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 ergometric 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 algorithm for type checking which is complete relative to an oracle for type equality. Process expressions in this explicit language are rather verbose, so we also introduce an implicit form and a sound and complete algorithm for reconstructing explicit programs, 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.

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

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

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

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

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 ϕ*" and is analogous to the *assertion* of ϕ 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 index domain from which the propositions ϕ are drawn is Presburger arithmetic and hence decidable, no proof of ϕ will actually be sent, but we can nevertheless verify the constraint statically (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 ϕ*, analogous to the *assumption* of ϕ), 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 ergometric [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 ($\exists n.\,A$, $\forall n.\,A$, $?\{\phi\}.\,A$, and $!\{\phi\}.\,A$) have explicit communication counterparts. Similarly, type constructs for receiving or sending *potential* ($\triangleleft^p A$ and $\triangleright^p A$) for amortized work analysis have corresponding process constructs to (conceptually) send and receive potential. Revisiting the queue example, the type

$$\text{queue}_A[n] = \&\{\textbf{ins} : \triangleleft^{2n}(A \multimap \text{queue}_A[n+1]),$$
$$\textbf{del} : \triangleleft^2 \oplus \{\textbf{none} : ?\{n=0\}.\,\mathbf{1},$$
$$\textbf{some} : ?\{n>0\}.\,A \otimes \text{queue}_A[n-1]\}\}$$

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 $\forall n.\,\exists k.\,(n = 2k \lor n = 2k+1)$ is true but the witness for k as a Skolem function of n (namely $\lfloor n/2 \rfloor$) cannot be expressed in Presburger arithmetic. Since witnesses are critical if we want to understand the work performed by a computation, we require that type quantifiers $\forall n.\,A$ and $\exists n.\,A$ 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*.

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

The resulting judgment formally states that process P provides a service of session type C along channel z, while using the services of session types $A_1, \ldots, A_n$ provided along channels $x_1, \ldots, x_n$ respectively. All these channels must be distinct. We abbreviate the antecedent of the sequent by Δ.

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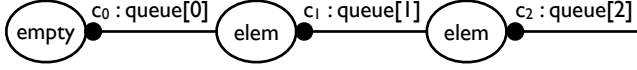
Figure 1. Implementation of queue data structure

```
1:   · ⊢ empty :: (s : queue_A[0])
2:     s ← empty =
3:       case s (
4:         ins ⇒ x ← recv s ;            % (x : A) ⊢ (s : qu_A[1])
5:               e ← empty ;  % (x : A), (e : qu_A[0]) ⊢ (s : qu_A[1])
6:               s ← elem[0] ← x, e
7:       | del ⇒ s.none ;                % · ⊢ (s : ?{0 = 0}. 1)
8:               assert s {0 = 0} ;           % · ⊢ (s : 1)
9:               close s)
10:  (x : A), (t : queue_A[n]) ⊢ elem[n] :: (s : queue_A[n + 1])
11:    s ← elem[n] ← x, t =
12:      case s (
13:        ins ⇒ y ← recv s ;
14:              t.ins ;
15:              send t y ;
                 % (x : A), (t : qu_A[n + 1]) ⊢ (s : qu_A[n + 2])
16:              s ← elem[n + 1] ← x, t
17:      | del ⇒ s.some ;
18:              assert s {n + 1 > 0} ;
19:              send s x ;         % (t : qu_A[n]) ⊢ (s : qu_A[n])
20:              s ← t)
```

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

of elements) and also the work performed by a computation must be nonnegative we work over the natural numbers $0, 1, \ldots$ rather than general integers. This includes a static validity check for types to ensure that all index refinements are nonnegative. For example, while checking the validity of $\text{queue}_A[n]$, we encounter the constraint $n > 0$ in the **some** branch, so we assume it and then verify that $n - 1 \geq 0$, ensuring the validity of $\text{queue}_A[n - 1]$.

One parallel implementation of such a queue data structure (Figure 1) is a sequence of *elem* processes, each storing an element of the queue, terminated by an *empty* process, representing the empty queue. The *empty* process offers along $c_0 : \text{queue}_A[0]$ (indicated by ● between *empty* and c_0), while each $elem[n]$ process uses $\text{queue}_A[n]$, and an element of type A (not shown) and offers on $\text{queue}_A[n + 1]$. In our notation, the process declarations will be written as

$$· \quad \vdash empty :: (s : \text{queue}_A[0])$$
$$(x : A), (t : \text{queue}_A[n]) \quad \vdash elem[n] :: (s : \text{queue}_A[n + 1])$$

Figure 1 shows a queue with 2 elements. The channel c_2 is the only one exposed to the client (not shown) and receives **ins** and **del** requests.

Figure 2 describes the implementation of *empty* and *elem* processes along with their type derivations on the right or line below (type $\text{queue}_A[n]$ abbreviated to $\text{qu}_A[n]$). Upon receiving the **ins** label and element $x : A$ (line 4), the *empty* process spawns a new *empty* process (line 5), binds it to channel e, and tail calls $elem[0]$ (line 6). On inputting the **del**

label, the *empty* process takes the **none** branch (line 7) since it stores no elements. Therefore, it needs to send a proof of $n = 0$, and since it offers $\text{queue}_A[0]$, it sends the trivial proof of $0 = 0$ (line 8), and closes the channel terminating communication (line 9). The *elem* process receives the **ins** label and element $y : A$ (line 13), passes on these two messages on the tail t (lines 14,15), and recurses with $elem[n + 1]$ (line 16). The type expected by $elem[n + 1]$ indeed matches the type of the input and output channels, as confirmed by the process declaration. On receiving the **del** label, the *elem* process replies with the **some** label (line 17) and the proof of $n + 1 > 0$ (line 18), again trivial since n is a natural number. It terminates with forwarding s along t (line 20). This forwarding is valid since the types of s and t exactly match as 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 **none** to the **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 ergometric types to Section 5. In addition to the type constructors arising from the connectives of intuitionistic linear logic ($\oplus, \&, \otimes, \mathbf{1} \multimap$), we have type names, indexed by a sequence of arithmetic expressions $V\overline{[e]}$, existential and universal quantification over natural numbers ($\exists n. A, \forall n. A$) and existential and universal constraints ($?\{\phi\}. A, !\{\phi\}. A$). We write i for constants.

Types	A	::=	$\oplus\{\ell : A\}_{\ell \in L} \mid \&\{\ell : A\}_{\ell \in L}$
		$\mid$	$A \otimes A \mid A \multimap A \mid \mathbf{1} \mid V\overline{[e]}$
		$\mid$	$?\{\phi\}. A \mid !\{\phi\}. A \mid \exists n. A \mid \forall n. A$
Arith. Exps.	e	::=	$i \mid e + e \mid e - e \mid i \times e \mid n$
Arith. Props.	ϕ	::=	$e = e \mid e > e \mid \top \mid \bot \mid \phi \land \phi$
		$\mid$	$\phi \lor \phi \mid \neg\phi \mid \exists n. \phi \mid \forall n. \phi$

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

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

Table 1. Basic session types with operational description

$$\frac{\mathcal{V} \; ; \; C \vDash q = 0}{\mathcal{V} \; ; \; C \; ; \; y : A \overset{g}{\vdash} (x \leftarrow y) :: (x : A)} \; \text{id}$$

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

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

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

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

Figure 3. Typing Rules for Basic Session Types

in blue, introduced in Section 5) leaving the complete set of rules to Appendix C.

The typing judgment has the form of a sequent

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

where $\mathcal{V}$ are index variables n, C are constraints over these variables expressed as a single proposition, Δ are the linear antecedents $x_i : A_i$, P is a process expression, and $x : A$ is the linear succedent. The *potential* q is explained in Section 5. We propose and maintain that the x_i and x are all distinct, and that all free index variables in C, Δ, P, and A are contained among $\mathcal{V}$. Finally, Σ is a fixed signature containing type and process definitions (explained in Section 3.1) Because it is fixed, we elide it from the presentation of the rules. In addition we write $\mathcal{V} \; ; \; C \vDash \phi$ for semantic entailment (proving ϕ assuming C) in the constraint domain where $\mathcal{V}$ contains all arithmetic variables in C and ϕ.

We formalize the operational semantics as a system of *multiset rewriting rules* [8]. We introduce semantic objects $\text{proc}(c, w, P)$ and $\text{msg}(c, w, M)$ which mean that process P or message M provide along channel c and have performed work w (as described in Section 5). A process configuration is a multiset of such objects, where any two channels provided are distinct (formally described at the end of Section 3.2).

3.1 Basic Session Types

External Choice The *external choice* type constructor $\&\{\ell : A_\ell\}_{\ell \in L}$ is an n-ary labeled generalization of the additive conjunction $A \& B$. Operationally, it requires the provider of $x : \&\{\ell : A_\ell\}_{\ell \in L}$ to branch based on the label $k \in L$ it receives from the client and continue to provide type A_k. The corresponding process term is written as case $x \; (\ell \Rightarrow P)_{\ell \in L}$. Dually, the client must send one of the labels $k \in L$ using the process term $(x.k \; ; \; Q)$ where Q is the continuation. The typing for the provider and client are rules $\&R$ and $\&L$ in Figure 3 respectively. Communication is asynchronous, so that the client $c.k \; ; \; Q$ sends a message k along c and continues as Q without waiting for it to be received. As a technical device to ensure that consecutive messages on a channel arrive in order, the sender also creates a fresh continuation channel c' so that the message k is actually represented as $(c.k \; ; \; c \leftarrow c')$ (read: send k along c and continue along c'). When the message k is received along c, we select branch k and also substitute the continuation channel c' for c. Rules $\&S$ and $\&C$ below describe the operational behavior of the provider and client respectively (c' fresh).

$(\&S) : \text{proc}(d, w, c.k \; ; \; Q) \; \mapsto \; \text{msg}(c', 0, c.k \; ; \; c' \leftarrow c),$
$\qquad\qquad\qquad\qquad\qquad \text{proc}(d, w, Q[c'/c])$
$(\&C) : \text{proc}(c, w, \text{case } c \; (\ell \Rightarrow Q_\ell)_{\ell \in L}),$
$\quad \text{msg}(c', w', c.k \; ; \; c' \leftarrow c) \; \mapsto \; \text{proc}(c', w + w', Q_k[c'/c])$

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

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 $x \; y \; ; \; P$ 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 $\otimes R$, $\otimes L$ in Figure 3. The dual operator $A \multimap B$ 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 $\Delta \vdash^g f\overline{[n]} = P :: (x : A)$ where f is the name of the process and P its definition. In addition, $\overline{n}$ is a sequence of arithmetic variables that Δ, P and A can refer to. All definitions are collected in a fixed global signature Σ. We require that $\overline{n}$; $\top$; $\Delta \vdash^g P :: (x : A)$ for every definition, thereby allowing definitions to be mutually recursive. A new instance of a defined process f can be spawned with the expression $x \leftarrow f\overline{[e]} \leftarrow \overline{y}$; Q where $\overline{y}$ is a sequence of channels matching the antecedents Δ and $\overline{[e]}$ is a sequence of arithmetic expression matching the variables $\overline{[n]}$. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q.

$$\frac{\mathcal{V} \; ; \; C \vDash r \geq q[\overline{e}/\overline{n}] \quad \overline{y : B} \vdash^g f\overline{[n]} = P_f :: (x : A) \in \Sigma \\ \Delta' = \overline{(y : B)}[\overline{e}/\overline{n}] \quad \mathcal{V} \; ; \; C \; ; \; \Delta, (x{:}A[\overline{e}/\overline{n}]) \vdash^{r-q} Q :: (z{:}C)}{\mathcal{V} \; ; \; C \; ; \; \Delta, \Delta' \vdash^r (x \leftarrow f\overline{[e]} \leftarrow \overline{y} \; ; \; Q) :: (z : C)} \; \text{def}$$

The declaration of f is looked up in the signature Σ, and $\overline{e}$ is substituted for $\overline{n}$ while matching the types in Δ' and $\overline{y}$ (second premise). The corresponding semantics rule also performs a similar substitution (a fresh)

(defC) $\text{proc}(c, w, x \leftarrow f\overline{[e]} \leftarrow \overline{d} \; ; \; Q) \mapsto$
$\text{proc}(a, 0, P_f[a/x, \overline{d}/\overline{y}, \overline{e}/\overline{n}]), \; \text{proc}(c, w, Q[a/x])$

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

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

Type Definitions As our queue example already showed, session types can be defined recursively, departing from a strict Curry-Howard interpretation of linear logic, analogous to the way pure ML or Haskell depart from a pure interpretation of intuitionistic logic. For this purpose we allow (possibly mutually recursive) type definitions $V\overline{[n]} = A$ in the signature Σ. Here, $\overline{[n]}$ denotes a sequence of arithmetic variables. 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 $V\overline{[e]}$ indexed with arithmetic refinements by $A[\overline{e}/\overline{n}]$ during type checking, and no explicit rules for recursive types are needed.

All types in a signature must be *valid*, formally written as $\mathcal{V} \; ; \; C \vdash A \; valid$, which requires that all free arithmetic

variables of C and A are contained in $\mathcal{V}$, and that for each arithmetic expression e in A we can prove $\mathcal{V}' \; ; \; C' \vdash e :$ nat for the constraints C' known at the occurrence of e (implicitly proving that $e \geq 0$).

3.2 The Refinement Layer

We now introduce quantifiers ($\exists n. A$, $\forall n. A$) and constraints ($?\{\phi\}. A$, $!\{\phi\}. A$). An overview of the types, process expressions, and their operational meaning can be found in Table 2.

Quantification The provider of $(c : \exists n. A)$ should send a witness i along channel c and then continue as $A[i/n]$. The witness is specified by an arithmetic expression e which, since it must be closed at runtime, can be evaluated to a number i (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 $\mathcal{V}$. This is represented with the auxiliary judgment $\mathcal{V} \; ; \; C \vdash e :$ nat (implicitly proving that $e \geq 0$ under constraint C).

$$\frac{\mathcal{V} \; ; \; C \vdash e : \text{nat} \quad \mathcal{V} \; ; \; C \; ; \; \Delta \vdash^g P :: (x : A[e/n])}{\mathcal{V} \; ; \; C \; ; \; \Delta \vdash^g \text{send } x \; \{e\} \; ; \; P :: (x : \exists n. A)} \; \exists R$$

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

Operationally, the provider sends the arithmetic expression that the client receives and appropriately substitutes.

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

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

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

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

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

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

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

Table 2. Refined session types with operational description

and continues with P, while assume c $\{\phi\}$; Q *assumes* ϕ and continues with Q.

Thus, the typing rules for this new type constructor are

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

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

Notice how the provider must verify the truth of ϕ given the currently known constraints C ($\mathcal{V} \; ; \; C \vDash \phi$), while the client assumes ϕ by adding it to C. In well-typed configurations (which arise from executing well-typed processes, see Appendix C) the constraint ϕ in these rules will always be closed and true so there is no need to check this explicitly.

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

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

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

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

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

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

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

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

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 $\Delta_1 \Vdash_\Sigma \mathcal{S} :: \Delta_2$ denotes a set of semantic objects $\mathcal{S}$ using channels in Δ_1 and offering channels Δ_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* $\Delta_1 \Vdash_\Sigma \mathcal{S} :: \Delta_2$:

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

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 $A \multimap B$. Informally, two types are equal if they permit exactly the same communication behaviors. For example, if

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

we have that $\text{nat} \equiv \text{nat}'$ because each type allows any sequence of **succ** labels, followed by **zero** followed by *close*. This carries over to indexed types. If we define

$$\text{nat}[n] = \oplus\{\textbf{zero} : ?\{n = 0\}. \textbf{1}, \textbf{succ} : ?\{n > 0\}. \text{nat}[n-1]\}$$
$$\text{pos}[n] = \oplus\{\textbf{succ} : ?\{n > 0\}. \text{nat}[n-1]\}$$

then $\text{pos}[n+1] \equiv \text{nat}[n+1]$ 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* $\text{unfold}_\Sigma(A)$ *as*

$$\frac{V\overline{[n]} = A \in \Sigma}{\text{unfold}_\Sigma(V\overline{[e]}) = A[\overline{e}/\overline{n}]} \; \text{def} \qquad \frac{A \neq V\overline{[e]}}{\text{unfold}_\Sigma(A) = A} \; \text{str}$$

Like Gay and Hole, we require type definitions to be contractive, so the result of unfolding is never a type variable. Let *Type* be the set of all closed type expressions (no free variables). An interesting point in this definition is that constraint types with unsatisfiable constraints are related because neither can communicate any further messages.

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

- *If* $\text{unfold}_\Sigma(A) = \oplus\{\ell : A_\ell\}_{\ell \in L}$, *then* $\text{unfold}_\Sigma(B) = \oplus\{\ell : B_\ell\}_{\ell \in L}$ *and* $(A_\ell, B_\ell) \in \mathcal{R}$ *for all* $\ell \in L$.
- *If* $\text{unfold}_\Sigma(A) = A_1 \multimap A_2$, *then* $\text{unfold}_\Sigma(B) = B_1 \multimap B_2$ *and* $(A_1, B_1) \in \mathcal{R}$ *and* $(A_2, B_2) \in \mathcal{R}$.
- *If* $\text{unfold}_\Sigma(A) = !\{\phi\}. A'$, *then* $\text{unfold}_\Sigma(B) = !\{\psi\}. B'$ *and either* $\vDash \phi, \vDash \psi$, *and* $(A', B') \in \mathcal{R}$, *or* $\vDash \neg\phi$ *and* $\vDash \neg\psi$.
- *If* $\text{unfold}_\Sigma(A) = \exists m. A'$, *then* $\text{unfold}_\Sigma(B) = \exists n. B'$ *and for all* $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in \mathcal{R}$.

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

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 notation $\forall \mathcal{V}. C \Rightarrow A \equiv B$ 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* $\forall \mathcal{V}. C \Rightarrow A \equiv B$ *iff for all ground substitutions $\sigma : \mathcal{V}$ satisfying C (that is, $\cdot \vDash C[\sigma]$), we have $A[\sigma] \equiv B[\sigma]$.*

Theorem 2. *Checking* $\forall \mathcal{V}. C \Rightarrow A \equiv B$ *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 rule 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.

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

The ctr type outputs **lt** or **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 **le** exactly $\max(0, y - x)$ number of times terminating with **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 $\mathcal{V} ; C ; \Gamma \vdash A \equiv B$ to denote checking given variables $\mathcal{V}$ satisfying constraint C, whether types A and B are equal. Since the algorithm is coinductive, Γ stores equality constraints encountered so far. We initiate the algorithm with an empty Γ and $C = \top$, thereby checking $x, y ; \top ; \cdot \vdash \text{ctr}[x, y] \equiv \text{ctr}[x+1, y+1]$.

$$\frac{\begin{array}{c} V_1\overline{[v_1]} = A \in \Sigma \qquad V_2\overline{[v_2]} = B \in \Sigma \\ \gamma = \forall \mathcal{V}. C \Rightarrow V_1\overline{[e_1]} \equiv V_2\overline{[e_2]} \\ \mathcal{V} ; C ; \Gamma, \gamma \vdash A[\overline{e_1}/\overline{v_1}] \equiv B[\overline{e_2}/\overline{v_2}] \end{array}}{\mathcal{V} ; C ; \Gamma \vdash V_1\overline{[e_1]} \equiv V_2\overline{[e_2]}} \; \text{expd}$$

The expd rule adds the equality constraint $V_1\overline{[e_1]} \equiv V_2\overline{[e_2]}$ to Γ 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 Γ (α-renamed to avoid confusion) and

replace the type names with their definition. Next, we use the $\oplus$ rule below to explore each branch.

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Gamma \vdash A_\ell \equiv B_\ell \quad (\forall \ell \in L)}{\mathcal{V} \; ; \; C \; ; \; \Gamma \vdash \oplus\{\ell : A_\ell\}_{\ell \in L} \equiv \oplus\{\ell : B_\ell\}_{\ell \in L}} \; \oplus$$

First, we check the lt branch: $?\{x < y\}. \mathrm{ctr}[x + 1, y] \equiv ?\{x + 1 < y + 1\}. \mathrm{ctr}[x + 2, y + 1]$. We use the $?$ rule to check equivalence of the two constraints

$$\frac{\mathcal{V} \; ; \; C \vDash \phi \leftrightarrow \psi \quad \mathcal{V} \; ; \; C \wedge \phi \; ; \; \Gamma \vdash A \equiv B}{\mathcal{V} \; ; \; C \; ; \; \Gamma \vdash ?\{\phi\}. A \equiv ?\{\psi\}. B} \; ?$$

If the assumption of ϕ were to make the constraints contradictory we would succeed at this point, using the rule

$$\frac{\mathcal{V} \; ; \; C \vDash \bot}{\Sigma \; ; \; \mathcal{V} \; ; \; C \; ; \; \Gamma \vdash A \equiv B} \; \bot$$

in accordance with Definition 4. In our example, since $x < y \leftrightarrow x + 1 < y + 1$ and $x < y$ is consistent, we compare $\mathrm{ctr}[x + 1, y] \equiv \mathrm{ctr}[x + 2, y + 1]$. 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 $(\forall X, Y. \mathrm{ctr}[X, Y] \equiv \mathrm{ctr}[X+1, Y+1])$, 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.

$$\frac{\begin{array}{c} \forall \mathcal{V}'. \; C' \Rightarrow V_1\overline{[E_1]} \equiv V_2\overline{[E_2]} \in \Gamma \\ \mathcal{V} \; ; \; C \vDash \exists \mathcal{V}'. \; C' \wedge \overline{[E_1]} = \overline{[e_1]} \wedge \overline{[E_2]} = \overline{[e_2]} \end{array}}{\mathcal{V} \; ; \; C \; ; \; \Gamma \vdash V_1\overline{[e_1]} \equiv V_2\overline{[e_2]}} \; \mathrm{def}$$

For our example, this reduces to checking the validity of $\forall x, y. \exists X, Y. \; X = x + 1 \wedge Y = y \wedge X + 1 = x + 2 \wedge Y + 1 = y + 1$. Similarly, for the ge branch, we check if $?\{x \geq y\}. \mathbf{1} \equiv ?\{x + 1 \geq y + 1\}. \mathbf{1}$. Since the two constraints are equivalent and $\mathbf{1} \equiv \mathbf{1}$, we infer that the types match in both branches. Thus, we conclude that $\mathrm{ctr}[x, y] \equiv \mathrm{ctr}[x + 1, y + 1]$. 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 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 expd rule to the case where no assumption of the form $\forall \mathcal{V}'. \; C' \Rightarrow V_1\overline{[e_1']} \equiv V_2\overline{[e_2']}$ is already present in Γ. 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 Γ.

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* $\mathcal{V} \; ; \; C \; ; \; \cdot \vdash A \equiv B$, *then* $\forall \mathcal{V}. \; C \Rightarrow A \equiv B$.

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.

$$\frac{\mathcal{V} \; ; \; C \vDash e_1 = e_1' \wedge \ldots \wedge e_n = e_n'}{\mathcal{V} \; ; \; C \; ; \; \Gamma \vdash V\overline{[e]} = V\overline{[e']}} \; \mathrm{refl}$$

Traditional refinement languages such as DML [46] only use reflexivity as a criterion for equality of indexed type names. However, as exemplified in the ctr 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 inter-process 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 $\triangleright^r A$ and $\triangleleft^r A$ are used to exchange potential. The provider of $x : \triangleright^r A$ must *pay* r units of potential along x using process term (pay $x \; \{r\} \; ; \; P$), 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.

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

We allow potential q to refer to index variables in $\mathcal{V}$. The typing rules for $\triangleright^r A$ are

$$\frac{\mathcal{V} \; ; \; C \vDash q \geq r_1 = r_2 \quad \mathcal{V} \; ; \; C \; ; \; \Delta \vdash^{q-r_1} P :: (x : A)}{\mathcal{V} \; ; \; C \; ; \; \Delta \vdash^q \mathrm{pay} \; x \; \{r_1\} \; ; \; P :: (x : \triangleright^{r_2} A)} \; \triangleright R$$

$$\frac{\mathcal{V} \; ; \; C \vDash r_1 = r_2 \quad \mathcal{V} \; ; \; C \; ; \; \Delta, (x : A) \vdash^{q+r_1} Q :: (z : C)}{\mathcal{V} \; ; \; C \; ; \; \Delta, (x : \triangleright^{r_2} A) \vdash^q \mathrm{get} \; x \; \{r_1\} \; ; \; Q :: (z : C)} \; \triangleright L$$

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.

($\triangleright S$) $\text{proc}(c, w, \text{pay}\, c\, \{r\}\, ;\, P) \mapsto$
       $\text{proc}(c', w, P[c'/c]), \text{msg}(c, 0, \text{pay}\, c\, \{r\}\, ;\, c \leftarrow c')$
($\triangleright C$) $\text{msg}(c, w', \text{pay}\, c\, \{r\}\, ;\, c \leftarrow c'),\ \text{proc}(d, w,$
       $\text{get}\, c\, \{r\}\, ;\, Q) \mapsto \text{proc}(d, w + w', Q[c'/c])$

The dual type $\triangleleft^r A$ enables the provider to receive potential that is sent by its client.

$$\frac{\mathcal{V}\, ;\, C \models r_1 = r_2 \quad \mathcal{V}\, ;\, C\, ;\, \Delta \vdash^{g+r_1} P :: (x : A)}{\mathcal{V}\, ;\, C\, ;\, \Delta \vdash^{g} \text{get}\, x\, \{r_1\}\, ;\, P :: (x : \triangleleft^{r_2} A)}\ \triangleleft R$$

$$\frac{\mathcal{V}\, ;\, C \models q \geq r_1 = r_2 \quad \mathcal{V}\, ;\, C\, ;\, \Delta, (x : A) \vdash^{g-r_1} Q :: (z : C)}{\mathcal{V}\, ;\, C\, ;\, \Delta, (x : \triangleleft^{r_2} A) \vdash^{g} \text{pay}\, x\, \{r_1\}\, ;\, Q :: (z : C)}\ \triangleleft L$$

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.

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

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

$$\frac{\mathcal{V}\, ;\, C \models q \geq r \quad \mathcal{V}\, ;\, C\, ;\, \Delta \vdash^{g-r} P :: (x : A)}{\mathcal{V}\, ;\, C\, ;\, \Delta \vdash^{g} \text{work}\, \{r\}\, ;\, P :: (x : A)}\ \text{work}$$

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.

$\text{queue}_A[n] = \&\{\textbf{ins} : \triangleleft^{2n}(A \multimap \text{queue}_A[n + 1]),$
      $\textbf{del} : \triangleleft^2 \oplus \{\textbf{none} : ?\{n = 0\}.\, \mathbf{1},$
            $\textbf{some} : ?\{n > 0\}.\, A \otimes \text{queue}_A[n - 1]\}\}$
$\cdot \vdash^{0} empty :: (s : \text{queue}_A[0])$
$(x : A), (t : \text{queue}_A[n]) \vdash^{0} elem[n] :: (s : \text{queue}_A[n + 1])$

Thus, the queue requires $2n$ units of potential to insert and 2 units of potential to delete an element. The $2n$ units are used to carry the **ins** message and the element to insert to the end of the queue. While deletion, the process sends 2 messages, either **none** and close (lines 7,9 in Figure 2) or **some** and the element stored (lines 17,19). Operationally, when the *elem* process receives $2(n + 1)$ units on s, it consumes 2 units

$$\frac{\mathcal{V}\, ;\, C \models \phi \quad \mathcal{V}\, ;\, C\, ;\, \Delta \vdash_i^{g} P :: (x : A)}{\mathcal{V}\, ;\, C\, ;\, \Delta \vdash_i^{g} P :: (x : ?\{\phi\}.\, A)}\ ?R$$

$$\frac{\mathcal{V}\, ;\, C \wedge \phi\, ;\, \Delta, (x : A) \vdash_i^{g} Q :: (z : C)}{\mathcal{V}\, ;\, C\, ;\, \Delta, (x : ?\{\phi\}.\, A) \vdash_i^{g} Q :: (z : C)}\ ?L$$

Figure 4. Implicit Typing Rules

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 2 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 ℓ is extended by a branch $\ell \Rightarrow$ 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 $\mathcal{V}\, ;\, C\, ;\, \Delta \vdash_i^{g} P :: (x : A)$ using the rules in Figure 4 (analogous rules $!R, !L, \triangleleft R, \triangleleft L, \triangleright R, \triangleright L$ 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 $!R, ?L, \triangleleft R, \triangleright L$ 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 $\mathcal{V} ; C ; \Delta ; \Omega \vdash^{g} P :: (x : A)$. The context is split into two: the linear context Δ contains stable propositions on which the invertible left rules have been applied, while the ordered context Ω stores channels on which invertible rules can possibly still be applied to. First, we assign polarities to the type operators with implicit expressions, a notion borrowed from focusing [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.

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

Not all types can be polarized in this manner. For example, $!\{\phi\}.\, ?\{\psi\}.\, A$ or $\triangleright^{r} \triangleleft^{q} A$. 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 $\triangleright$ operators are positive, while ! and $\triangleleft$ are negative. The structural types, denoted by S are considered neutral. In the forcing calculus, the invertible rules are applied first (analogous $?L, \triangleright L$ omitted).

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

$$
\frac{\mathcal{V} ; C ; \Delta^{-} ; \Omega \vdash^{g+r} P :: (x : A^{-})}{\mathcal{V} ; C ; \Delta^{-} ; \Omega \vdash^{g} P :: (x : \triangleleft^{r} A^{-})} \; \triangleleft R
$$

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

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

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

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

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

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

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

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

$$
\frac{\mathcal{V} ; C \vDash q \geq r \qquad \mathcal{V} ; C ; \Delta^{-} ; \cdot \vdash^{g-r} P :: [x : A^{+}]}{\mathcal{V} ; C ; \Delta^{-} ; \cdot \vdash^{g} P :: [x : \triangleright^{r} A^{+}]} \; \triangleright R
$$

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

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

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

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

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

Theorem 4 (Soundness and Completeness). *For (valid) polarized types A and context Δ we have:*

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

Module	iLOC	eLOC	#Defs	R (ms)	T (ms)
arithmetic	69	143	8	0.353	1.325
integers	90	114	8	0.200	1.074
linlam	54	67	6	0.734	4.003
list	244	441	29	1.534	3.419
primes	90	118	8	0.196	1.646
segments	48	65	9	0.239	0.195
ternary	156	235	16	0.550	1.967
theorems	79	141	16	0.361	0.894
tries	147	308	9	1.113	5.283
Total	**977**	**1632**	**109**	**5.280**	**19.806**

Table 3. Case Studies

Proof. Proof of 1. follows by induction on the implicit typing judgment. Proof of 2. follows by induction on the forcing judgment. See Appendix D for the full rules and proof. □

If a process is well-typed in the implicit system, it is well-typed using the forcing calculus. Reading off the process expression from the typing derivation in the forcing calculus results in the corresponding explicit program. 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 implemented a protoype for the language in Standard ML (about 6500 lines of code) available open-source that closely adheres to the theory presented here. Command line options determine whether to use explicit or implicit syntax, and the result of reconstruction can be displayed if desired. We use a straightforward implementation of Cooper's algorithm [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 $x = e$ by substituting e for x in order to reduce the number of variables. For the implementation of type equality we assign internal names to subexpressions parameterized over their free index variables in order to facilitate efficient loop detection. After checking the validity of types, the implementation reconstructs missing branches, then constraints, and finally work. Verifying constraints is postponed to the final pass of type-checking the reconstructed process expression.

We have a variety of 9 case studies implemented, totaling about 1000 lines of code type-checked in our implementation. Table 3 describes the results: iLOC describes the lines of source code in implicit syntax, eLOC describes the lines of code after reconstruction (which inserts implicit constructs), #Defs shows the number of process definitions, R (ms) and T (ms) show the reconstruction and type-checking time in milliseconds respectively. The experiments were run on an

Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory. We briefly describe each case study.

1. **arithmetic**: natural numbers in unary and binary representation indexed by their value and processes implementing standard arithmetic operations.
2. **integers**: an integer counter represented using two indices x and y with value $x - y$.
3. **linlam**: expressions in the linear λ-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 $\text{seg}[n] = \forall k. \text{list}[k] \multimap \text{list}[n + k]$ representing partial lists with constant-work append operation.
7. **ternary**: natural numbers represented in balanced ternary form with digits $0, 1, -1$, indexed by their value, and some standard operations on them.
8. **theorems**: processes representing (circular [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 ergometric types.

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

Queue as Two Lists The running $\text{queue}_A[n]$ 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 ergometric types support amortized analysis, we obtain the constant cost type

$\text{queue}_A[n] = \&\{\textbf{ins} : \triangleleft^6(A \multimap \text{queue}_A[n + 1]),$
$\qquad \textbf{del} : \triangleleft^4 \oplus \{\textbf{none} : ?\{n = 0\}.\, \mathbf{1},$
$\qquad\qquad \textbf{some} : ?\{n > 0\}.\, A \otimes \text{queue}_A[n - 1]\}\}$

Binary Numbers We can represent binary numbers indexed by their value as sequences of bits **b0** and **b1** followed by **e** to indicate the end of the sequence.

$\text{bin}[n] = \oplus\{\ \textbf{b0} : ?\{n > 0\}.\, \exists k.\, ?\{n = 2 * k\}.\, \text{bin}[k],$
$\qquad \textbf{b1} : ?\{n > 0\}.\, \exists k.\, ?\{n = 2 * k + 1\}.\, \text{bin}[k],$
$\qquad \textbf{e} : ?\{n = 0\}.\, \mathbf{1}\ \}$

A binary number on outputting **b0** (or **b1**) must send a proof of $n > 0$ (no leading zeros) and a witness k such that $n = 2k$ (resp. $n = 2k + 1$) before continuing with $\text{bin}[k]$. While outputting **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

$(m : \text{bin}[n + 1]) \vdash pred[n] :: (n : \text{bin}[n])$

they can skip the impossible case of label **e** (since $n + 1 \neq 0$).

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

$$\exp[n] = \oplus\{\textbf{lam} : ?\{n > 0\}. \forall n_1. \exp[n_1] \multimap \exp[n_1 + n - 1],$$
$$\textbf{app} : \exists n_1. \exists n_2. ?\{n = n_1 + n_2 + 1\}. \exp[n_1] \otimes \exp[n_2]\}$$

An expression is either a λ (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 λ-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 λ expression

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

so the **app** label is not permitted. Type checking verifies that that the result of evaluating a linear λ-term is no larger than the original term.

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

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 λ-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 π-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 π-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 multiset 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 non-linear 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 ergometric types.

Acknowledgments

References

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

[2] Pedro Baltazar, Dimitris Mostrous, and Vasco T. Vasconcelos. 2012. Linearly Refined Session Types. *Electronic Proceedings in Theoretical Computer Science* 101 (Nov 2012), 38–49. https://doi.org/10.4204/eptcs.101.4

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

[4] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2019. Asynchronous Timed Session Types. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 583–610.

[5] Luís Caires and Frank Pfenning. 2010. Session Types As Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR'10)*. Springer-Verlag, Berlin, Heidelberg, 222–236. http://dl.acm.org/citation.cfm?id=1887654.1887670

[6] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 760 (11 2014). https://doi.org/10.1017/S0960129514000218

[7] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 316–329. https://doi.org/10.1145/3009837.3009858

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

[9] Ezgi Çiçek, Deepak Garg, and Umut Acar. 2015. Refinement Types for Incremental Computational Complexity. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–431.

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

[11] Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. Association for Computing Machinery, New York, NY, USA, 184–198. https://doi.org/10.1145/325694.325716

[12] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. IEEE Computer Society, USA, 133–142. https://doi.org/10.1109/LICS.2011.22

[13] Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. *SIGPLAN Not.* 48, 1 (Jan. 2013), 167–178. https://doi.org/10.1145/2480359.2429090

[14] Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. *SIGPLAN Not.* 43, 1 (Jan. 2008), 133–144. https://doi.org/10.1145/1328897.1328457

[15] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2019. Resource-Aware Session Types for Digital Contracts. arXiv:cs.PL/1902.06056

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

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

[18] Farzaneh Derakhshan and Frank Pfenning. 2019. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *arXiv e-prints*, Article arXiv:1908.01909 (Aug 2019), arXiv:1908.01909 pages. arXiv:cs.LO/1908.01909

[19] Juliana Franco and Vasco Thudichum Vasconcelos. 2014. A Concurrent Programming Language with Refined Session Types. In *Software Engineering and Formal Methods*, Steve Counsell and Manuel Núñez (Eds.). Springer International Publishing, Cham, 15–28.

[20] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

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

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

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

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

[25] Dennis Griffith and Elsa L. Gunter. 2013. LiquidPi: Inferrable Dependent Session Types. In *NASA Formal Methods*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–197.

[26] Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. *Proc. ACM Program. Lang.* 4, POPL, Article Article 24 (Dec. 2019), 27 pages.

https://doi.org/10.1145/3371092

[27] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

[28] Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. Springer-Verlag New York, Inc., New York, NY, USA, 132–157. https://doi.org/10.1007/978-3-662-46669-8_6

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

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

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

[32] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 253–268. https://doi.org/10.1145/3314221.3314602

[33] Sam Lindley and J. Garrett Morris. 2016. Talking Bananas: Structural Recursion for Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 434–447. https://doi.org/10.1145/2951913.2951921

[34] Chris Okasaki. 1995. Simple and efficient purely functional queues and deques. *Journal of Functional Programming* 5, 4 (1995), 583–592. https://doi.org/10.1017/S0956796800001489

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

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

[37] Weihao Qu, Marco Gaboardi, and Deepak Garg. 2019. Relational Cost Analysis for Functional-Imperative Programs. *Proc. ACM Program. Lang.* 3, ICFP, Article Article 92 (July 2019), 29 pages. https://doi.org/10.1145/3341696

[38] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[39] RE Tarjan. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318.

[40] Peter Thiemann and Vasco T. Vasconcelos. 2019. Label-Dependent Session Types. *Proc. ACM Program. Lang.* 4, POPL, Article Article 67 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371135

[41] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent Session Types via Intuitionistic Linear Type Theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP '11)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/2003476.2003499

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

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

[44] Philip Wadler. 2012. Propositions as Sessions. *SIGPLAN Not.* 47, 9 (Sept. 2012), 273–286. https://doi.org/10.1145/2398856.2364568

[45] Hanwen Wu and Hongwei Xi. 2017. Dependent Session Types. *CoRR* abs/1704.07004 (2017). arXiv:1704.07004 http://arxiv.org/abs/1704.07004

[46] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

[47] Hongwei Xi, Dengping Zhu, and Yanka Li. 1998. Applied Type System with Stateful Views. In *Carnegie Mellon University*. e, 189.

[48] Christoph Zenger. 1997. Indexed Types. *Theor. Comput. Sci.* 187, 1–2 (Nov. 1997), 147–165. https://doi.org/10.1016/S0304-3975(97)00062-5

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

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

Abstract Syntax	Concrete Syntax
$\oplus\{l_1 : A_1, l_2 : A_2\}$	`+{l1 : A1, l2 : A2}`
$\&\{l_1 : A_1, l_2 : A_2\}$	`&{l1 : A1, l2 : A2}`
$A \otimes B$	`A * B`
$A \multimap B$	`A -o B`
$\mathbf{1}$	`1`
$\exists n.\, A$	`?n. A`
$\forall n.\, A$	`!n. A`
$?\{n = 0\}.\, A$	`?{n = 0}. A`
$!\{n = 0\}.\, A$	`!{n = 0}. A`
$\triangleright^r A$	`|{r}> A`
$\triangleleft^r A$	`<{r}| A`
$V[n_1, n_2]$	`V{n1}{n2}`
$x.k$	`x.k`
case $x\,(l \Rightarrow P)$	`case x (l => P)`
send $x\,w$	`send x w`
$y \leftarrow$ recv x	`y <- recv x`
close x	`close x`
wait x	`wait x`
send $x\,\{e\}$	`send x {e}`
$\{n\} \leftarrow$ recv x	`{n} <- recv x`
assert $x\,\{n = 0\}$	`assert x {n = 0}`
assume $x\,\{n = 0\}$	`assume x {n = 0}`
pay $x\,\{r\}$	`pay x {r}`
get $x\,\{r\}$	`get x {r}`

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.

```
type V{n} = A
decl f{n1}{n2}: (x1 : A1) ... (xn : An) |{q}- (x : A)
proc x <- f{n1}{n2} <- x1 ... xn = P
```

The first line shows a type definition of V with one index n defined with type expression A. The second line shows a process declaration: process f with two indices n_1 and n_2, channels $x_1 \ldots x_n$ in its context with types $A_1 \ldots A_n$ respectively, storing potential q, and offering along channel x of type A. The last line shows the process definition of f offering channel x and using channels $x_1 \ldots x_n$ with P being a process expression.

A.1 Lists with Potential

The type list$[n, p]$ is the type of lists of length n where each element carries potential p. Since we do not have polymorphism, we use elements of type nat to stand in for an arbitrary type. We start with the basic, unindexed version

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

```
type list = +{ cons : nat * list,
               nil : 1 }
```

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:

```
cons, a3, cons, a4, cons, a5, 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 transfered to the client of the list before each element.

```
type list{n}{p} =
+{ cons : ?{n > 0}. |{p}> nat * list{n-1}{p},
   nil : ?{n = 0}. 1 }
```

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

```
cons, assert {3 > 0}, pay 2, a3,
cons, assert {2 > 0}, pay 2, a4,
cons, assert {1 > 0}, pay 2, a5,
nil, assert {0 = 0}, close
```

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

```
decl nil{p} : . |{2}- (l : list{0}{p})
```

```
proc l <- nil{p} <- = l.nil ; close l
```

Reconstruction will add the assertion of $0 = 0$ 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.

```
proc l <- nil{p} <-  =
  work ;
  l.nil ;
  assert l {0 = 0} ;
  work ;
  close l
```

A cons *process* should take an element x and a list t and provide the list that sends **cons**, then x, and then behaves like t. This time, let's first examine the code before the type:

```
proc l <- cons{n}{p} <- x t =
  l.cons ;
  send l x ;
```

```
    l <- 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

```
decl cons{n}{p} :
(x:nat) (t : list{n}{p}) |{p+2}- (l : list{n+1}{p})
```

If we make a mistake, for example, state the potential as $p + 1$ we get an error message:

```
error:insufficient potential: true |/= p+1-1 >= 1
proc l <- cons{n}{p} <- x t =
l.cons ; send l x ; l <- t
          ~~~~~~~~
```

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.

```
proc l <- append{n}{k}{p} <- l1 l2 =
  case l1
    ( cons => x <- recv l1 ;
              l.cons ; send l x ;
              l <- append{n-1}{k}{p} <- l1 l2
    | nil => wait l1 ; l <- l2 )
```

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

```
decl append{n}{k}{p} :
(l1 : list{n}{p+2}) (l2 : list{k}{p}) |-
     (l : 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.

```
decl append{n}{k}{p} :
(l1 : list{n}{p}) (l2 : list{k}{p}) |{2*n}-
     (l : 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.

```
decl rev{n}{k}{p} :
(l : list{n}{p+2}) (a : list{k}{p}) |-
     (r : list{n+k}{p})


proc r <- rev{n}{k}{p} <- l a =
  case l ( cons => x <- recv l ;
                   a' <- cons{k}{p} <- x a ;
                   r <- rev{n-1}{k+1}{p} <- l a'
         | nil => wait l ; r <- a )
```

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.

```
decl reverse{n}{p} : (l : list{n}{p+2}) |{2}-
     (r : list{n}{p})
proc r <- reverse{n}{p} <- l =
  a <- nil{p} <- ;
  r <- rev{n}{0}{p} <- 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 l.

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*.

```
decl charge2{n}{p} :
    (k : list{n}{p}) |{4*n+2}- (l : list{n}{p+2})


proc l <- charge2{n}{p} <- k =
  case k ( cons => x <- recv k ;
                   l.cons ; send l x ;
```

```
                    l <- charge2{n-1}{p} <- k
        | nil => wait k ; l.nil ; close l )
```

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} :
    (k : list{n}{p}) |{(q+2)*n+2}- (l : list{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

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

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 =
  case k ( cons => x <- recv k ;
                   m.next ; send m x ;
                   y <- recv m ;
                   l.cons ; send l y ;
                   l <- map{n-1}{p} <- k m
         | nil => wait k ;
                  m.done ; wait m ;
                  l.nil ; close l )
```

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} :
(k : list{n}{p+4}) (m:mapper) |{3}- (l : list{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 $\text{bdd\_list}[n, p] = \exists m. \,?\{m \leq n\}. \,\text{list}[m, p]$.

```
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{0}{p})
decl bdd_cons{n}{p} :
  (x : nat) (k : bdd_list{n}{p}) |{p+2}-
    (l : bdd_list{n+1}{p})


decl bdd_resize{n}{p} :
  (k : bdd_list{n}{p}) |- (l : bdd_list{n+1}{p})


proc l <- bdd_resize{n}{p} <- k =
  {m} <- recv k ;
  send l {m} ;
  l <- k
```

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.

$$\text{selector}_A = \&\{\textbf{next} : A \multimap \oplus\{\textbf{false} : \text{selector}_A,$$
$$\textbf{true} : A \otimes \text{selector}_A\}$$
$$\textbf{done} : \mathbf{1}\}$$

Concretely (using nat for A):

```
type selector =
&{ next : nat -o +{ false : selector,
                    true : nat * selector },
   done : 1 }
```

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 enqueuing and dequeuing 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 -o 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 $\triangleleft^6$ and $\triangleleft^4$ which mean that the client has to transfer this potential, while in lists we used $\triangleright^p$ so the list provider payed 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 λ-Calculus

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

Ignoring issues of size for the moment, λ-calculus expressions are represented with the type exp, values are of type val.

$$exp = \oplus\{ \textbf{lam} : exp \multimap exp,$$
$$\textbf{app} : exp \otimes exp \}$$

$$val = \oplus\{ \textbf{lam} : exp \multimap exp \}$$

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 λ-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

```
decl id : . |- (e : exp)
proc e <- id <- =
  e.lam ;
  x <- recv e ;
  e <- x
```

The application of the identity function to itself:

```
decl idid : . |- (e : exp)
proc e <- idid <- =
  i1 <- id <- ;
  i2 <- id <- ;
  e.app ; send e i1 ; e <- i2
```

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

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

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

Evaluation of a λ-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.

```
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 $e_1\,e_2$ will first send e_1 : exp and then behave like e_2. So what we send to v_1 two lines below in fact represents e_2.

To track the sizes of λ-expressions we need quantifiers. For example, an application $e_1\,e_2$ has size n if there exist sizes n_1 and n_2 such that $n = n_1 + n_2 + 1$ and of e_1 and e_2 have size n_1 and n_2, 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 n_1 and then an expression of size n_1.

These considerations lead us to the types

$\exp[n] =$
$\oplus\{\,\mathbf{lam} : ?\{n > 0\}.\,\forall n_1.\,\exp[n_1] \multimap \exp[n_1 + n - 1],$
$\quad \mathbf{app} : \exists n_1.\,\exists n_2.\,?\{n = n_1 + n_2 + 1\}.\,\exp[n_1] \otimes \exp[n_2]\,\}$
$\mathsf{val}[n] =$
$\oplus\{\,\mathbf{lam} : ?\{n > 0\}.\,\forall n_1.\,\exp[n_1] \multimap \exp[n_1 + n - 1]\,\}$

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 <= 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

$$
\begin{aligned}
\overline{0} &= \mathbf{e},\ close \\
\overline{2n} &= \mathbf{b0}, \overline{n} \qquad n > 0 \\
\overline{2n+1} &= \mathbf{b1}, \overline{n}
\end{aligned}
$$

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 v <- eval{n} <- e =
  case e
    ( lam => send v {n} ;
               v <- lambda{n-1} <- e
    | app => {n1} <- recv e ;
               {n2} <- recv e ;            % n = n1 + n2 + 1
               e1 <- recv e ;              % e1 : exp{n1}, e = e2 : exp{n2}
               v1 <- eval{n1} <- e1 ;
               {k2} <- recv v1 ;           % v1 : val{k2} for some k2 <= n1
               case v1
                ( lam => send v1 {n2} ;
                           send v1 e ;    % v1 : exp{n2+k2-1}
                           v2 <- eval{n2+k2-1} <- v1 ; % v2 : val{l} for some l <= n2+k2-1 <= n2+n1-1 = n-2
                           {l} <- recv v2 ;
                           send v {l} ; v <- v2
                )
    )
```

Figure 7. Evaluating expressions in linear λ-calculus

bin $= \oplus\{\mathbf{b0} : \text{bin}, \mathbf{b1} : \text{bin}, \mathbf{e} : \mathbf{1}\}$

However, this representation does not enforce the absence of leading zeros. One way to enforce that is by indexing this type by the natural number it represents. However, we would have to be able to express that a number is even (which is possible in Presburger arithmetic) since even$(n) \triangleq \exists k. \, 2*k = n$.

bin$[n] = \oplus\{\mathbf{b0} : ?\{n > 0 \land \text{even}(n)\}. \, \text{bin}[n/2], \ldots\}$

However, the value of the remaining sequence of bits would have to be $n/2$ and that function is neither available nor expressible. Instead, we use a quantifier at the type level to give a name to $n/2$:

bin$[n] = \oplus\{$ $\mathbf{b0} : ?\{n > 0\}. \, \exists k. \, ?\{n = 2 * k\}. \, \text{bin}[k],$
$\quad\quad\quad \mathbf{b1} : \exists k. \, ?\{n = 2 * k + 1\}. \, \text{bin}[k],$
$\quad\quad\quad \mathbf{e} : ?\{n = 0\}. \, \mathbf{1} \}$

Zero and Successor Processes   Now we can straightforwardly express zero and the successor processes and verify their correctness, but we pay the price of having to send and receive k. In the code below name this index variable n' to represent $\lfloor n/2 \rfloor$. We receive the bits along channel x and send the bits for $n + 1$ along channel y. If we receive $\mathbf{b0}$ we just flip the bit to $\mathbf{b1}$, send it to the output, and forward all the remaining bit. If we receive $\mathbf{b1}$ we flip it to $\mathbf{b0}$ but we have to call succ recursively to account for the "carry". If we receive $\mathbf{e}$ we output the representation of 1.

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

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

proc y <- succ{n} <- x =
```

```
case x ( b0 => {n'} <- recv x ;
                  y.b1 ; send y {n'} ;
                  y <- x
       | b1 => {n'} <- recv x ;
                  y.b0 ; send y {n'+1} ;
                  y <- succ{n'} <- x
       | e => y.b1 ; send y {0} ;
                  y.e ; wait x ; close y )
```

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 ϵ operator or definite description ι, 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{n|n > 0} : (x : bin{n}) |- (y : bin{n-1})
proc y <- pred{n} <- x =
  case x
    ( b0 => {n'} <- recv x ;     % 2*k-1 = 2*(k-1)+1
            y.b1 ; send y {n'-1} ;
            y <- pred{n'} <- x
    | b1 => {n'} <- recv x ;
            y <- dbl0{n'} <- x   % 2*k+1-1 = 2*k
    % no case for e
    )
```

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 dbl0 is a process with type

```
decl dbl0{n} : (x : bin{n}) |- (y : bin{2*n})
```

which is not as trivial as one might think. In fact, if we try

```
proc y <- dbl0{n} <- x =
  y.b0 ; send y {n} ;
```

we get the error message

```
error:assertion not entailed: true |/= 2*n > 0
  y.b0 ; send y {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 <- dbl0{n} <- x =
  case x ( b0 => {n'} <- recv x ;
                 y.b0 ; send y {n} ;
                 y.b0 ; send y {n'} ;
                 y <- x
         | b1 => {n'} <- recv x ;
                 y.b0 ; send y {n} ;
                 y.b1 ; send y {n'} ;
                 y <- x
         | e => y.e ; wait x ; close y )
```

```
decl dbl1{n} : (x : bin{n}) |- (y : bin{2*n+1})
proc y <- dbl1{n} <- x =
  y.b1 ; send y {n} ;
  y <- x
```

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 drop{n} : (x : bin{n}) |- (u : 1)
proc u <- drop{n} <- x =
  case x ( b0 => {n'} <- recv x ; u <- drop{n'} <- x
         | b1 => {n'} <- recv x ; u <- drop{n'} <- x
         | e => wait x ; close u )
```

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

```
type ord{m}{n} = +{ lt : ?{m < n}. 1,
                    eq : ?{m = n}. 1,
                    gt : ?{m > n}. 1 }
```

Then the type

```
decl compare{m}{n} :
(x : bin{m}) (y : bin{n}) |- (o : ord{m}{n})
```

guarantees the correctness of compare because it can produce **lt** only if $m < n$, and similarly for **eq** and **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.

```
proc o <- compare{m}{n} <- x y =
  case x
    ( b0 => {m'} <- recv x ;
            case y
              ( b0 => {n'} <- recv y ;
                      o <- compare{m'}{n'} <- x y ;
                      ... ) )
```

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

$$m, n, m', n' ;$$
$$m' > 0 \land m = 2 * m' \land n' > 0 \land n = 2 * n' ; \cdot \vdash$$
$$\text{ord}[m, n] \equiv \text{ord}[m', n']$$

This is certainly true declaratively (according to the definition of type bisimulation) because $2 * m' < 2 * n'$ iff $m' < n'$ 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.

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

decl zero : . |- (x : nat{0})
decl succ{n} : (x : nat{n}) |- (y : nat{n+1})
decl drop{n} : (x : nat{n}) |- (u : 1)
decl dup{n} :
(x : nat{n}) |- (xx : nat{n} * nat{n} * 1)
```

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

```
type stream{k} = +{prime : ?{k > 0}. stream{k-1},
                   composite : ?{k > 0}. stream{k-1},
                   end : ?{k = 0}. 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 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 **prime**, because initially all natural numbers starting at 2 are candidates to be primes.

```
decl candidates{n} : (x : nat{n}) |- (s : stream{n})
proc s <- candidates{n} <- x =
  case x ( succ => s.prime ;
                   s <- candidates{n-1} <- x
         | zero => wait x ;
                   s.end ; close s )
```

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 **composite**. The bit **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 : 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 $n + k$ 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.

```
decl head{k}{n}{kn|kn = k+n} :
(t : stream{k}) (x : nat{n}) |- (s : stream{k})

proc s <- head{k}{n}{kn} <- t x =
  case t
    ( prime => s.prime ; % not divisible: new prime
               z <- zero <- ;
               xx <- dup{n} <- x ;
               x1 <- recv xx ;
               x2 <- recv xx ;
               wait xx ;
               f <- filter{k-1}{n}{0}{n} <- t x1 z ;
               x' <- succ{n} <- x2 ;
```

```
                s <- head{k-1}{n+1}{kn} <- f x'
  | composite => s.composite ;
                x' <- succ{n} <- x ;
                s <- head{k-1}{n+1}{kn} <- t x'
  | end => wait t ;
          u <- drop{n} <- x ; wait u ;
          s.end ; close s )
```

Finally, the filter process filter itself. It implements a cyclic counter as two natural numbers c and d 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 deallocate 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 although 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*k}. |{p}> bin{k}{p},
     b1 : ?{n > 0}. ?k. ?{n = 2*k+1}. |{p}> 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**.

```
decl empty :
. |- (c : ctr{0})

decl bit0{n|n > 0} :
(d : ctr{n}) |{2}- (c : ctr{2*n})

decl bit1{n} :
(d : ctr{n}) |{3}- (c : ctr{2*n+1})

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

proc c <- bit0{n} <- d =
  case c ( inc => c <- bit1{n} <- d
         | val => c.b0 ; send c {n};
                  d.val ; c <- d )

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

The type-checker 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 $m \leq n$: 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 subtrie, 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|n = n1+n2} : (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'
                           | 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 example 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.

$$\text{trie}[n] =$$
$$\&\{ \mathbf{ins} : \triangleleft^4 \forall k.\, \text{bin}[k, 5] \multimap \text{trie}[n + 1],$$
$$\quad \mathbf{del} : \triangleleft^5 \forall k.\, \text{bin}[k, 5] \multimap \exists m.\, ?\{m \le n\}.$$
$$\text{bin}[m, 0] \otimes \text{trie}[n - m] \}$$

```
type trie{n} =
  &{ ins : <{4}| !k. bin{k}{5} -o trie{n+1},
     del : <{5}| !k. bin{k}{5} -o ?m. ?{m <= n}.
                               bin{m}{0} * trie{n-m} }
```

Trie Implementation We have two kinds of nodes: leaf nodes (process leaf[0]) not holding any elements and element nodes (process nodes[n_0, m, n_1]) representing an element of multiplicity m with n_0 and n_1 elements in the left and right subtries, respectively. A node therefore has type trie[$n_0 + m + n_1$]. Neither process carries any potential.

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

```
(t : trie{n0+m+n1})
```

The code 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}{1}{0} <- l c0 r )
    | del => {k} <- recv t ;
             x <- recv t ;
```

```
            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}{n1+1} <- l c r
             | e => wait x ;
                    c.inc ;
                    t <- node{n0}{m+1}{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	$::=$	$\oplus\{\ell : A\}_{\ell \in L} \mid \&\{\ell : A\}_{\ell \in L}$
			$\mid\quad A \otimes A \mid A \multimap A \mid \mathbf{1} \mid V\overline{[e]}$
			$\mid\quad ?\{\phi\}.\,A \mid !\{\phi\}.\,A \mid \exists n.\,A \mid \forall n.\,A$
Arith. Exps.	e	$::=$	$i \mid e + e \mid e - e \mid i \times e \mid n$
Arith. Props.	ϕ	$::=$	$e = e \mid e > e \mid \top \mid \bot \mid \phi \wedge \phi$
			$\mid\quad \phi \vee \phi \mid \neg\phi \mid \exists n.\,\phi \mid \forall n.\,\phi$

B.1 Type Equality Definition

Definition 5. *For a type A, define $\mathrm{unfold}_\Sigma(A)$ in the presence of signature Σ:*

$$\frac{V\overline{[v]} = A \in \Sigma}{\mathrm{unfold}_\Sigma(V\overline{[e]}) = A[\overline{e}/\overline{v}]}\ \mathrm{def} \qquad \frac{A \neq V\overline{[e]}}{\mathrm{unfold}_\Sigma(A) = A}\ \mathrm{str}$$

Definition 6. *Let Type be the set of all ground session types. A relation $\mathcal{R} \subseteq Type \times Type$ is a type simulation if $(A, B) \in \mathcal{R}$ implies the following conditions:*

- *If $\mathrm{unfold}_\Sigma(A) = \oplus\{\ell : A_\ell\}_{\ell \in L}$, then $\mathrm{unfold}_\Sigma(B) = \oplus\{\ell : B_\ell\}_{\ell \in L}$ and $(A_\ell, B_\ell) \in \mathcal{R}$ for all $\ell \in L$.*
- *If $\mathrm{unfold}_\Sigma(A) = \&\{\ell : A_\ell\}_{\ell \in L}$, then $\mathrm{unfold}_\Sigma(B) = \&\{\ell : B_\ell\}_{\ell \in L}$ and $(A_\ell, B_\ell) \in \mathcal{R}$ for all $\ell \in L$.*
- *If $\mathrm{unfold}_\Sigma(A) = A_1 \multimap A_2$, then $\mathrm{unfold}_\Sigma(B) = B_1 \multimap B_2$ and $(A_1, B_1) \in \mathcal{R}$ and $(A_2, B_2) \in \mathcal{R}$.*
- *If $\mathrm{unfold}_\Sigma(A) = A_1 \otimes A_2$, then $\mathrm{unfold}_\Sigma(B) = B_1 \otimes B_2$ and $(A_1, B_1) \in \mathcal{R}$ and $(A_2, B_2) \in \mathcal{R}$.*
- *If $\mathrm{unfold}_\Sigma(A) = \mathbf{1}$, then $\mathrm{unfold}_\Sigma(B) = \mathbf{1}$.*
- *If $\mathrm{unfold}_\Sigma(A) = ?\{\phi\}.\,A'$, then $\mathrm{unfold}_\Sigma(B) = ?\{\psi\}.\,B'$ and either $\models \phi$, $\models \psi$, and $(A', B') \in \mathcal{R}$ or $\models \neg\phi$ and $\models \neg\psi$.*
- *If $\mathrm{unfold}_\Sigma(A) = !\{\phi\}.\,A'$, then $\mathrm{unfold}_\Sigma(B) = !\{\psi\}.\,B'$ and either $\models \phi$, $\models \psi$, and $(A', B') \in \mathcal{R}$, or $\models \neg\phi$ and $\models \neg\psi$.*
- *If $\mathrm{unfold}_\Sigma(A) = \exists m.\,A'$, then $\mathrm{unfold}_\Sigma(B) = \exists n.\,B'$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in \mathcal{R}$.*
- *If $\mathrm{unfold}_\Sigma(A) = \forall m.\,A'$, then $\mathrm{unfold}_\Sigma(B) = \forall n.\,B'$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in \mathcal{R}$.*

Definition 7. *The type equality relation $\equiv_\mathcal{R}$ is defined by $A \equiv_\mathcal{R} B$ iff $\mathcal{R}$ is a type simulation and $(A, B) \in \mathcal{R}$.*

Definition 8. *The coinductive type equality definition $\equiv$ is defined by $A \equiv B$ iff there exists a relation $\mathcal{R}$ such that $A \equiv_\mathcal{R} B$.*

Lemma 1. *The relation $\equiv$ is reflexive.*

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

Lemma 2. *The relation $\equiv$ is symmetric.*

Proof. Consider $A \equiv B$. There exists a type simulation $\mathcal{R}_1$ such that $(A, B) \in \mathcal{R}_1$. Define $\mathcal{R}_2$ as follows.

$$\mathcal{R}_2 = \{(B, A) \mid (A, B) \in \mathcal{R}_1\}$$

It is easy to see that $\mathcal{R}_2$ is a type simulation since all invariants in Definition 6 are symmetric. $\square$

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 $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $(A_1, A_2) \in \mathcal{R}_1$ and $(A_2, A_3) \in \mathcal{R}_2$. Denoting relational composition by $\mathcal{R}_1 \cdot \mathcal{R}_2$,

$$\begin{aligned}\mathcal{R} &= (\mathcal{R}_1 \cdot \mathcal{R}_2) \\ &= \{(A, C) \mid \exists B.\ (A, B) \in \mathcal{R}_1, (B, C) \in \mathcal{R}_2\}\end{aligned}$$

Clearly, $(A_1, A_3) \in \mathcal{R}$. To establish $\mathcal{R}$ as a type simulation, consider $(A, C) \in \mathcal{R}$ and the case $A = ?\{\phi_1\}.\,A'$. Since $(A, B) \in \mathcal{R}_1$ for some B, we get that $B = ?\{\phi_2\}.\,B'$. As a first case, assume $\models \phi_1$ and $\models \phi_2$ and $(A', B') \in \mathcal{R}_1$. Similarly, since $(B, C) \in \mathcal{R}_2$, we get that $C = ?\{\phi_3\}.\,C'$. And since $\models \phi_2$, we get that $\models \phi_3$ and $(B', C') \in \mathcal{R}_2$. Using the definition of $\mathcal{R}$, we get that $(A', C') \in \mathcal{R}$. Thus, we have $A = ?\{\phi_1\}.\,A'$ and $C = ?\{\phi_3\}.\,C'$ and and $\models \phi_1$ and $\models \phi_3$ and $(A', C') \in \mathcal{R}$. Therefore, $\mathcal{R}$ is a type simulation. A similar argument holds if $\models \neg\phi_1$. $\square$

B.2 Type Equality Algorithm

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

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

A	$::=$	$\oplus\{\ell : T\}_{\ell \in L} \mid \&\{\ell : T\}_{\ell \in I} \mid T \multimap T \mid T \otimes T \mid \mathbf{1}$
		$\mid\quad ?\{\phi\}.\,T \mid !\{\phi\}.\,T \mid \exists n.\,T \mid \forall n.\,T$
T	$::=$	$V\overline{[e]}$

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 $\mathcal{V}$; $\mathcal{C}$; $\Gamma \vdash A \equiv B$. Either A and B are 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. $\square$

$$\frac{\mathcal{V}\;;\;C\;;\;\Gamma \vdash T_l \equiv U_l \quad (\forall l \in L)}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash \oplus\{l : T_l\}_{l \in L} \equiv \oplus\{l : U_l\}_{l \in L}}\ \oplus$$

$$\frac{\mathcal{V}\;;\;C\;;\;\Gamma \vdash T_l \equiv U_l \quad (\forall l \in L)}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash \&\{l : T_l\}_{l \in L} \equiv \&\{l : U_l\}_{l \in L}}\ \&$$

$$\frac{\mathcal{V}\;;\;C\;;\;\Gamma \vdash T_1 \equiv U_1 \qquad \mathcal{V}\;;\;C\;;\;\Gamma \vdash T_2 \equiv U_2}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash T_1 \otimes T_2 \equiv U_1 \otimes U_2}\ \otimes$$

$$\frac{\mathcal{V}\;;\;C\;;\;\Gamma \vdash T_1 \equiv U_1 \qquad \mathcal{V}\;;\;C\;;\;\Gamma \vdash T_2 \equiv U_2}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash T_1 \multimap T_2 \equiv U_1 \multimap U_2}\ \multimap$$

$$\frac{}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash \mathbf{1} \equiv \mathbf{1}}\ \mathbf{1}$$

$$\frac{\mathcal{V}\;;\;C \vDash \phi \leftrightarrow \psi \qquad \mathcal{V}\;;\;C \wedge \phi\;;\;\Gamma \vdash T \equiv U}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash ?\{\phi\}.\,T \equiv ?\{\psi\}.\,U}\ ?$$

$$\frac{\mathcal{V}\;;\;C \vDash \phi \leftrightarrow \psi \qquad \mathcal{V}\;;\;C \wedge \phi\;;\;\Gamma \vdash T \equiv U}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash !\{\phi\}.\,T \equiv !\{\psi\}.\,U}\ !$$

$$\frac{\mathcal{V},k\;;\;C\;;\;\Gamma \vdash T[k/m] \equiv U[k/n]}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash \exists m.\,T \equiv \exists n.\,U}\ \exists$$

$$\frac{\mathcal{V},k\;;\;C\;;\;\Gamma \vdash T[k/m] \equiv U[k/n]}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash \forall m.\,T \equiv \forall n.\,U}\ \forall$$

$$\frac{\mathcal{V}\;;\;C \vDash \bot}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash T \equiv U}\ \bot$$

$$\frac{\forall \mathcal{V}'.\,C' \Rightarrow V_1\overline{[E_1]} \equiv V_2\overline{[E_2]} \in \Gamma \qquad \mathcal{V}\;;\;C \vDash \exists \mathcal{V}'.\,C' \wedge \overline{[E_1]} = \overline{[e_1]} \wedge \overline{[E_2]} = \overline{[e_2]}}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash V_1\overline{[e_1]} \equiv V_2\overline{[e_2]}}\ \text{def}$$

$$\frac{\begin{array}{c} V_1\overline{[v_1]} = A \in \Sigma \qquad V_2\overline{[v_2]} = B \in \Sigma \\ \gamma = \forall \mathcal{V}.\,C \Rightarrow V_1\overline{[e_1]} \equiv V_2\overline{[e_2]} \\ \mathcal{V}\;;\;C\;;\;\Gamma,\gamma \vdash A\overline{[e_1/v_1]} \equiv B\overline{[e_2/v_2]} \end{array}}{\mathcal{V}\;;\;C\;;\;\Gamma \vdash V_1\overline{[e_1]} \equiv V_2\overline{[e_2]}}\ \text{expd}$$

Figure 9. Algorithmic Rules for Type Equality

B.3 Soundness of Type Equality

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

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

Lemma 5. *Suppose $\forall \mathcal{V}'.C' \Rightarrow V_1\overline{[e_1']} \equiv_{\mathcal{R}} V_2\overline{[e_2']}$ holds. And suppose, $\mathcal{V}\;;\;C \vDash \exists \mathcal{V}'.C' \wedge \overline{e_1'} = \overline{e_1} \wedge \overline{e_2'} = \overline{e_2}$ for some $\mathcal{V}, C, \overline{[e_1]}, \overline{[e_2]}$. Then, $\forall \mathcal{V}.C \Rightarrow V_1\overline{[e_1]} \equiv_{\mathcal{R}} V_2\overline{[e_2]}$ holds.*

Proof. To prove $\forall \mathcal{V}.C \Rightarrow V_1\overline{[e_1]} \equiv_{\mathcal{R}} V_2\overline{[e_2]}$, it is sufficient to show that $V_1\overline{[e_1[\sigma]]} \equiv_{\mathcal{R}} V_2\overline{[e_2[\sigma]]}$ for any substitution $\sigma : \mathcal{V}$ such that $\cdot \vDash C[\sigma]$. Applying this substitution to $\mathcal{V}\;;\;C \vDash \exists \mathcal{V}'.C' \wedge \overline{e_1'} = \overline{e_1} \wedge \overline{e_2'} = \overline{e_2}$, we get that $\exists \mathcal{V}'.C' \wedge \overline{e_1'} = \overline{e_1[\sigma]} \wedge \overline{e_2'} = \overline{e_2[\sigma]}$ since $C[\sigma]$ is true. Thus, there exists $\sigma' : \mathcal{V}'$ such that $\cdot \vDash C'[\sigma']$ holds, and $\overline{e_1'[\sigma']} = \overline{e_1[\sigma]}$ and $\overline{e_2'[\sigma']} = \overline{e_2[\sigma]}$. And since $\forall \mathcal{V}'.C' \Rightarrow V_1\overline{[e_1']} \equiv_{\mathcal{R}} V_2\overline{[e_2']}$, we can use substitution σ' to get that $V_1\overline{[e_1'[\sigma']]} \equiv_{\mathcal{R}} V_2\overline{[e_2'[\sigma']]}$. This implies that $V_1\overline{[e_1[\sigma]]} \equiv_{\mathcal{R}} V_2\overline{[e_2[\sigma]]}$ since $\overline{e_1'[\sigma']} = \overline{e_1[\sigma]}$ and $\overline{e_2'[\sigma']} = \overline{e_2[\sigma]}$. □

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

Theorem 5. *If $\mathcal{V}\;;\;C\;;\;\cdot \vdash A \equiv B$, then $\forall \mathcal{V}.C \Rightarrow A \equiv B$.*

Proof. Let $\mathcal{S}$ be the set such that $\mathsf{conc}(\mathcal{V}\;;\;C\;;\;\cdot \vdash A \equiv B) = \mathcal{S}$. Define a relation $\mathcal{R}$ such that

$$\mathcal{R} = \{(A[\sigma], B[\sigma]) \mid (\forall \mathcal{V}.C \Rightarrow A \equiv B) \in \mathcal{S},$$
$$\forall \sigma : \mathcal{V}.\ \cdot \vDash C[\sigma]\}$$

We prove that $\mathcal{R}$ is a type simulation. Consider $(A[\sigma], B[\sigma]) \in \mathcal{R}$ where $(\mathcal{V}\;;\;C\;;\;A \equiv B) \in \mathcal{S}$ for some $\sigma : \mathcal{V}$ and $\cdot \vDash C[\sigma]$. We case analyze on the structure of $A[\sigma]$.

Consider the case where $A = \oplus\{l : A_l\}_{l \in L}$. Since A and B are both structural, $B = \oplus\{l : B_l\}_{l \in L}$. Since $(\forall \mathcal{V}.C \Rightarrow A \equiv B) \in \mathcal{S}$, by definition of $\mathcal{S}$, we get $(\forall \mathcal{V}.C \Rightarrow A_l \equiv B_l) \in \mathcal{S}$ for all $l \in L$. Then, by the definition of $\mathcal{R}$, we get that $(A_l[\sigma], B_l[\sigma]) \in \mathcal{R}$. Also, $A[\sigma] = \oplus\{l : A_l[\sigma]\}_{l \in L}$ and similarly, $B[\sigma] = \oplus\{l : B_l[\sigma]\}_{l \in L}$. Hence, $\mathcal{R}$ satisfies Definition 6 and is indeed a type simulation.

Next, consider the case where $A = ?\{\phi\}.A'$. Since A and B are both structural, $B = ?\{\psi\}.B'$. Since $(\forall \mathcal{V}.C \Rightarrow A \equiv B) \in \mathcal{S}$, we get that $\mathcal{V}\;;\;C \vDash \phi \leftrightarrow \psi$ and $(\forall \mathcal{V}.C \wedge \phi \Rightarrow A' \equiv B') \in \mathcal{S}$. Thus, for any substitution σ such that $\cdot \vDash C[\sigma] \wedge \phi[\sigma]$, we get that $(A'[\sigma], B'[\sigma]) \in \mathcal{R}$. And $A[\sigma] = ?\{\phi[\sigma]\}.A'[\sigma]$, and $B[\sigma] = ?\{\psi[\sigma]\}.B'[\sigma]$. Since $\cdot \vDash \phi[\sigma]$, we get that $\phi[\sigma] = \top$ and since $\mathcal{V}\;;\;C \vDash \phi \leftrightarrow \psi$, we get that $\psi[\sigma] = \top$. Thus, $\mathcal{R}$ is a type simulation.

Next, consider the case where $A = \exists m.A'$. Since A and B are both structural, $B = \exists n.B'$. Since $(\forall \mathcal{V}.C \Rightarrow A \equiv B) \in \mathcal{S}$, we get that $(\forall \mathcal{V},k.C \Rightarrow A'[k/m] \equiv B'[k/n]) \in \mathcal{S}$. Thus, for any substitution σ over $\mathcal{V},k$ such that $\cdot \vDash C[\sigma]$, we get that $(A'[k/m][\sigma], B'[k/n][\sigma]) \in \mathcal{R}$. Since σ is over $\mathcal{V},k$, we know that k is substituted over a ground term, and since C does not mention k, we know that k can be substituted with any ground term. Thus, $\forall i \in \mathbb{N}$, $(A'[i/k][k/m][\sigma], B'[i/k][k/n][\sigma]) \in \mathcal{R}$. Thus, $\mathcal{R}$ is a type simulation.

The only case where a conclusion is not added to $\mathcal{S}$ is the def rule. In this case, adding $(\forall \mathcal{V}.C \Rightarrow V_1\overline{[e_1]} \equiv V_2\overline{[e_2]})$ is redundant. Because Lemma 5 states that $V_1\overline{[e_1[\sigma]]} \equiv_{\mathcal{R}}$

$$\frac{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_l \equiv U_l) = \mathcal{S}_l \quad (\forall l \in L) \qquad \mathcal{S} = \bigcup_{l \in L} \mathcal{S}_l}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash \oplus\{l : T_l\}_{l \in L} \equiv \oplus\{l : U_l\}_{l \in L}) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow \oplus\{l : T_l\}_{l \in L} \equiv \oplus\{l : U_l\}_{l \in L})} \; \oplus$$

$$\frac{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_l \equiv U_l) = \mathcal{S}_l \quad (\forall l \in L) \qquad \mathcal{S} = \bigcup_{l \in L} \mathcal{S}_l}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash \&\{l : T_l\}_{l \in L} \equiv \&\{l : U_l\}_{l \in L}) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow \&\{l : T_l\}_{l \in L} \equiv \&\{l : U_l\}_{l \in L})} \; \&$$

$$\frac{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_1 \equiv U_1) = \mathcal{S}_1 \qquad \mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_2 \equiv U_2) = \mathcal{S}_2 \qquad \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_1 \otimes T_2 \equiv U_1 \otimes U_2) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow T_1 \otimes T_2 \equiv U_1 \otimes U_2)} \; \otimes$$

$$\frac{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_1 \equiv U_1) = \mathcal{S}_1 \qquad \mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_2 \equiv U_2) = \mathcal{S}_2 \qquad \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash T_1 \multimap T_2 \equiv U_1 \multimap U_2) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow T_1 \multimap T_2 \equiv U_1 \multimap U_2)} \; \multimap$$

$$\frac{}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash \mathbf{1} \equiv \mathbf{1}) = \{(\mathcal{V}\,;\,C\,;\,\mathbf{1} \equiv \mathbf{1})\}} \; \mathbf{1}$$

$$\frac{\mathcal{V}\,;\,C \vDash \phi \leftrightarrow \psi \qquad \mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C \wedge \phi\,;\,\Gamma \vdash T \equiv U) = \mathcal{S}}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash \,?\{\phi\}.\,T \equiv \,?\{\psi\}.\,U) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow \,?\{\phi\}.\,T \equiv \,?\{\psi\}.\,U)} \; ?$$

$$\frac{\mathcal{V}\,;\,C \vDash \phi \leftrightarrow \psi \qquad \mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C \wedge \phi\,;\,\Gamma \vdash T \equiv U) = \mathcal{S}}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash \,!\{\phi\}.\,T \equiv \,!\{\psi\}.\,U) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow \,!\{\phi\}.\,T \equiv \,!\{\psi\}.\,U)} \; !$$

$$\frac{\mathsf{conc}(\Sigma\,;\,\mathcal{V}, k\,;\,C\,;\,\Gamma \vdash T[k/m] \equiv U[k/n]) = \mathcal{S}}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash \exists m.\,T \equiv \exists n.\,U) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow \exists m.\,T \equiv \exists n.\,U)} \; \exists$$

$$\frac{\mathsf{conc}(\Sigma\,;\,\mathcal{V}, k\,;\,C\,;\,\Gamma \vdash T[k/m] \equiv U[k/n]) = \mathcal{S}}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash \forall m.\,T \equiv \forall n.\,U) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow \forall m.\,T \equiv \forall n.\,U)} \; \forall$$

$$\frac{\forall \mathcal{V}'.\,C' \Rightarrow V_1\overline{[E_1]} \equiv V_2\overline{[E_2]} \in \Gamma \qquad \forall \mathcal{V}.C \rightarrow \exists \mathcal{V}'.C' \wedge \overline{E_1} = \overline{e_1} \wedge \overline{E_2} = \overline{e_2}}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash V_1\overline{[e_1]} \equiv V_2\overline{[e_2]}) = \{\}} \; \mathsf{def}$$

$$\frac{\begin{array}{ccc} V_1\overline{[v_1]} = A \in \Sigma & V_2\overline{[v_2]} = B \in \Sigma & \gamma = \forall \mathcal{V}.\, C \Rightarrow V_1\overline{[e_1]} \equiv V_2\overline{[e_2]} \end{array} \quad \mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma, \gamma \vdash A\overline{[e_1/v_1]} \equiv B\overline{[e_2/v_2]}) = \mathcal{S}}{\mathsf{conc}(\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Gamma \vdash V_1\overline{[e_1]} \equiv V_2\overline{[e_2]}) = \mathcal{S}, (\forall \mathcal{V}.C \Rightarrow V_1\overline{[e_1]} \equiv V_2\overline{[e_2]})} \; \mathsf{exp}$$

Figure 10. Judgment to collect all the equality conclusions

$V_2\overline{[e_2[\sigma]]}$, this implies $(V_1\overline{[e_1[\sigma]]}, V_2\overline{[e_2[\sigma]]}) \in \mathcal{R}$. Thus, $\mathcal{R}$ is indeed a type simulation.

□

B.4 Undecidability of Type Equality

We prove the undecidability of type equality by exhibiting a reduction from an undecidable problem about two counter machines.

A two counter machine $\mathcal{M}$ is given a sequence of instructions $\iota_1, \iota_2, \ldots, \iota_m$ where each instruction is one of the following.

- "inc(c_j); goto k" (increment counter j by 1 and go to instruction k)
- "zero(c_j)? goto k : dec(c_j); goto l" (if the value of the counter j is 0, go to instruction k, else decrement the counter by 1 and go to instruction l)
- "halt" (stop computation)

A configuration C of the machine $\mathcal{M}$ 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 $C \mapsto C'$ if C' is the result of executing the i-th instruction on C. If $\iota_i = $ halt, then $C = (i, c_1, c_2)$ has no successor configuration.

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

The *halting problem* refers to determining whether the computation of a two counter machine $\mathcal{M}$ 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

28

two counter machine M to an instance of the type equality problem.

Consider the instruction ι_i. 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 $(\iota_i = \mathrm{inc}(c_1); \mathrm{goto}\ k)$: In this case, we define the type $T_i[c_1, c_2] = \oplus\{\mathrm{inc}_1 : T_k[c_1 + 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'_i[c_1, c_2] = \oplus\{\mathrm{inc}_1 : T'_k[c_1 + 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$.

- Case $(\iota_i = \mathrm{inc}(c_2); \mathrm{goto}\ k)$: In this case, we define the type $T_i[c_1, c_2] = \oplus\{\mathrm{inc}_2 : T_k[c_1, c_2 + 1]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'_i[c_1, c_2] = \oplus\{\mathrm{inc}_2 : T'_k[c_1, c_2 + 1]\}$ for all $c_1, c_2 \in \mathbb{N}$.

- Case $(\iota_i = \mathrm{zero}(c_1)?\ \mathrm{goto}\ k : \mathrm{dec}(c_j); \mathrm{goto}\ l)$: In this case, we define the type $T_i[c_1, c_2] = \oplus\{\mathrm{zero}_1 : ?\{c_1 = 0\}.\, T_k[c_1, c_2], \mathrm{dec}_1 : ?\{c_1 > 0\}.\, T_l[c_1 - 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'_i[c_1, c_2] = \oplus\{\mathrm{zero}_1 : ?\{c_1 = 0\}.\, T'_k[c_1, c_2], \mathrm{dec}_1 : ?\{c_1 > 0\}.\, T'_l[c_1 - 1, c_2]\}$.

- Case $(\iota_i = \mathrm{zero}(c_2)?\ \mathrm{goto}\ k : \mathrm{dec}(c_j); \mathrm{goto}\ l)$: In this case, we define the type $T_i[c_1, c_2] = \oplus\{\mathrm{zero}_2 : ?\{c_2 = 0\}.\, T_k[c_1, c_2], \mathrm{dec}_2 : ?\{c_2 > 0\}.\, T_l[c_1, c_2 - 1]\}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'_i[c_1, c_2] = \oplus\{\mathrm{zero}_2 : ?\{c_2 = 0\}.\, T'_k[c_1, c_2], \mathrm{dec}_2 : ?\{c_2 > 0\}.\, T'_l[c_1, c_2 - 1]\}$ for all $c_1, c_2 \in \mathbb{N}$

- Case $(\iota_i = \mathrm{halt})$: In this case, we define $T_i[c_1, c_2] = T_{inf}$ for all $c_1, c_2 \in \mathbb{N}$. We also define $T'_i[c_1, c_2] = T'_{\inf}$ 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 $T_1[m, n] \equiv T'_1[m, n]$. The two types only differ at the halting instruction. If and only if M halts, the first type sends infinitely many ℓ labels while the second type sends infintely many ℓ' 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.

$$\dfrac{}{(\cdot) \overset{0}{\vDash} (\cdot) :: (\cdot)} \; emp$$

$$\dfrac{\Delta \overset{E}{\vDash} \mathcal{S} :: \Delta' \qquad \Sigma \,;\, \Delta' \overset{E'}{\vDash} \mathcal{S}' :: \Delta''}{\Delta \overset{E+E'}{\vDash} (\mathcal{S}\,\mathcal{S}') :: \Delta''} \; compose$$

$$\dfrac{\cdot\,;\,\cdot\,;\,\Delta_1 \overset{p}{\vdash} P :: (x : A)}{\Delta, \Delta_1 \overset{p+w}{\vDash} (\mathrm{proc}(x, w, P)) :: (\Delta, (x : A))} \; proc$$

$$\dfrac{\cdot\,;\,\cdot\,;\,\Delta_1 \overset{p}{\vdash} P :: (x : A)}{\Delta, \Delta_1 \overset{p+w}{\vDash} (\mathrm{msg}(x, w, P)) :: (\Delta, (x : A))} \; msg$$

Figure 12. Configuration Typing Rules

$$\dfrac{\mathcal{V}\,;\,C \vDash \phi \qquad \mathcal{V}\,;\,C\,;\,\Delta \overset{g}{\vdash_i} P :: (x : A)}{\mathcal{V}\,;\,C\,;\,\Delta \overset{g}{\vdash_i} P :: (x : ?\{\phi\}.A)} \; ?R$$

$$\dfrac{\mathcal{V}\,;\,C \wedge \phi\,;\,\Delta, (x : A) \overset{g}{\vdash_i} Q :: (z : C)}{\mathcal{V}\,;\,C\,;\,\Delta, (x : ?\{\phi\}.A) \overset{g}{\vdash_i} Q :: (z : C)} \; ?L$$

$$\dfrac{\mathcal{V}\,;\,C \wedge \phi\,;\,\Delta \overset{g}{\vdash_i} P :: (x : A)}{\mathcal{V}\,;\,C\,;\,\Delta \overset{g}{\vdash_i} P :: (x : !\{\phi\}.A)} \; !R$$

$$\dfrac{\mathcal{V}\,;\,C \vDash \phi \qquad C\,;\,\Delta, (x : A) \overset{g}{\vdash_i} Q :: (z : C)}{\mathcal{V}\,;\,C\,;\,\Delta, (x : !\{\phi\}.A) \overset{g}{\vdash_i} Q :: (z : C)} \; !L$$

$$\dfrac{\mathcal{V}\,;\,C \vDash q \geq r \qquad \mathcal{V}\,;\,C\,;\,\Delta \overset{g-r}{\vdash_i} P :: (x : A)}{\mathcal{V}\,;\,C\,;\,\Delta \overset{g}{\vdash_i} P :: (x : \triangleright^r A)} \; \triangleright R$$

$$\dfrac{\mathcal{V}\,;\,C\,;\,\Delta, (x : A) \overset{g+r}{\vdash_i} Q :: (z : C)}{\mathcal{V}\,;\,C\,;\,\Delta, (x : \triangleright^r A) \overset{g}{\vdash_i} Q :: (z : C)} \; \triangleright L$$

$$\dfrac{\mathcal{V}\,;\,C\,;\,\Delta \overset{g+r}{\vdash_i} P :: (x : A)}{\mathcal{V}\,;\,C\,;\,\Delta \overset{g}{\vdash_i} P :: (x : \triangleleft^r A)} \; \triangleleft R$$

$$\dfrac{\mathcal{V}\,;\,C \vDash q \geq r \qquad \mathcal{V}\,;\,C\,;\,\Delta, (x : A) \overset{g-r}{\vdash_i} Q :: (z : C)}{\mathcal{V}\,;\,C\,;\,\Delta, (x : \triangleleft^r A) \overset{g}{\vdash_i} Q :: (z : C)} \; \triangleleft L$$

Figure 11. Implicit Typing Rules

$$\frac{\mathcal{V}\,;\,C \vDash q = 0 \qquad \mathcal{V}\,;\,C\,;\,\cdot \vdash A \equiv B}{\mathcal{V}\,;\,C\,;\,y:A \vdash^g x \leftarrow y :: (x:B)}\ \text{id}$$

$$\frac{\begin{array}{c} q = p[\overline{e_c}/\overline{v}] \qquad \mathcal{V}\,;\,C \vDash r \geq q \qquad X\overline{[v]}:\overline{y_i:A_i} \vdash^p x':A \in \Sigma \\ \mathcal{V}\,;\,C\,;\,\cdot \vdash \Delta_1 \equiv_\alpha \overline{y_i:A_i[e_c/v]} \\ \Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Delta_2,(x:A[\overline{e_c}/\overline{v}]) \vdash^{r-q} Q_x :: (z:C) \end{array}}{\Sigma\,;\,\mathcal{V}\,;\,C\,;\,\Delta_1,\Delta_2 \vdash^r (x \leftarrow X\overline{[e_c]} \leftarrow y\,;\,Q_x) :: (z:C)}\ \text{spawn}$$

$$\frac{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g P :: (x:A_k) \quad (k \in L)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g (x.k\,;\,P) :: (x:\oplus\{\ell:A_\ell\}_{\ell \in L})}\ \oplus R_k \qquad \frac{\mathcal{V}\,;\,C\,;\,\Delta,(x:A_\ell) \vdash^g Q_\ell :: (z:C) \quad (\forall \ell \in L)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:\oplus\{\ell:A_\ell\}_{\ell \in L}) \vdash^g \text{case } x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z:C)}\ \oplus L$$

$$\frac{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g P_\ell :: (x:A_\ell) \quad (\forall \ell \in L)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g \text{case } x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x:\&\{\ell:A_\ell\}_{\ell \in L})}\ \&R \qquad \frac{\mathcal{V}\,;\,C\,;\,\Delta,(x:A_k) \vdash^g Q :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:\&\{\ell:A_\ell\}_{\ell \in L}) \vdash^g x.k\,;\,Q :: (z:C)}\ \&L_k$$

$$\frac{\mathcal{V}\,;\,C\,;\,\Delta,(y:A) \vdash^g P_y :: (x:B)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g (y \leftarrow \text{recv } x\,;\,P_y) :: (x:A \multimap B)}\ \multimap R \qquad \frac{\mathcal{V}\,;\,C\,;\,\cdot \vdash A_1 \equiv A_2 \qquad \mathcal{V}\,;\,C\,;\,\Delta,(x:B) \vdash^g Q :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(w:A_1),(x:A_2 \multimap B) \vdash^g (\text{send } x\ w\,;\,Q) :: (z:C)}\ \multimap L$$

$$\frac{\mathcal{V}\,;\,C\,;\,\cdot \vdash A_1 \equiv A_2 \qquad \mathcal{V}\,;\,C\,;\,\Delta \vdash^g P :: (x:B)}{\mathcal{V}\,;\,C\,;\,(w:A_1),\Delta \vdash^g \text{send } x\ w\,;\,P :: (x:A_2 \otimes B)}\ \otimes R \qquad \frac{\mathcal{V}\,;\,C\,;\,\Delta,(y:A),(x:B) \vdash^g Q_y :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:A \otimes B) \vdash^g y \leftarrow \text{recv } x\,;\,Q_y :: (z:C)}\ \otimes L$$

$$\frac{\mathcal{V}\,;\,C \vDash q = 0}{\mathcal{V}\,;\,C\,;\,\cdot \vdash^g \text{close } x :: (x:\mathbf{1})}\ \mathbf{1}R \qquad \frac{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g Q :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:\mathbf{1}) \vdash^g \text{wait } x\,;\,Q :: (z:C)}\ \mathbf{1}L$$

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

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

$$\frac{\mathcal{V}\,;\,C \vDash q \geq c \qquad \mathcal{V}\,;\,C\,;\,\Delta \vdash^{g-c} P :: (x:A)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g \text{work}\,\{c\}\,;\,P :: (x:A)}\ \text{work}$$

$$\frac{\mathcal{V}\,;\,C \vDash q \geq r_1 = r_2 \qquad \mathcal{V}\,;\,C\,;\,\Delta \vdash^{g-r_1} P :: (x:A)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g \text{pay } x\ \{r_1\}\,;\,P :: (x:\triangleright^{r_2} A)}\ \triangleright R \qquad \frac{\mathcal{V}\,;\,C \vDash r_1 = r_2 \qquad \mathcal{V}\,;\,C\,;\,\Delta,(x:A) \vdash^{g+r_1} Q :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:\triangleright^{r_2} A) \vdash^g \text{get } x\ \{r_1\}\,;\,Q :: (z:C)}\ \triangleright L$$

$$\frac{\mathcal{V}\,;\,C \vDash r_1 = r_2 \qquad \mathcal{V}\,;\,C\,;\,\Delta \vdash^{g+r_1} P :: (x:A)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g \text{get } x\ \{r_1\}\,;\,P :: (x:\triangleleft^{r_2} A)}\ \triangleleft R \qquad \frac{\mathcal{V}\,;\,C \vDash q \geq r_1 = r_2 \qquad \mathcal{V}\,;\,C\,;\,\Delta,(x:A) \vdash^{g-r_1} Q :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:\triangleleft^{r_2} A) \vdash^g \text{pay } x\ \{r_1\}\,;\,Q :: (z:C)}\ \triangleleft L$$

$$\frac{\mathcal{V}\,;\,C \vDash \phi \quad \mathcal{V}\,;\,C \vDash \phi' \quad \mathcal{V}\,;\,C\,;\,\Delta \vdash^g P :: (x:A)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g \text{assert } x\ \{\phi\}\,;\,P :: (x:\,?\{\phi'\}.\,A)}\ ?R \qquad \frac{\mathcal{V}\,;\,C \wedge \phi' \vDash \phi \quad \mathcal{V}\,;\,C \wedge \phi\,;\,\Delta,(x:A) \vdash^g Q :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:\,?\{\phi'\}.\,A) \vdash^g \text{assume } x\ \{\phi\}\,;\,Q :: (z:C)}\ ?L$$

$$\frac{\mathcal{V}\,;\,C \wedge \phi' \vDash \phi \quad \mathcal{V}\,;\,\Gamma \wedge \phi\,;\,\Delta \vdash^g P :: (x:A)}{\mathcal{V}\,;\,C\,;\,\Delta \vdash^g \text{assume } x\ \{\phi\}\,;\,P :: (x:\,!\{\phi'\}.\,A)}\ !R \qquad \frac{\mathcal{V}\,;\,C \vDash \phi \quad \mathcal{V}\,;\,C \vDash \phi' \quad \mathcal{V}\,;\,C\,;\,\Delta,(x:A) \vdash^g Q :: (z:C)}{\mathcal{V}\,;\,C\,;\,\Delta,(x:\,!\{\phi'\}.\,A) \vdash^g \text{assert } x\ \{\phi\}\,;\,Q :: (z:C)}\ !L$$

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

Figure 13. Explicit Typing Rules

(id^+C) $\mathrm{msg}(d, w, M), \mathrm{proc}(c, w', s, c \leftarrow d) \;\mapsto\; \mathrm{msg}(c, w + w', M[c/d])$

(id^-C) $\mathrm{proc}(c, w, s, c \leftarrow d), \mathrm{msg}(e, w', M(c)) \;\mapsto\; \mathrm{msg}(e, w + w', M(c)[d/c])$

$(\mathrm{def}C)$ $\mathrm{proc}(c, w, x \leftarrow f\,\overline{[e]} \leftarrow \overline{y}\;;\; Q) \;\mapsto\; \mathrm{proc}(a, 0, P_f[a/x, \overline{y}/\Delta_f, \overline{e}/\overline{n}]),\; \mathrm{proc}(c, w, Q[a/x])$ $(a \text{ fresh})$

$(\oplus S)$ $\mathrm{proc}(c, w, c.k\;;\; P) \;\mapsto\; \mathrm{proc}(c', w, P[c'/c]), \mathrm{msg}(c, 0, c.k\;;\; c \leftarrow c')$ $(c' \text{ fresh})$

$(\oplus C)$ $\mathrm{msg}(c, w', c.k\;;\; c \leftarrow c'), \mathrm{proc}(d, w, \mathrm{case}\; c\; (\ell \Rightarrow Q_\ell)_{\ell \in L}) \;\mapsto\; \mathrm{proc}(d, w + w', Q_k[c'/c])$

$(\&S)$ $\mathrm{proc}(d, w, c.k\;;\; Q) \;\mapsto\; \mathrm{msg}(c', 0, c.k\;;\; c' \leftarrow c), \mathrm{proc}(d, w, Q[c'/c])$ $(c' \text{ fresh})$

$(\&C)$ $\mathrm{proc}(c, w, \mathrm{case}\; c\; (\ell \Rightarrow Q_\ell)_{\ell \in L}), \mathrm{msg}(c', w', c.k\;;\; c' \leftarrow c) \;\mapsto\; \mathrm{proc}(c', w + w', Q_k[c'/c])$

$(\mathbf{1}S)$ $\mathrm{proc}(c, w, \mathrm{close}\; c) \;\mapsto\; \mathrm{msg}(c, w, \mathrm{close}\; c)$

$(\mathbf{1}C)$ $\mathrm{msg}(c, w', \mathrm{close}\; c), \mathrm{proc}(d, w, \mathrm{wait}\; c\;;\; Q) \;\mapsto\; \mathrm{proc}(d, w + w', Q)$

$(\otimes S)$ $\mathrm{proc}(c, w, \mathrm{send}\; c\; d\;;\; P) \;\mapsto\; \mathrm{proc}(c', w, P[c'/c]), \mathrm{msg}(c, 0, \mathrm{send}\; c\; d\;;\; c \leftarrow c')$ $(c' \text{ fresh})$

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

$(\multimap S)$ $\mathrm{proc}(e, w, \mathrm{send}\; c\; d\;;\; Q) \;\mapsto\; \mathrm{msg}(c', 0, \mathrm{send}\; c\; d\;;\; c' \leftarrow c), \mathrm{proc}(e, w, Q[c'/c])$ $(c' \text{ fresh})$

$(\multimap C)$ $\mathrm{proc}(c, w, x \leftarrow \mathrm{recv}\; x\;;\; P), \mathrm{msg}(c', w', \mathrm{send}\; c\; d\;;\; c' \leftarrow c) \;\mapsto\; \mathrm{proc}(c', w + w', P[c', d/c, x])$

$(\exists S)$ $\mathrm{proc}(c, w, \mathrm{send}\; c\; \{e\}\;;\; P) \;\mapsto\; \mathrm{proc}(c', w, P[c'/c]),\; \mathrm{msg}(c, 0, \mathrm{send}\; c\; \{e\}\;;\; c \leftarrow c')$ $(c' \text{ fresh})$

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

$(\forall S)$ $\mathrm{proc}(d, w, \mathrm{send}\; c\; \{e\}\;;\; P) \;\mapsto\; \mathrm{msg}(c', 0, \mathrm{send}\; c\; \{e\}\;;\; c' \leftarrow c),\; \mathrm{proc}(d, w, P[c'/c])$ $(c' \text{ fresh})$

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

$(?S)$ $\mathrm{proc}(c, w, \mathrm{assert}\; c\; \{\phi\}\;;\; P) \;\mapsto\; \mathrm{proc}(c', w, P[c'/c]),\; \mathrm{msg}(c, 0, \mathrm{assert}\; c\; \{\phi\}\;;\; c \leftarrow c')$ $(c' \text{ fresh})$

$(?C)$ $\mathrm{msg}(c, w', \mathrm{assert}\; c\; \{\phi_1\}\;;\; c \leftarrow c'),\; \mathrm{proc}(d, w, \mathrm{assume}\; c\; \{\phi_2\}\;;\; Q) \;\mapsto\; \mathrm{proc}(d, w + w', Q[c'/c])$

$(!S)$ $\mathrm{proc}(d, w, \mathrm{assert}\; c\; \{\phi\}\;;\; P) \;\mapsto\; \mathrm{msg}(c', 0, \mathrm{assert}\; c\; \{\phi\}\;;\; c' \leftarrow c),\; \mathrm{proc}(d, w, P[c'/c])$ $(c' \text{ fresh})$

$(!C)$ $\mathrm{proc}(d, w, \mathrm{assume}\; c\; \{\phi_1\}\;;\; Q),\; \mathrm{msg}(c', w', \mathrm{assert}\; c\; \{\phi_2\}\;;\; c' \leftarrow c) \;\mapsto\; \mathrm{proc}(d, w + w', Q[c'/c])$

$(\triangleright S)$ $\mathrm{proc}(c, w, \mathrm{pay}\; c\; \{r\}\;;\; P) \;\mapsto\; \mathrm{proc}(c', w, P[c'/c]),\; \mathrm{msg}(c, 0, \mathrm{pay}\; c\; \{r\}\;;\; c \leftarrow c')$ $(c' \text{ fresh})$

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

$(\triangleleft S)$ $\mathrm{proc}(d, w, \mathrm{pay}\; c\; \{r\}\;;\; P) \;\mapsto\; \mathrm{msg}(c', 0, \mathrm{pay}\; c\; \{r\}\;;\; c' \leftarrow c),\; \mathrm{proc}(d, w, P[c'/c])$ $(c' \text{ fresh})$

$(\triangleleft C)$ $\mathrm{proc}(c, w, \mathrm{get}\; c\; \{r\}\;;\; Q),\; \mathrm{msg}(c', w', \mathrm{pay}\; c\; \{r\}\;;\; c' \leftarrow c) \;\mapsto\; \mathrm{proc}(c', w + w', Q[c'/c])$

(work) $\mathrm{proc}(c, w, \mathrm{work}\; \{r\}\;;\; P) \;\mapsto\; \mathrm{proc}(c, w + r, P)$

Figure 14. Basic and Refined Operational Semantics

D Forcing Calculus

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

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

Here, S denotes a structural type. Next, we define the rules for eagerly applying the $!R, ?L, \triangleleft R, \triangleright L$ rules.

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

$$
\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \vdash^{g+r} P :: (x : A^-)}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \vdash^g P :: (x : \triangleleft^r A^-)} \;\; \triangleleft R
$$

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

$$
\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \cdot (x : A^+) \vdash^{g+r} P :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \cdot (x : \triangleright^r A^+) \vdash^g P :: (z : C^+)} \;\; \triangleright L
$$

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

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

The invertible rules are applied until we reach a stable sequent defined using the normal form $\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash^g P :: (x : A^+)$. 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,

$$
\frac{\mathcal{V} \; ; \; C \; ; \; (y : B^-) \; ; \; \cdot \vdash^g x \leftarrow y :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; (y : B^-) \; ; \; \cdot \vdash^g x \leftarrow y :: (x : A^+)} \;\; \text{id}{-}F_R
$$

then force left.

$$
\frac{\mathcal{V} \; ; \; C \; ; \; [y : B^-] \; ; \; \cdot \vdash^g x \leftarrow y :: (x : S)}{\mathcal{V} \; ; \; C \; ; \; (y : B^-) \; ; \; \cdot \vdash^g x \leftarrow y :: [x : S]} \;\; \text{id}{-}F_L
$$

While forced, we apply the lazy rules $?R, !L, \triangleright R, \triangleleft L$.

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

$$
\frac{\mathcal{V} \; ; \; C \vDash q \geq r \qquad \mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash^{g-r} P :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash^g P :: [x : \triangleright^r A^+]} \;\; \triangleright R
$$

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

$$
\frac{\mathcal{V} \; ; \; C \vDash q \geq r \qquad \mathcal{V} \; ; \; C \; ; \; \Delta^-, [x : A^-] \; ; \; \cdot \vdash^{g-r} P :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [x : \triangleleft^r A^-] \; ; \; \cdot \vdash^g P :: (z : C^+)} \;\; \triangleleft L
$$

Finally, the forward is typechecked after the forcing.

$$
\frac{\mathcal{V} \; ; \; C \vDash q = 0}{\mathcal{V} \; ; \; C \; ; \; [y : S] \; ; \; \cdot \vdash^g x \leftarrow y :: (x : S)} \;\; \text{id}
$$

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 spawn$-F_n$ and spawn$-F_i$ 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.

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

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

$$
\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash^g \text{case } x \; (\ell \Rightarrow P_\ell)_{\ell \in L} :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \vdash^g \text{case } x \; (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : A^+)} \;\; \& F_R
$$

$$
\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [x : A^-] \; ; \; \cdot \vdash^g x.k \; ; \; Q :: (z : C)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (x : A^-) \; ; \; \cdot \vdash^g x.k \; ; \; Q :: (z : C^+)} \;\; \& F_L
$$

For the tensor and lolli rules, we force both the channel involved in the send while sending, and while receiving, we

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [y_n : A^-] \; ; \; \cdot \Vdash^g (x \leftarrow \mathcal{X}\overline{[e_c]} \leftarrow y_1 \ldots y_n \; ; \; Q_x) :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (y_n : A^-) \; ; \; \cdot \Vdash^g (x \leftarrow \mathcal{X}\overline{[e_c]} \leftarrow y_1 \ldots y_n \; ; \; Q_x) :: (z : C^+)} \; \text{spawn}-F_n$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [y_i : A^-], (y_{i+1} : S) \; ; \; \cdot \Vdash^g (x \leftarrow \mathcal{X}\overline{[e_c]} \leftarrow y_1 \ldots y_n \; ; \; Q_x) :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (y_i : A^-), [y_{i+1} : S] \; ; \; \cdot \Vdash^g (x \leftarrow \mathcal{X}\overline{[e_c]} \leftarrow y_1 \ldots y_n \; ; \; Q_x) :: (z : C^+)} \; \text{spawn}-F_i$$

$$\frac{\begin{array}{c} q = p[\overline{e_c}/\overline{v}] \quad\quad \mathcal{V} \; ; \; C \vDash r \geq q \quad\quad \mathcal{X}\overline{[v]} : \overline{y_i' : A_i} \Vdash^p x' : A \in \Sigma \\ \mathcal{V} \; ; \; C \vDash \overline{y_i' : B_i} \equiv_\alpha \Delta_1^-, y_1 : S \quad\quad \Sigma \; ; \; \mathcal{V} \; ; \; C \; ; \; \Delta_2^- \; ; \; (x : A[\overline{e_c}/\overline{v}]) \Vdash^{r-q} Q_x :: (z : C) \end{array}}{\Sigma \; ; \; \mathcal{V} \; ; \; C \; ; \; \Delta_1^-, [y_1 : S], \Delta_2^- \; ; \; \cdot \Vdash^r (x \leftarrow \mathcal{X}\overline{[e_c]} \leftarrow y_1 \ldots y_n \; ; \; Q_x) :: (z : C^+)} \; \text{spawn}$$

Figure 15. Spawn rules in forcing calculus

place the new channel in the ordered context.

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [w : A^-] \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; P :: (x : B^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (w : A^-) \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; P :: (x : B^+)} \; \otimes F_R^1$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (w : S) \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; P :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [w : S] \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; P :: (x : A^+)} \; \otimes F_R^2$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [x : A^-] \; ; \; \cdot \Vdash^g y \leftarrow \text{recv } x \; ; \; Q_y :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (x : A^-) \; ; \; \cdot \Vdash^g y \leftarrow \text{recv } x \; ; \; Q_y :: (z : C^+)} \; \otimes F_L$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \Vdash^g y \leftarrow \text{recv } x \; ; \; P_y :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \Vdash^g y \leftarrow \text{recv } x \; ; \; P_y :: (x : A^+)} \; \multimap F_R$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [w : A^-] \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; Q :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (w : A^-) \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; Q :: (z : C^+)} \; \multimap F_L^1$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (w : S), [x : A^-] \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; Q :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [w : S], (x : A^-) \; ; \; \cdot \Vdash^g \text{send } x \, w \; ; \; Q :: (z : C^+)} \; \multimap F_L^2$$

The rules for **1** are analogous.

$$\frac{\mathcal{V} \; ; \; C \; ; \; \cdot \; ; \; \cdot \Vdash^g \text{close } x :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \cdot \; ; \; \cdot \Vdash^g \text{close } x :: (x : A^+)} \; 1F_R$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [x : A^-] \; ; \; \cdot \Vdash^g \text{wait } x \; ; \; Q :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (x : A^-) \; ; \; \cdot \Vdash^g \text{wait } x \; ; \; Q :: (z : C^+)} \; 1F_L$$

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

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \Vdash^g \text{send } x \, \{t\} \; ; \; P :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \Vdash^g \text{send } x \, \{t\} \; ; \; P :: (x : A^+)} \; \exists F_R$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [x : A^-] \; ; \; \cdot \Vdash^g \{k\} \leftarrow \text{recv } x \; ; \; Q :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (x : A^-) \; ; \; \cdot \Vdash^g \{k\} \leftarrow \text{recv } x \; ; \; Q :: (z : C^+)} \; \exists F_L$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \Vdash^g \{k\} \leftarrow \text{recv } x \; ; \; P :: [x : A^+]}{\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \cdot \Vdash^g \{k\} \leftarrow \text{recv } x \; ; \; P :: (x : A^+)} \; \forall F_R$$

$$\frac{\mathcal{V} \; ; \; C \; ; \; \Delta^-, [x : A^-] \; ; \; \cdot \Vdash^g \text{send } x \, \{t\} \; ; \; Q :: (z : C^+)}{\mathcal{V} \; ; \; C \; ; \; \Delta^-, (x : A^-) \; ; \; \cdot \Vdash^g \text{send } x \, \{t\} \; ; \; Q :: (z : C^+)} \; \forall F_L$$

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

Lemma 6 (Invertibility). *The rules !R, ?L, ◃R, ▹L and move are invertible.*

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

$$\begin{array}{llllll} [!\{\phi\}. A]_R^! & = & \phi \wedge [A]_R^! & \quad [A^+]_R^! & = & \top \\ [!\{\phi\}. A]_R^\tau & = & [A]_R^\tau & \quad [A^+]_R^\tau & = & A^+ \\ [?\{\phi\}. A]_L^? & = & \phi \wedge [A]_L^? & \quad [A^-]_L^? & = & A^- \\ [?\{\phi\}. A]_L^\tau & = & [A]_L^\tau & \quad [A^-]_L^\tau & = & A^- \end{array}$$

Lemma 7 (Move Left). *If $\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \cdot (x : A) \Vdash^g P :: (z : C^+)$, then $\mathcal{V} \; ; \; C \wedge [A]_L^? \; ; \; \Delta^-, (x : [A]_L^\tau) \; ; \; \Omega \Vdash^g P :: (z : C^+)$.*

Proof. If $[A]_L^\tau = S$ and $[A]_L^? = \phi$, then $\mathcal{V} \; ; \; C \wedge \phi \; ; \; \Delta^-, (x : S) \; ; \; \Omega \Vdash^g P :: (z : C^+)$ by inverting !L or move. □

Lemma 8 (Stable Sequent). *If $\mathcal{V} \; ; \; C \; ; \; \Delta_1^- \; ; \; \Delta_2 \Vdash^g P :: (x : A)$, then $\mathcal{V} \; ; \; C \wedge [\Delta_2]_L^? \wedge [A]_R^! \; ; \; \Delta_1^-, [\Delta_2]_L^\tau \; ; \; \cdot \Vdash^g P :: (x : [A]_R^\tau)$.*

Proof. Successive application of Lemma 7 after inverting ?R. □

Lemma 9 (Lazy ?R). *If $\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \Vdash^g P :: (x : A^+)$ and $\mathcal{V} \; ; \; C \vDash \phi$, then $\mathcal{V} \; ; \; C \; ; \; \Delta^- \; ; \; \Omega \Vdash^g P :: (x : ?\{\phi\}. A^+)$.*

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 idR 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 idR, then id. All the spawn rules still apply in either case. The same holds

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

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

$$\frac{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g P_\ell :: (x : A_\ell)}{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g \text{case } x \,(\ell \Rightarrow P_\ell)_{\ell \in L} :: [x : \&\{\ell : A_\ell\}_{\ell \in L}]} \;\&R \qquad \frac{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; (x : A_k) \vdash^g Q :: (z : C^+)}{\mathcal{V} \;;\; C \;;\; \Delta^-, [x : \&\{\ell : A_\ell\}_{\ell \in L}] \vdash^g x.k \;;\; Q :: (z : C^+)} \;\&L_k$$

$$\frac{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g P :: (x : B)}{\mathcal{V} \;;\; C \;;\; \Delta^-, (w : S) \;;\; \cdot \vdash^g \text{send } x \, w \;;\; P :: [x : S \otimes B]} \;\otimes R \qquad \frac{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; (x : B) \cdot (y : S) \vdash^g Q_y :: (z : C^+)}{\mathcal{V} \;;\; C \;;\; \Delta^-, [x : S \otimes B] \;;\; \cdot \vdash^g y \leftarrow \text{recv } x \;;\; Q_y :: (z : C^+)} \;\otimes L$$

$$\frac{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; (y : S) \vdash^g P_y :: (x : B)}{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g y \leftarrow \text{recv } x \;;\; P_y :: [x : S \multimap B]} \;\multimap R$$

$$\frac{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; (x : B) \vdash^g Q :: (z : C^+)}{\mathcal{V} \;;\; C \;;\; \Delta^-, (w : S), [x : S \multimap B] \;;\; \cdot \vdash^g \text{send } x \, w \;;\; Q :: (z : C^+)} \;\multimap L$$

$$\frac{\mathcal{V} \;;\; C \vdash e : \text{nat} \quad \mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g P :: (x : A[e/n])}{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g \text{send } x \, \{e\} \;;\; P :: [x : \exists n. A]} \;\exists R \qquad \frac{\mathcal{V}, n \;;\; C \;;\; \Delta^- \;;\; (x : A) \vdash^g Q_n :: (z : C^+)}{\mathcal{V} \;;\; C \;;\; \Delta^-, [x : \exists n. A] \;;\; \cdot \vdash^g \{n\} \leftarrow \text{recv } x \;;\; Q_n :: (z : C^+)} \;\exists L$$

$$\frac{\mathcal{V}, n \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g P_n :: (x : A)}{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g \{n\} \leftarrow \text{recv } x \;;\; P_k :: [x : \forall n. A]} \;\forall R \qquad \frac{\mathcal{V} \;;\; C \vdash e : \text{nat} \quad \mathcal{V} \;;\; C \;;\; \Delta \;;\; (x : A[e/n]) \vdash^g Q :: (z : C^+)}{\mathcal{V} \;;\; C \;;\; \Delta^-, [x : \forall n. A] \;;\; \cdot \vdash^g \text{send } x \, \{e\} \;;\; Q :: (z : C^+)} \;\forall L$$

$$\frac{\mathcal{V} \;;\; C \vDash q = 0}{\mathcal{V} \;;\; C \;;\; \cdot \;;\; \cdot \vdash^g \text{close } x :: [x : 1]} \;1R \qquad \frac{\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \cdot \vdash^g Q :: (z : C^+)}{\mathcal{V} \;;\; C \;;\; \Delta^-, [x : 1] \;;\; \cdot \vdash^g \text{wait } x \;;\; Q :: (z : C^+)} \;1L$$

Figure 16. Structural rules in forcing calculus

for all the forcing rules. Finally, the right structural rules do not apply in either case, and the left rules are allowed in either case. □

Lemma 10 (Lazy !L). *If* $\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \Omega \cdot (x : A^-) \vdash^g P :: (z : C^+)$ *and* $\mathcal{V} \;;\; C \vDash \phi$, *then* $\mathcal{V} \;;\; C \;;\; \Delta^- \;;\; \Omega \cdot (x : !\{\phi\}. A^-) \vdash^g P :: (z : C^+)$.

Theorem 6 (Soundness of QRecon). *If* $\mathcal{V} \;;\; C \;;\; \cdot \;;\; \Delta \vdash^g P :: (x : A)$, *then* $\mathcal{V} \;;\; C \;;\; \Delta \;_i\vdash^g P :: (x : A)$.

Proof. Soundness is trivial, since every rule in forcing calculus is valid in the implicit type system. □

Theorem 7 (Completeness of QRecon). *If* $\mathcal{V} \;;\; C \;;\; \Delta \;_i\vdash^g P :: (x : A)$, *then* $\mathcal{V} \;;\; C \;;\; \cdot \;;\; \Delta \vdash^g P :: (x : A)$.

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 $(\oplus R_k)$: $P = x.k \;;\; P$ and $A = \oplus\{\ell : A_\ell\}_{\ell \in L}$.

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

By the induction hypothesis, $\mathcal{V} \;;\; C \;;\; \cdot \;;\; \Delta \vdash^g P :: (x : A_k)$. Using Lemma 8, we get $\mathcal{V} \;;\; C \wedge [\Delta]_L^? \wedge [A_k]_R^! \;;\; [\Delta]_L^\tau \;;\; \cdot \vdash^g P :: (x : [A_k]_R^\tau)$. Since this is a stable sequent, we reapply the $\oplus R_k$ rule, we get (note that $[A_k]_R^\tau = A_k$ and $[A_k]_R^! = \top$)

$$\frac{\mathcal{V} \;;\; C \wedge [\Delta]_L^? \;;\; [\Delta]_L^\tau \;;\; \cdot \vdash^g P :: (x : A_k)}{\mathcal{V} \;;\; C \wedge [\Delta]_L^? \;;\; [\Delta]_L^\tau \;;\; \cdot \vdash^g x.k \;;\; P :: [x : \oplus\{\ell : A_\ell\}]} \;\oplus R_k$$

Now, applying the $\oplus F_R$ rule,

$$\frac{\mathcal{V} \;;\; C \wedge [\Delta]_L^? \;;\; [\Delta]_L^\tau \;;\; \cdot \vdash^g x.k \;;\; P :: [x : \oplus\{\ell : A_\ell\}]}{\mathcal{V} \;;\; C \wedge [\Delta]_L^? \;;\; [\Delta]_L^\tau \;;\; \cdot \vdash^g x.k \;;\; P :: (x : \oplus\{\ell : A_\ell\})} \;\oplus F_R$$

Reapplying the move and !L rules, we get $\mathcal{V} \;;\; C \;;\; \cdot \;;\; \Delta \vdash^g x.k \;;\; P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})$.

- Case $(\oplus L)$: $P = $ case $x\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$ and $\Delta = \Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L})$.

$$\frac{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : A_\ell)\ {}_i\!\vdash^g Q_\ell :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L})\ {}_i\!\vdash^g \atop \text{case } x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)} \oplus L$$

By the induction hypothesis, $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (x : A_\ell) \cdot \Delta\ \vdash^g Q_\ell :: (z : C)$. First, we use Lemma 6 to get $\mathcal{V}\ ;\ C \wedge [C]^!_R\ ;\ \cdot\ ;\ (x : A_\ell) \cdot \Delta\ \vdash^g Q_\ell :: (z : [C]^\tau_R)$. Then, we use Lemma 7 successively on Δ to get $\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ (x : A_\ell)\ \vdash^g Q_\ell :: (z : [C]^\tau_R)$. Now, we apply the $\oplus L$ rule to get

$$\frac{\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ (x : A_\ell)\ \vdash^g Q_\ell :: (z : [C]^\tau_R)}{\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L, [x : \oplus\{\ell : A_\ell\}_{\ell \in L}]\ ;\ \cdot\ \vdash^g \atop \text{case } x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : [C]^\tau_R)} \oplus L$$

Applying the $\oplus F_L$ rule, we get $\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L, (x : \oplus\{\ell : A_\ell\}_{\ell \in L})\ ;\ \cdot\ \vdash^g$ case $x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : [C]^\tau_R)$. Reapplying the invertible rules $!R$ on C and move and $?L$ on $\Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L})$, we get $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L})\ \vdash^g$ case $x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)$.
- Case $(\&R, \&L_k)$: Analogous to $\oplus$ proofs.
- Case $(\multimap R)$: $P = y \leftarrow \text{recv } x\ ;\ P_y$ and $A = S \multimap B$.

$$\frac{\mathcal{V}\ ;\ C\ ;\ \Delta, (y : S)\ {}_i\!\vdash^g P_y :: (x : B)}{\mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash^g y \leftarrow \text{recv } x\ ;\ P_y :: (x : S \multimap B)} \multimap R$$

By the induction hypothesis, $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : S) \cdot \Delta\ \vdash^g P_y :: (x : B)$. Using Lemma 8, we get $\mathcal{V}\ ;\ C \wedge [B]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ (y : S)\ \vdash^g P_y :: (x : [B]^\tau_R)$. Note that since B is a positive type, $[B]^!_R = \top$ and $[B]^\tau_R = B$. Thus, we have $\mathcal{V}\ ;\ C \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ (y : S)\ \vdash^g P_y :: (x : B)$. Now, apply the $\multimap R$ rule to get

$$\frac{\mathcal{V}\ ;\ C \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ (y : S)\ \vdash^g P_y :: (x : B)}{\mathcal{V}\ ;\ C \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ \cdot\ \vdash^g \atop y \leftarrow \text{recv } x\ ;\ P_y :: [x : S \multimap B]} \multimap R$$

Now, we drop force using the $\multimap F_R$ rule to get $\mathcal{V}\ ;\ C \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ \cdot\ \vdash^g y \leftarrow \text{recv } x\ ;\ P_y :: (x : S \multimap B)$. Finally, we apply the $?L$, move rules to get $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta\ \vdash^g y \leftarrow \text{recv } x\ ;\ P_y :: (x : S \multimap B)$.
- Case $(\multimap L)$: $P = \text{send } x\ w\ ;\ Q$ and $\Delta = \Delta, (w : S), (x : S \multimap B)$.

$$\frac{\mathcal{V}\ ;\ C\ ;\ \Delta, (x : B)\ {}_i\!\vdash^g Q :: (z : C)}{\mathcal{V}\ ;\ C\ ;\ \Delta, (w : S), (x : S \multimap B)\ {}_i\!\vdash^g \atop \text{send } x\ w\ ;\ Q :: (z : C)} \multimap L$$

By the induction hypothesis, we get $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (x : B) \cdot \Delta\ \vdash^g Q :: (z : C)$. First, we use Lemma 6 to get $\mathcal{V}\ ;\ C \wedge [C]^!_R\ ;\ \cdot\ ;\ (x : B) \cdot \Delta\ \vdash^g Q :: (z : [C]^\tau_R)$. Then, we use Lemma 7 on Δ to get $\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ (x :$

$B)\ \vdash^g Q :: (z : [C]^\tau_R)$. Now, we apply the $\multimap L$ rule

$$\frac{\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L\ ;\ (x : B)\ \vdash^g Q::(z:[C]^\tau_R)}{\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L, (w : S), [x : S \multimap B]\ \vdash^g \atop \text{send } x\ w\ ;\ Q :: (z : [C]^\tau_R)} \multimap L$$

Applying the $\multimap F^2_L$ and $\multimap F^1_L$ rules in order, we get $\mathcal{V}\ ;\ C \wedge [C]^!_R \wedge [\Delta]^?_L\ ;\ [\Delta]^\tau_L, (w : S), (x : S \multimap B)\ \vdash^g$ send $x\ w\ ;\ Q :: (z : [C]^\tau_R)$. Now, reapplying the invertible rules, we get $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta, (w : S), (x : S \multimap B)\ \vdash^g$ send $x\ w\ ;\ Q :: (z : C)$.
- Case $(\otimes R, \otimes L, 1R, 1L)$: Analogous to $\multimap$ rules.
- Case (id) : $P = x \leftarrow y$.

$$\frac{}{C\ ;\ (y : A)\ {}_i\!\vdash^g x \leftarrow y :: (x : A)} \text{id}$$

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

$$\frac{\dfrac{\dfrac{\dfrac{}{\mathcal{V}\ ;\ C\ ;\ [y : S]\ ;\ \cdot\ \vdash^g x \leftarrow y :: (x : S)} \text{id}}{\mathcal{V}\ ;\ C\ ;\ (y : S)\ ;\ \cdot\ \vdash^g x \leftarrow y :: [x : S]} \text{id}L}{\mathcal{V}\ ;\ C\ ;\ (y : S)\ ;\ \cdot\ \vdash^g x \leftarrow y :: (x : S)} \text{id}R}{\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : S)\ \vdash^g x \leftarrow y :: (x : S)} \text{move}$$

If $A = ?\{\phi\}.\ A'$, by the induction hypothesis, we know that $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : A')\ \vdash^g x \leftarrow y :: (x : A')$. In the forcing calculus, we apply the derivation

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : A')\ \vdash^g x \leftarrow y :: (x : A')}}{\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : A')\ \vdash^g x \leftarrow y :: [x : A']} \text{lose}R}{\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : A')\ \vdash^g x \leftarrow y :: [x : ?\{\phi\}.\ A']} ?R}{\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : A')\ \vdash^g x \leftarrow y :: (x : ?\{\phi\}.\ A')} \text{id}R}{\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ (y : ?\{\phi\}.\ A')\ \vdash^g x \leftarrow y :: (x : ?\{\phi\}.\ A')} ?L$$

A similar argument would work for $A = !\{\phi\}.\ A'$.
- Case (spawn) : Since all types involved in a spawn are structural, this case is trivial.
- Case $(?R)$: $A = ?\{\phi\}.\ A$.

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

By the induction hypothesis, we get $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta\ \vdash^g P :: (x : A)$. We use Lemma 9 to get $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta\ \vdash^g P :: (x : ?\{\phi\}.\ A)$ since we know that $\mathcal{V}\ ;\ C \vDash \phi$.
- Case $(?L)$: $\Delta = \Delta, (x : ?\{\phi\}.\ A)$ and $P = Q$.

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

By the induction hypothesis, we get that $\mathcal{V}\ ;\ C \wedge \phi\ ;\ \cdot\ ;\ \Delta \cdot (x : A)\ \vdash^g Q :: (z : C)$. First, we invert the $!R$ rule to get $\mathcal{V}\ ;\ C \wedge \phi \wedge [C]^!_R\ ;\ \cdot\ ;\ \Delta \cdot (x : A)\ \vdash^g Q :: (z : [C]^\tau_R)$. Then, we apply the $?L$ rule to get $\mathcal{V}\ ;\ C \wedge [C]^!_R\ ;\ \cdot\ ;\ \Delta \cdot (x : ?\{\phi\}.\ A)\ \vdash^g Q :: (z : [C]^\tau_R)$. Then we reapply the $!R$ to get back $\mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta \cdot (x : ?\{\phi\}.\ A)\ \vdash^g Q :: (z : C)$.

$\square$