C^*)$ by inverting $L$ or move.

**Lemma 8** (Stable Sequent). If $!\mathcal{V} C \Delta^- \Delta^- \overline{b} P :: (x : A)$, then $!\mathcal{V} C \Delta^- \Delta^- \overline{b} P :: (x : A)$.


**Lemma 9** (Lazy $?R$). If $!\mathcal{V} C \Delta^- \Omega \overline{b} P :: (x : A^*)$ and $\mathcal{V} C \phi \Delta^- \Omega \overline{b} P :: (x : ?\phi).A^*$.

Proof. The proof proceeds by induction on the typing judgment. Since we do not allow quantifier alternation, the $!R$ rule cannot be applied in either case. The $?L$ rule can apply in either case. The move rule applies in either case. The id$R$ rule can be applied in either case. The $idL$ cannot be applied in either case since it requires the offered channel to be forced. The same holds for $?R$. The $!L$ rule can be applied in either case. If the last rule applied is id, we first apply id$R$, then id. All the spawn rules still apply in the same holds.
Theorem 7 (Completeness of QRecon). If \( \forall V; C; \cdot; \Delta \beta P :: (x : A) \), then \( \forall V; C; \cdot; \Delta \beta P :: (x : A) \).

Proof. The proof proceeds by induction on the typing judgment of the implicit type system. We case analyze on the last rule applied. First, we consider the structural cases.

• Case (\( \oplus R_k \)): \( P = x.k \) and \( A = \oplus \{ \ell : A \}_{\ell \in L} \).

\[
\forall V; C; \Delta \beta P :: (x : A) \\
\forall V; C; \Delta \beta P :: (x : A_k) \quad \forall R_k
\]

By the induction hypothesis, \( \forall V; C; \cdot; \Delta \beta P :: (x : A_k) \). Using Lemma 8, we get \( \forall V; C \land [A]_k^L \land [A_k]_R^L; [A]_k^L; \cdot \beta P :: (x : [A_k]_R^L) \). Since this is a stable sequent, we reapply the \( \oplus R_k \) rule, we get (note that \( [A_k]_R^L = A_k \) and \( [A_k]_R^L = A_k \)).

\[
\forall V; C \land [A]_k^L; [A]_k^L; \cdot \beta P :: (x : A_k) \\
\forall V; C \land [A]_k^L; [A]_k^L; \cdot \beta x.k :: P :: (x : \oplus \{ \ell : A \}_{\ell \in L}) \quad \oplus R_k
\]

Now, applying the \( \oplus F_R \) rule,

\[
\forall V; C \land [A]_k^L; [A]_k^L; \cdot \beta x.k :: P :: (x : \oplus \{ \ell : A \}_{\ell \in L}) \\
\forall V; C \land [A]_k^L; [A]_k^L; \cdot \beta x.k :: P :: (x : \oplus \{ \ell : A \}_{\ell \in L}) \quad \oplus F_R
\]

Reapplying the move and \( !L \) rules, we get \( \forall V; C; \cdot; \Delta \beta x.k :: P :: (x : \oplus \{ \ell : A \}_{\ell \in L}) \).
• Case ($\oplus L$) : $P = case x (∂ \Rightarrow Q_ℓ)_{τ ∈ L}$ and $Δ = Δ(x : Sℓ(x : A))_{τ ∈ L}$. 

$$\forall v : C; Δ(x : A) : β Q_ℓ ≡ (z : C)\quad \oplus L$$

$$\forall v : C; Δ(x : Sℓ(x : A))_{τ ∈ L} : β case x (∂ \Rightarrow Q_ℓ)_{τ ∈ L} : (z : C)$$

By the induction hypothesis, $\forall v : C; Δ : \beta Q_ℓ \equiv (z : C)$. First, we use Lemma 6 to get $\forall v : C \land [C]_{R} : \beta Q_ℓ \equiv (z : [C]_{R})$. Then, we use Lemma 7 successively on $Δ$ to get $\forall v : C \land [C]_{R} \land [Δ]_{L} ; \beta Q_ℓ \equiv (z : [C]_{R})$. Now, we apply the $\oplus L$ rule to get

$$\forall v : C \land [C]_{R} \land [Δ]_{L} ; [Δ]_{L} ; (x : A) : β Q_ℓ \equiv (z : [C]_{R}) \quad \oplus L$$

Applying the $\oplus L$ rule, we get $\forall v : C \land [C]_{R} \land [Δ]_{L} ; [Δ]_{L} ; (x : A) : \beta case x (∂ \Rightarrow Q_ℓ)_{τ ∈ L} : (z : [C]_{R})$. Reapplying the invertible rules! On $C$ and move and $?L$ on $Δ(x : Sℓ(x : A))_{τ ∈ L}$, we get $\forall v : C ; ; Δ. (x : Sℓ(x : A))_{τ ∈ L} : β case x (∂ \Rightarrow Q_ℓ)_{τ ∈ L} : (z : C)$. 

• Case ($\& R, \& L, ?L$) : Analogous to $?L$. 

• Case ($\rightarrow R$) : $P = y \leftarrow \text{recv} x ; P_y$ and $A = S \rightarrow B$.

$$\forall v : C ; Δ(y : S) : β P_y : (x : B)$$

$$\forall v : C ; Δ ; β y \leftarrow \text{recv} x ; P_y : (x : S \rightarrow B) \quad \rightarrow R$$

By the induction hypothesis, $\forall v : C ; ; (y : S) : β P_y : (x : B)$. Using Lemma 8, we get $\forall v : C \land [B]_{R} \land [Δ]_{L} ; ([Δ]_{L} ; (y : S) : β P_y : (x : B))_{τ ∈ L}$. Note that since $B$ is a positive type, $[B]_{R} = \top$ and $[B]_{R} = B$. Thus, we have $\forall v : C \land [Δ]_{L} ; ([Δ]_{L} ; (y : S) : β P_y : (x : B))_{τ ∈ L}$. Now, apply the $\rightarrow R$ rule to get

$$\forall v : C \land [Δ]_{L} ; ([Δ]_{L} ; (y : S) : β P_y : (x : B))_{τ ∈ L} \quad \rightarrow R$$

$$\forall v : C \land [Δ]_{L} ; ([Δ]_{L} ; (y : S) : β P_y : (x : B)))_{τ ∈ L} \quad \rightarrow R$$

Now, we drop force using the $\rightarrow F_2$ rule to get $\forall v : C \land [Δ]_{L} ; ([Δ]_{L} ; (y : S) : β y \leftarrow \text{recv} x ; P_y : (x : S \rightarrow B))_{τ ∈ L}$. Finally, we apply the $?L$, move rules to get $\forall v : C ; ; y : β y \leftarrow \text{recv} x ; P_y : (x : S \rightarrow B)$.

• Case ($\rightarrow L$) : $P = \text{send} x w ; Q$ and $Δ = Δ(w : S), (x : S \rightarrow B)$.

$$\forall v : C ; Δ(w : S), (x : S \rightarrow B) : β Q : (z : C)$$

$$\forall v : C ; Δ(w : S), (x : S \rightarrow B) : β send x w ; Q : (z : C) \quad \rightarrow L$$

By the induction hypothesis, we get $\forall v : C ; ; (x : B) : β Q : (z : C)$. First, we use Lemma 6 to get $\forall v : C \land [C]_{R} ; ; (x : B) : β Q : (z : [C]_{R})$. Then, we use Lemma 7 on $Δ$ to get $\forall v : C \land [C]_{R} \land [Δ]_{L} ; [Δ]_{L} ; (x : B) : β Q : (z : [C]_{R})$. Now, we apply the $\rightarrow L$ rule to get $\forall v : C \land [C]_{R} \land [Δ]_{L} ; [Δ]_{L} ; (x : B) : β Q : (z : [C]_{R}) \quad \rightarrow L$

Applying the $\rightarrow F_2$ and $\rightarrow F_1$ rules in order, we get $\forall v : C \land [C]_{R} \land [Δ]_{L} ; [Δ]_{L} ; (x : B) : β send x w ; Q : (z : [C]_{R})$. Now, reapplying the invertible rules, we get $\forall v : C ; ; Δ(w : S), (x : S \rightarrow B) : β send x w ; Q : (z : C)$.

• Case ($\& R, \& L, 1R, 1L$) : Analogous to $\rightarrow L$.

• Case (id) : $P = x \leftarrow y$.

$$\forall c : \beta x \leftarrow y : (x : A)$$

We proceed by induction on the structure of $A$. If $A$ is structural, we use the following derivation in the forcing calculus.

$$\forall v : C ; ; (y : A) : β x \leftarrow y : (x : A')$$

By the induction hypothesis, we apply the derivation

$$\forall v : C ; ; (y : A) : β x \leftarrow y : (x : A') \quad \text{loseR}$$

$$\forall v : C ; ; (y : A) : β x \leftarrow y : (x : A') \quad \text{idR}$$

$$\forall v : C ; ; (y : ?(φ, A')) : β x \leftarrow y : (x : ?(φ, A')) \quad \text{?L}$$

A similar argument would work for $A = !(φ, A')$.

• Case (spawn) : Since all types involved in a spawn are structural, this case is trivial.

• Case ($\exists R$) : $A = ?!(φ, A)$.

$$\forall v : C ; ; φ : φ \quad \forall v : C ; ; A : β P : (x : A) \quad ?R$$

By the induction hypothesis, we get $\forall v : C ; ; Δ : β P : (x : A)$. We use Lemma 9 to get $\forall v : C ; ; Δ : β P : (x : ?!(φ, A))$ since we know that $\forall v : C ; ; Δ : φ$. 

• Case ($\exists L$) : $Δ = Δ, (x : ?!(φ, A))$ and $P = Q$.

$$\forall v : C ; ; φ : φ \quad ∀ v : C ; ; A : β Q : (z : C) \quad ?L$$

By the induction hypothesis, we get that $\forall v : C ; ; Δ : A : β Q : (z : C)$. First, we invert the $?R$ rule to get $\forall v : C ; ; φ : φ \land [C]_{R} : ; Δ : A : β Q : (z : [C]_{R})$. Then, we apply the $?L$ rule to get $\forall v : C ; ; Δ : (x : ?!(φ, A)) : β Q : (z : [C]_{R})$. Then we reapply the $?R$ to get back $\forall v : C ; ; Δ : (x : ?!(φ, A)) : β Q : (z : C)$.

□