Practical Refinement Session Type Inference

Toby Ueno and Ankush Das

Boston University, Boston, MA, USA
{uenot,ankushd}@bu.edu

Abstract. Session types express and enforce safe communication in concurrent message-passing systems by statically capturing the interaction protocols between processes in the type. Recent works extend session types with arithmetic refinements, which enable additional fine-grained description of communication, but impose additional annotation burden on the programmer. To alleviate this burden, we propose a type inference algorithm for a session type system with arithmetic refinements. We develop a theory of subtyping for session types, including an algorithm which we prove sound with respect to a semantic definition based on type simulation. We also provide a formal inference algorithm that generates type and arithmetic constraints, which are then solved using the z3 SMT solver. The algorithm has been implemented on top of the Rast language, and includes 3 key optimizations that make inference feasible and practical. We evaluate the efficacy of our inference engine by evaluating it on 6 challenging benchmarks, ranging from unary and binary natural numbers to linear λ-calculus. We show the performance benefits provided by our optimizations in coercing z3 into solving the arithmetic constraints in reasonable time.

Keywords: Session Types · Type Inference · Refinement Types

1 Introduction

Binary session types [5, 6, 26–28] provide a structured way to express communication protocols in concurrent message-passing systems. Types are assigned to bi-directional channels connecting processes and describe the type and direction of communication. Type checking then ensures that processes on either end of the channel respect the interaction described by the channel type, i.e., they follow both the type and direction. As a result, standard type safety theorems guarantee session fidelity (preservation), i.e., protocols are not violated at runtime and deadlock-freedom (progress), i.e., no process ever gets stuck. However, even in the presence of recursion, vanilla session types can only specify very basic protocols which has led to a number of extensions, such as context-free session types [41, 42], label-dependent session types [43], and general dependent session types [24, 44–46].

One such extension is to integrate session types with refinements [12, 13, 15], in the style of DML [18, 47] to enable lightweight verification of message-passing programs. For instance, consider the simple (unrefined) session type for naturals:

$$\mathit{type}\ \mathsf{nat} \triangleq \oplus\{\mathbf{succ} : \mathsf{nat}, \mathbf{zero} : \mathbf{1}\}$$

The communication protocol governed by this type is that the sender can either produce the **succ** label or the **zero** label, on which the receiver must branch upon. However, the type does not give any information beyond this basic behavior. In particular, the type does not define exactly how many **succ** labels will be produced by the type. To address this limitation, one can refine the type:

$$\mathit{type}\ \mathsf{nat}[n] \triangleq \oplus\{\mathbf{succ} : ?\{n > 0\}.\, \mathsf{nat}[n-1], \mathbf{zero} : ?\{n = 0\}.\, \mathbf{1}\}$$

The type is indexed by a natural number n that denotes the number of **succ** labels produced by the sender. On sending the **succ** label, the type $?\{n > 0\}.\, \mathsf{nat}[n-1]$ describes that the sender must produce a *proof* that $n > 0$, after which the type transitions to $\mathsf{nat}[n-1]$, meaning that the sender will now produce $(n-1)$ **succ** labels. Such refinements enable lightweight verification of protocol processes; for instance, one can define that an add process can take two naturals $\mathsf{nat}[m]$ and $\mathsf{nat}[n]$ to produce $\mathsf{nat}[m+n]$.

As useful as refinements are, they introduce a significant syntactic burden on the programmer. First, programmers need to annotate process declarations (like add) with the refined type of all channels connected to it. Second, they need to add explicit assertions and assumptions in their programs for successful type checking. Finally, there is usually very little feedback from the compiler in the event that the type annotations are incorrect. As programs get more complicated, so do the annotations, further exacerbating the cost of this overhead.

To address this syntactic overhead, this paper proposes an *automatic type inference* algorithm for refinement session types. The goal is to infer the (possibly recursive) type of processes from their definitions. However, this task poses several challenges, both in theory and implementation. First, type checking, and therefore, type inference is *undecidable* [13] even in the presence of simple linear arithmetic refinements. Thus, any practical algorithm must be approximate and rely on appropriate heuristics. Second, session types follow *structural* and *equi-recursive* typing; type equality does not depend on type names, rather the structure of types. Moreover, there are no explicit folds or unfolds in programs to handle recursion. This more expressive nature of typing introduces complications for the inference engine, which needs to keep track of the type structure based on its communication. Third, the most *general* refined type for a process is often not the most useful one. As an example, consider a process that produces $\mathsf{nat}[0]$ by sending a **zero** label and terminating. The most general type of this process is $\oplus\{\mathbf{zero} : \mathbf{1}\}$, which is a *subtype* of $\mathsf{nat}[0]$, but probably not the most useful type. On the other hand, producing *any* supertype of the most general type is also not satisfactory. As an example, we can infer many possibilities for index arguments; for example, a valid type for the add process is to take $\mathsf{nat}[2m]$ and $\mathsf{nat}[n]$ to produce $\mathsf{nat}[2m+n]$. Although valid, this is not the most precise type for add. In particular, this type would not work for numbers where the first argument is odd. Therefore, the most general type is sometimes useful, but not always— the type that is reported to the user must be general enough to accommodate a large set of programs.

To infer a more realistic type for processes, we extend the notion of subtyping in session types to arithmetic refinements. To that end, we adopt and generalize the declarative definition for subtyping introduced in seminal work by Gay and Hole [20]. Using this definition as our foundation, we introduce a set of algorithmic subtyping rules that produce a set of type constraints from which the most general type can be inferred. We show that this algorithm is *sound*: if A is a subtype of B according to our algorithm, it must be so under our definition. Next, we introduce a range of heuristics in our inference engine that carefully balance precision and practicality to achieve user-friendly types.

We have implemented our type inference engine on top of the Rast [10–12] programming language. Rast is a language targeted towards analyzing parallel and sequential complexity of session-typed message-passing programs. Rast supports type refinements and type checking, but not type inference. Our implementation performs inference in two stages: first inferring the base session type for each channel, and then inferring the type indices. Separating out these two stages not only simplifies the inference engine but rejects ill-typed programs early. The first stage of inference collects the subtyping constraints and solves them using a standard unification algorithm. The second stage extracts the arithmetic constraints from subtyping and ships them to the z3 SMT solver [16]. To improve the performance of type inference, we introduce 3 key *optimizations* in both stages: *(i) transitivity* to eliminate the intermediate types in a process definition, *(ii) polynomial templates* to reduce the search space of arithmetic expressions for z3, and *(iii) theory of reals* to find a satisfying assignment faster, which is then converted to natural numbers, if possible.

We analyze the efficacy of our inference engine and the performance improvements provided by the optimizations using 6 benchmarks from the Rast language. These include unary and binary natural numbers indexed by their value, lists indexed by their size, and linear λ-calculus expressions indexed by the size of the term. We infer the type of a few standard processes, e.g., standard arithmetic operations on numbers like addition and doubling; standard list operations like append and split; and evaluation of expressions in the calculus. Our experiments reveal that transitivity provides an order of magnitude performance benefit in the first stage. Moreover, both polynomial templates and theory of reals are important to make the second stage feasible. Without these two optimizations, z3 times out on even the simplest of examples. We conclude that all 3 optimizations are necessary to make inference scalable and practical.

To summarize, the paper makes the following contributions:

- A declarative definition of subtyping and a sound algorithm for subtyping of refinement session types (Section 4),
- A type inference algorithm that generates typing and arithmetic constraints and its proof of soundness (Section 5),
- An implementation (Section 6) of our inference engine along with the 3 optimizations, and
- An evaluation (Section 7) of our algorithm on 6 challenging benchmarks.

The supplementary material contains the complete set of subtyping and inference rules along with proofs of soundness.

2 Motivating Examples

This section will informally introduce the main challenges underlying our type inference engine through a series of motivating examples. We follow the syntax of the Rast language [12, 15] for message-passing programs where communication happens via bi-directional channels that are *typed* using a session type. This session type can be viewed as being *offered* by a *provider process*, while being *used* by a *client process*. To communicate safely, the provider and client must perform dual matching actions, as governed by the type. Revisiting the example of natural numbers, the basic type is defined as

$$\mathit{type}\ \mathsf{nat} = \oplus\{\mathbf{zero} : \mathbf{1},\ \mathbf{succ} : \mathsf{nat}\}$$

The $\oplus$ type is associated with a set of labels $\{\mathbf{zero}, \mathbf{nat}\}$ of which the channel provider can choose one to send. The type after the message transmission is indicated by the colon: if the provider sends **zero**, the continuation type is **1** type, indicating termination and closing of the channel; if instead the provider sends **succ**, the type recurses back to `nat`, meaning the protocol simply repeats.

Processes can be defined in Rast using a *declaration* describing the process type and a *definition* describing the implementation. We follow the declaration and definition of a process called `two` that represents the number 2:

```
decl two : . ⊢  (x: nat)
proc x ← two = x.succ; x.succ; x.zero; close x
```

The first line shows the declaration: the left of the turnstile ($\vdash$) shows the channels (and their types) used by the process, while the right shows the offered channel and type. The `two` process does not use any channels, hence a dot ($\cdot$) on the left and offers channel `x` of type `nat`. The process definition for `two` (beginning with `proc`) expresses that the process sends exactly two **succ** labels on channel x (`x.k` denotes sending label `k` on channel `x`) followed by a **zero** label, and then ultimately terminates by closing the channel `x`. Without type inference, Rast only supports type checking meaning the programmer is required to provide both the process declaration and definition. The goal of this work is to automatically generate the declaration given the definition, in the style of languages like OCaml and SML.

Structural Subtyping in Inference At first, we might try a naïve approach at type inference to produce the most general type. For instance, if we are sending label `k` on offered channel `x`, the most general type of `x` must be $\oplus\{k : A_k\}$, where A_k is the type of channel `x` after the communication. If we apply this technique on the process `two` above, we will arrive at the following type:

```
proc x ← two = x.succ;          % x: ⊕{succ : ⊕{succ : ⊕{zero : 1}}}
               x.succ;          % x: ⊕{succ : ⊕{zero : 1}}
               x.zero;          % x: ⊕{zero : 1}
               close x          % x: 1
```

It is easier to follow this type inference bottom-up to build the inferred type incrementally. The last operation on `x` is closing, hence its type at that point

must be **1**, as indicated by the comment on the right. Next, we send the label **zero**, meaning the most general type for `x` must be $\oplus\{\textbf{zero} : \textbf{1}\}$ because this is the simplest type that allows sending of label **zero**. Following this intuition for the next 2 labels, we get the type of `x` to be $\oplus\{\textbf{succ} : \oplus\{\textbf{succ} : \oplus\{\textbf{zero} : \textbf{1}\}\}\}$.

This type however is not satisfactory for a variety of reasons. First, this type is not generalizable: the process types for `one`, `two`, `three`, ... will all be different which, in turn, hampers code reuse. Even a programmer would be more inclined to use `nat` as the process type. Second, this type would not be usable for a larger process like `add` which needs to be defined for generic natural numbers, and not every particular number and type. Second, although the generated type (or the *most general type*) is correct for typechecking purposes, using this type in type error messages would be unncessarily verbose and will progressively become more incomprehensible as programs grow in complexity.

Evident from this is the necessity of a notion of *subtyping*: we would like to have a system where $\oplus\{\textbf{succ} : \oplus\{\textbf{succ} : \oplus\{\textbf{zero} : \textbf{1}\}\}\}$ is a subtype of `nat`. This subtyping system must also account for the *structural* nature of types rather than a *nominal* one. This is in contrast with refinement type systems for functional languages like Liquid Types [38] and DML [47] where the base types can be inferred using a standard Hindley-Milner inference algorithm. As will be evident from our subtyping algorithm, there is no notion of base types in structural type systems. Concretely, this decision entails that message labels like **zero** and **succ** are not tied to the `nat` type specifically, unlike e.g. the constructors of an algebraic datatype in most functional languages. For instance, we can use the same constructors for multiple types as follows:

$$\textit{type}\ \mathsf{even} = \oplus\{\textbf{zero} : \textbf{1},\ \textbf{succ} : \mathsf{odd}\} \qquad \textit{type}\ \mathsf{odd} = \oplus\{\textbf{succ} : \mathsf{even}\}$$

And both `even` and `nat` are valid types for process `two`. Thus, our algorithm for subtyping must analyze the (possibly mutually recursive) structures of types. Our approach is to infer the most general type that is *provided in the signature* by the programmer. With subtyping in place, we can say that a process offers a channel of the most general type *or any valid supertype*, thereby significantly increasing readability while maintaining correctness.

2.1 Introducing Refinements

The notion of subtyping becomes more involved in the presence of *arithmetic refinements*. For instance, revisiting the refinement of natural number type:

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

The refinements enforce that the provider is only permitted to send the **zero** label when $n = 0$, and the **succ** when $n > 0$, respectively. By specifying refinements on types, we are able to convey more type-level information about our processes: for instance, we can guarantee that the amended `two` process will send *exactly* two **succ** messages if the offered type is `nat`[2].

```
decl two : . ⊢  (x: nat[2])
proc x ← two = x.succ; assert x {2 > 0}; x.succ; assert x {1 > 0};
               x.zero; assert x {0 = 0}; close x
```

We describe the implementation of the process to show how the proof obligations are upheld. Each time we send a **succ** message, we need to send a proof that $n > 0$ which is achieved using an assertion. Therefore, the process sends first asserts that $\{2 > 0\}$ after sending **succ** when the type of `x` is `nat`[2]. After this, the type transitions to `nat`[1], hence the process sends another **succ** followed by asserting that $\{1 > 0\}$. Finally, the type transitions to `nat`[0], hence the process sends the **zero** label and an assertion that $\{0 = 0\}$ and closes.

As can be observed from this example, these assertions add a significant syntactic overhead on the programmer. To eliminate this burden, our inference algorithm also performs *program reconstruction* which inserts these assertions automatically. Our inference engine deduces these assertions by following the type structure of the refined nat type. Remarkably, this reconstruction is performed effectively even if multiple types (e.g., even, odd, nat) are using the same label constructors.

Refinement Inference. In addition to inferring type names like `nat`, we must also infer the *refinements* on those types. These refinements are not guaranteed to be numbers, as in the `two` process; rather, they are arbitrary expressions which can involve free arithmetic variables. The following process, which adds two numbers, is shown after reconstruction, but before refinement inference:

```
decl add[m][n]: (x: nat[e0(m,n)]) (y: nat[e1(m,n)]) ⊢  (z: nat[e2(m,n)])
proc z ← add[m][n] x y = case x (
    zero  ⇒  assume x {e0(m,n) = 0}; wait x; z ↔ y
  | succ  ⇒  assume x {e0(m,n) > 0};
        z.succ; assert z {e2(m,n) > 0};
        z ← add[m-1][n] x y )
```

The `add` process case analyzes on `x`. If `x` sends the **zero** label (meaning `x` is 0), we simply *identify* channels `y` and `z` (equivalent to returning `y`) in a functional setting. On the other hand, in the **succ** branch, we *assume* that `x` is greater than 0, we send the **succ** label on `z` followed by asserting that $e_2(m, n) > 0$ (as required by `nat` type). Finally, we recurse by calling the `add` process again.

Suppose we have already determined that each channel has type `nat`; we must still find expressions e_0, e_1, and e_2 in variables m, n such that all of our assertions hold. We delegate this non-trivial task to z3: we first identify any *constraints* on any candidate expressions, e.g. that $e_0(m, n) = 0$ implies $e_2(m, n) = e_1(m, n)$ since channel y is forwarded on to z in the **zero** branch. Similarly, in the **succ** branch, we get that $e_0(m, n) > 0$ implies $e_2(m, n) > 0$. With these constraints, we query z3 to find a satisfying assignment for our refinements. In the case of `add`, we would expect that $e_0(m, n) = m$, $e_1(m, n) = n$, and $e_2(m, n) = m + n$.

Naturally, our first attempt was to treat e_i's as *uninterpreted functions*, which z3 has the ability to solve for. We simply shipped our constraints over to z3, asking for a satisfying assignment. In practice, however, this approach is not feasible: either the solver times out or it returns a non-polynomial expression that cannot be expressed in our refinement layer, such as if-then-else constructs. Thus, we first substitute each expression with a *polynomial template*: for instance, we

transform $e_0(m, n)$ into $c_0m+c_1n+c_2$, and only ask z3 to solve for the coefficients c_i's. Any expression in our language takes this form, so we lose no generality, and we find that this optimization greatly improves both the reliability and performance of z3. In fact, polynomial templating is only one such optimization which enables our algorithm to succeed on reasonably complex examples; we detail others in Section 6.

3 Background on Refinement Session Types

We describe the language of session types upon which we implement type inference. We first introduce the basic session types, which constitute the core of our communication protocols, followed by the refinement layer, which extends the core with constructs to send and receive proofs and witnesses. We include detailed examples of each type constructor in Appendix A.

3.1 Basic Session Types

Our basic session type system is in correspondence with intuitionistic linear logic [5]. The types allow exchange of labels, other channels, and close, i.e., termination messages. The type and process syntax is defined as:

$$
\begin{array}{rll}
A, B ::= & \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} & \text{(internal and external choice)} \\
& \mid A \otimes B \mid A \multimap B & \text{(tensor and lolli)} \\
& \mid \mathbf{1} \mid V & \text{(unit and type variable)} \\
P ::= & x.k \,;\, P \mid \mathsf{case}\ x\ (\ell \Rightarrow P_\ell) & \text{(send and receive labels)} \\
& \mid \mathsf{send}\ x\ e \,;\, P \mid y \leftarrow \mathsf{recv}\ x \,;\, P & \text{(send and receive channels)} \\
& \mid \mathsf{close}\ x \mid \mathsf{wait}\ x \,;\, P & \text{(close and wait for close)} \\
& \mid x \leftrightarrow y \mid x \leftarrow f\ \overline{y} \,;\, P & \text{(forward and spawn)}
\end{array}
$$

The structural types are divided into pairs of *dual* types— namely, we say $\oplus$ and $\&$ are dual to each other, as are $\otimes$ and $\multimap$. The dual of a type exhibits communication in the opposite direction. For instance, a provider of a channel x of type $\oplus\{\ell : A_\ell\}_{\ell \in L}$, must *send* one of the labels $k \in L$ via the expression $x.k$ and continue as A_k. Dually, if a provider offers $x : \&\{\ell : A_\ell\}_{\ell \in L}$, it *receives* a label in L, which it case-analyzez via the expression $\mathsf{case}\ x\ (\ell \Rightarrow P_\ell)$; the process continues as P_ℓ and the channel as A_ℓ. The other types, $\otimes$ and $\multimap$, are analogous, but for sending *channels* instead of labels: provider of $x : A \otimes B$ will send a channel e of type A via $\mathsf{send}\ x\ e$. Dually, a provider of $x : A \multimap B$ will receive a channel of type A which it binds to y via $y \leftarrow \mathsf{recv}\ x$. Finally, a provider of $(x : \mathbf{1})$ must use $\mathsf{close}\ x$ to terminate channel x and the process, while the client takes the form $\mathsf{wait}\ x \,;\, P$, i.e., waits for x to close and then continue as P. The remaining pieces of syntax, $x \leftrightarrow y$ and $x \leftarrow f\ \overline{y}$, refer to forwarding and spawning, respectively. Forwarding $x \leftrightarrow y$ means that we identify channels x and y passing all messages between them, and spawning a new process f is analogous to a function call in other languages: we provide f with channels $\overline{y}$ and receive a channel x back. Finally, note that a process *offers* one channel and *consumes* any number of other channels. And the role of a type flips based on whether it is provided or consumed.

3.2 Refinement Layer

Types are refined using the following type and process constructs:

$$
\begin{array}{lll}
A ::= & ?\{\phi\}.A \mid !\{\phi\}.A & \text{(assertion and assumption)} \\
& \mid \exists n.A \mid \forall n.A & \text{(quantifiers)} \\
& \mid V[\overline{e}] & \text{(indexed type variable)} \\
P ::= & \mathsf{assert}\ x\ \{\phi\}\ ;\ P \mid \mathsf{assume}\ x\ \{\phi\}\ ;\ P & \text{(assert and assume)} \\
& \mid \mathsf{send}\ x\ \{e\}\ ;\ P \mid \{n\} \leftarrow \mathsf{recv}\ x\ ;\ P_n & \text{(send and receive witnesses)}
\end{array}
$$

The crucial types for verifying program behavior are ?, and its dual, !. When we provide $(x : ?\{\phi\}.A)$, we send a *proof* of ϕ along x via $\mathsf{assert}\ x\ \{\phi\}$, and the client of x receives this proof of ϕ via $\mathsf{assume}\ x\ \{\phi\}$. Note that no actual proof objects are sent at runtime; instead, we merely communicate that such a proof exists. As before, the dual $(x : !\{\phi\}.A)$ reverses the direction of communication, allowing for the provider of x to receive a proof and for a client to send one. We also allow channels to send and receive *witnesses* through the quantifier types $\exists$ and $\forall$. A provider of $(x : \exists n.\, A)$ sends a witness expression e via $\mathsf{send}\ x\ \{e\}$, a client of x receives the value of e for n and substitutes $[e/n]A$ in the continuation via $\{n\} \leftarrow \mathsf{recv}\ x$. Type $\forall n.\, A$ exhibits dual behavior. We use these witnesses to communicate natural numbers which can themselves be used in future assertions and assumptions. For instance, we could write $\exists k.\ ?\{n = 2 * k\}.\mathsf{nat}[k]$ to signify that n is even and k is its witness.

The language for arithmetic expressions e and propositions ϕ is standard and described below. We use i to denote constant numbers and n for variables.

$$
\begin{array}{l}
e ::= i \mid e + e \mid \mid e - e \mid e * e \mid n \\
\phi ::= e = e \mid e > e \mid \top \mid \bot \mid \neg\phi \mid \phi \vee \phi
\end{array}
$$

Although, in principle, our grammar allows arbitrary polynomial arithmetic expressions, most of our examples are restricted to linear expressions. We found that for any higher-degree polynomials, inference via z3 becomes infeasible.

3.3 Type Variables and Signatures

We operate within a signature Σ containing *type definitions*, *process declarations*, and *process definitions* as follows:

$$
\begin{array}{lll}
\Sigma ::= & \cdot \mid \Sigma, V[\overline{n} \mid \phi] = A & \text{(type definition)} \\
& \mid \Sigma, \Delta \vdash f[\overline{n}] :: (z : A) & \text{(process declaration)} \\
& \mid \Sigma, x \leftarrow f[\overline{n}]\ \overline{y} = P & \text{(process definition)} \\
\Delta ::= & \cdot \mid \Delta, (x : A) & \text{(process context)}
\end{array}
$$

The type definition $V[\overline{n} \mid \phi] = A$ means that V is indexed by some refinements $\overline{n}$ such that ϕ holds; when we write $V[\overline{e}]$, we interpret this as $A[\overline{e}/\overline{n}]$. We adopt an *equirecursive* and *structural* approach to types, foregoing any explicit communication for unfolding. The process *declaration* $\Delta \vdash f[\overline{n}] :: (z : A)$ tells us

that f offers a channel z of type A, and that f consumes all channels x of type A in Δ. The process f is also indexed by some refinements $\overline{n}$ which can be freely used in the types in Δ and A. Its corresponding *definition* $x \leftarrow f[\overline{n}]\ \overline{y} = P$ is likewise indexed by $\overline{n}$, but instead of type information, gives us the process expression P for f using channels x and $\overline{y}$. Again, variables in $\overline{n}$ can appear freely in definitions, e.g., in assertions and assumptions.

4 Subtyping

To define a subtyping algorithm and prove its correctness, we must first begin with a semantic definition of subtyping, which is adopted from Gay and Hole [20]'s definition using a *type simulation*.

Type Simulation Our definition of a type simulation generalizes prior work by Das and Pfenning [14], who defined a type *bi*simulation for the purposes of type equality. We rely upon the notion of *closed* types: a type is closed if it contains no free arithmetic variables. To account for types with refinement variables, we introduce the notion of *validity*. Recall that the signature Σ collects *all* type definitions of the form $V[\overline{n} \mid \phi] = A$. This signature is then called *valid* if the implicit constraint ϕ holds for all occurrences $V[\overline{e}]$ in the signature. Formally, if $V[\overline{e}]$ appears in any type definition in Σ, then $\vDash \phi[\overline{e}/\overline{n}]$. We further require a valid signature to only contain *contractive* type definitions, disallowing definitions of the form $V[\overline{n} \mid \phi] = V'[\overline{e}]$. We include a complete formal definition of valid types and signatures in Appendix B.

Definition 1. *On a valid signature Σ, we define $\mathsf{unfold}_\Sigma(V[\overline{e}]) = A[\overline{e}/\overline{n}]$ if $V[\overline{n} \mid \phi] = A \in \Sigma$ and $\mathsf{unfold}_\Sigma(A) = A$ otherwise (when A is not a type variable).*

Definition 2 (Type Simulation). *A relation $\mathcal{R}$ on closed, valid types is a* type simulation *under Σ if, for any types A, B such that $(A, B) \in \mathcal{R}$, when we take $S = \mathsf{unfold}_\Sigma(A)$ and $T = \mathsf{unfold}_\Sigma(B)$, the following holds:*

i) If $S = \oplus\{\ell : A_\ell\}_{\ell \in L}$, then $T = \oplus\{m : B_m\}_{m \in M}$. Also, $L \subseteq M$ and $(A_\ell, B_\ell) \in \mathcal{R}$ for all $\ell \in L$.
ii) If $S = \&\{\ell : A_\ell\}_{\ell \in L}$, then $T = \&\{m : B_m\}_{m \in M}$. Also, $L \supseteq M$ and $(A_m, B_m) \in \mathcal{R}$ for all $m \in M$.
iii) If $S = A_1 \otimes A_2$, then $T = B_1 \otimes B_2$. Also, $(A_1, B_1) \in R$ and $(A_2, B_2) \in R$.
iv) If $S = A_1 \multimap A_2$, then $T = B_1 \multimap B_2$. Also, $(B_1, A_1) \in R$ and $(A_2, B_2) \in R$.
v) If $S = \mathbf{1}$, then $T = \mathbf{1}$.
vi) If $S = ?\{\phi\}.A$, then $T = ?\{\psi\}.B$. Also, either $\vDash \phi$, $\vDash \psi$, and $(A, B) \in \mathcal{R}$; or $\vDash \neg\phi$.
vii) If $S = !\{\phi\}.A$, then $T = !\{\psi\}.B'$. Also, either $\vDash \psi$, $\vDash \phi$, and $(A, B) \in \mathcal{R}$; or $\vDash \neg\psi$.
viii) If $S = \exists m.A$, then $T = \exists n.B$. Also, $\forall i \in \mathbb{N}$ we have $(A[i/m], B[i/n]) \in \mathcal{R}$.
ix) If $S = \forall m.A$ then $T = \forall n.B$. Also, $\forall i \in \mathbb{N}$ we have $(A[i/m], B[i/n]) \in \mathcal{R}$.

Each type constructor has a corresponding case in Definition 2 which intuitively reduces to the notion that the first type in a pair A should *simulate* the

behavior of the second type B. Concretely, the communication behaviors allowed by A must be a subset of behaviors allowed by B. For instance, in case (i) for $\oplus$, we require that the label-set for A is a subset of the label-set for B i.e. every label that a provider of A can send can be received by a client of B but not vice-versa. Likewise, case (iv) for $S = A_1 \multimap A_2$ and $T = B_1 \multimap B_2$ notably adds (B_1, A_1) to $\mathcal{R}$ since $\multimap$ (like arrow types) is contravariant in the first argument. Cases $(i) - (v)$ have been borrowed from Gay and Hole [20]'s subtyping definition.

Worth noting are novel cases $(vi) - (ix)$ as they concern refinements. Case (vi) states that for $?\{\phi\}.\, A$, either ϕ must be true or false. In the former case, ψ must also be true and $(A, B) \in \mathcal{R}$. However, in the latter case, the relation holds vacuously since the provider of such a channel will cease to communicate, which effectively simulates *any* continuation type. Case (vii) is analogous but flips the directionality of constraints. Finally, case $(viii)$ (resp. (ix)) say that a quantified type $\exists m.\, A$ (resp. $\forall m.\, A$) simulates another one if all substitutions of m are already in the type simulation. With type simulations in tow, we then define subtyping simply as follows:

Definition 3. *For closed valid types A and B, we say $A <: B$, i.e. A is a* subtype *of B, if there is a type simulation $\mathcal{R}$ such that $(A, B) \in \mathcal{R}$.*

4.1 Algorithmic Subtyping

Subtyping is fundamentally undecidable in the presence of refinements. This fact is entailed by the fact that even the *simpler* problem of type equality is undecidable in the presence of refinements, as shown by prior work [14]. Nevertheless, we propose a sound algorithm which approximates subtyping, expressed via a series of inference rules with a primary judgment of the form $\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Gamma \Vdash A <: B$. Here, $\mathcal{V}$ and $\mathcal{C}$ respectively represent the list of free arithmetic variables and the governing constraint, and we likewise invoke the auxiliary judgment $\mathcal{V}\ ;\ \mathcal{C} \vDash \phi$ to represent semantic entailment. Γ is a list of closures of the form $\langle \mathcal{V}\ ;\ \mathcal{C}\ ;\ V_1[\overline{e_1}] <: V_2[\overline{e_2}] \rangle$ that have already been encountered which we capture for the purposes of loop detection. We omit some standard rules for brevity; the full series of subtyping rules can be found in appendix C.

Figure 1 describes selected rules for subtyping concerning arithmetic refinements, where the most interesting rules are $\mathsf{st}_?$ and $\mathsf{st}_!$ rules. The first premise of rule $\mathsf{st}_?$ states that ϕ must imply ψ, capturing the intuition of Definition 2 that either ϕ and ψ both hold or ϕ is false. The second premise requires that continuation type A must be a subtype of B under the constraint $\mathcal{C} \wedge \phi$. Rule $\mathsf{st}_!$ is analogous, only flipping the direction of implication, similar to Definition 2. Rule $\mathsf{st}_\bot$ handles the cases where the constraint $\mathcal{C}$ is contradictory. Under such a constraint, arbitrary types A and B are subtypes since such a situation will never arise at runtime. This rule comes in handy when type checking branches that are impossible due to refinements (e.g. **zero** branch on type $\mathsf{nat}[n+1]$).

The heart of loop detection lies in the $\mathsf{st}_{\mathsf{expd}}$ and $\mathsf{st}_{\mathsf{def}}$ rules, each of which may apply when we encounter type names $V_1[\overline{e_1}]$ and $V_2[\overline{e_2}]$. $\mathsf{st}_{\mathsf{expd}}$ effectively unrolls those type names according to our signature Σ, but it also generates a closure

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash \phi \to \psi \qquad \mathcal{V}\,;\,\mathcal{C} \wedge \phi\,;\,\Gamma \Vdash A <: B}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Gamma \Vdash ?\{\phi\}.A <: ?\{\psi\}.B}\ \mathsf{st}_?$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash \psi \to \phi \qquad \mathcal{V}\,;\,\mathcal{C} \wedge \psi\,;\,\Gamma \Vdash A <: B}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Gamma \Vdash !\{\phi\}.A <: !\{\psi\}.B}\ \mathsf{st}_!$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash \bot}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Gamma \Vdash A <: B}\ \mathsf{st}_\bot$$

$$\frac{V_1[\overline{v_1}|\phi_1] = A \in \Sigma \qquad V_2[\overline{v_2}|\phi_2] = B \in \Sigma \qquad \gamma = \langle \mathcal{V}\,;\,\mathcal{C}\,;\,V_1[\overline{e_1}] <: V_2[\overline{e_2}] \rangle \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Gamma, \gamma \Vdash A[\overline{e_1}/\overline{v_1}] <: B[\overline{e_2}/\overline{v_2}]}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Gamma \Vdash V_1[\overline{e_1}] <: V_2[\overline{e_2}]}\ \mathsf{st}_{\mathsf{expd}}$$

$$\frac{\langle \mathcal{V}'\,;\,\mathcal{C}'\,;\,V_1[\overline{e_1}'] <: V_2[\overline{e_2}'] \rangle \in \Gamma \qquad \mathcal{V}\,;\,\mathcal{C} \vDash \exists \mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Gamma \Vdash V_1[\overline{e_1}] <: V_2[\overline{e_2}]}\ \mathsf{st}_{\mathsf{def}}$$

Fig. 1: A selection of rules for the subtyping algorithm.

γ and adds it to our context Γ, denoting that we have "seen" V_1 and V_2, refined by these particular expressions $\overline{e_1}$ and $\overline{e_2}$, under this particular $\mathcal{V}$ and $\mathcal{C}$. Now if, later in our algorithm, we encounter those same V_1 and V_2 again, we eagerly follow the $\mathsf{st}_{\mathsf{def}}$ rule. Our new $\mathcal{V}$ and $\mathcal{C}$ are likely different than those in γ, so the second premise asserts (informally) that we can find a substitution between them, and if such a substitution does exist, we conclude that our algorithm succeeds. This coinductive structure allows our algorithm to handle types which could otherwise be expanded infinitely. This is also the source of incompleteness of our algorithm: since we can only expand a finite number of times, we must apply the $\mathsf{st}_{\mathsf{def}}$ rule eventually, which may then fail to detect a loop.

4.2 Soundness

We now prove that our algorithm is sound with respect to Definition 3. Proving subtyping of $A <: B$ effectively reduces to *constructing* a type simulation $\mathcal{R}$ such that $(A, B) \in \mathcal{R}$. The main intuition here is that such a type simulation can be constructed from the subtyping derivation built using our algorithmic rules. We here motivate the key ideas; a full proof sketch can be found in Appendix D.

Definition 4. *We define $\forall \mathcal{V}.\mathcal{C} \to A <: B$ (read: for all $\mathcal{V}$, $\mathcal{C}$ implies A is a subtype of B) if there exists a type simulation $\mathcal{R}$ such that, for all ground substitutions σ over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$, we get $(A[\sigma], B[\sigma]) \in \mathcal{R}$.*

This definition extends the idea of subtyping from closed types (e.g., $\mathsf{nat}[2]$) to symbolic types (e.g., $\mathsf{nat}[n]$). This is necessary since our subtyping algorithm works on symbolic types. The intuition behind this definition is that A is a subtype of B if for all substitutions σ that satisfy $\mathcal{C}$, $A[\sigma]$ is a subtype of $B[\sigma]$. As an example, we can say $\forall n.\, n > 0 \to \mathsf{nat}[n] <: \mathsf{nat}[n]$ because $\mathsf{nat}[1] <: \mathsf{nat}[1]$, $\mathsf{nat}[2] <: \mathsf{nat}[2]$, and so on.

Definition 5. *Given a derivation $\mathcal{D}$ of $\mathcal{V}\,;\,\mathcal{C}\,;\,\Gamma \Vdash A <: B$, we define the set of closures $S(\mathcal{D})$ such that, for each sequent $\mathcal{V}'\,;\,\mathcal{C}'\,;\,\Gamma' \Vdash A' <: B'$, we include $\langle \mathcal{V}'\,;\,\mathcal{C}'\,;\,A' <: B' \rangle \in S(\mathcal{D})$.*

This set of closures from a derivation is exactly what we need to construct the type simulation. The key idea here is that for a valid derivation, all these closures must represent valid subtyping relations as well. Finally, the type simulation is constructed by applying ground substitutions to all closures found in the derivation. This intuition is captured in the following theorem.

Theorem 1 (Soundness). *If* $\mathcal{V} \,;\, \mathcal{C} \,;\, \cdot \Vdash A <: B$*, then* $\forall \mathcal{V}.C \to A <: B$*.*

Proof. By construction of the type simulation

$$\mathcal{R} = \{(A[\sigma], B[\sigma]) \mid \langle \mathcal{V} \,;\, \mathcal{C} \,;\, A <: B \rangle \in S(\mathcal{D}) \text{ and } \sigma : \mathcal{V} \text{ and } \vDash \mathcal{C}[\sigma]\}$$

We show in Appendix D that $\mathcal{R}$ is indeed a type simulation.

5 Inference Algorithm

This section presents our rules for constraint generation, which form the theoretical backbone of our type inference algorithm. The rules themselves are in close correspondence with the standard typechecking rules for our language. Our primary judgment takes the same form $\mathcal{V} \,;\, \mathcal{C} \,;\, \Delta \vdash P :: (z : C)$ for a program P offering channel z of type C, and consuming all channels $(x : A_x) \in \Delta$. As with subtyping, we also operate within the context of a list of free arithmetic variables $\mathcal{V}$ and a governing constraint $\mathcal{C}$. For comparison, we include below the typechecking rule for $\oplus R$:

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

Since we are doing inference as opposed to typechecking, we amend this rule to reflect that we are not given x's type structure *a priori*. We treat it instead as a variable to which we apply a *type constraint*, which conveniently takes the form of the subtyping judgment $\mathcal{V} \,;\, \mathcal{C} \vDash A <: B$ for types A and B asserting that A is a subtype of B. We highlight the introduced intermediate type names and new arithmetic variables in blue, and assume all such names are fresh.

$$\frac{\mathcal{V} \,;\, \mathcal{C} \Vdash \oplus\{k : B\} <: A \qquad \mathcal{V} \,;\, \mathcal{C} \,;\, \Delta \vdash P :: (x : B)}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Delta \vdash x.k \,;\, P :: (x : A)} \oplus\text{R}$$

Our resulting $\oplus$R rule is a simple structural rule which generates a single subtyping constraint. Informally, when we encounter the program $\Delta \vdash x.succ \,;\, P :: (x : A)$, we know that A must be a $\oplus$-type, and that it must contain the label succ: both pieces of information are captured via the type constraint premise. We would then continue our algorithm on the latter premise with the freshly introduced intermediate type B. The other structural rules are analogous and omitted for brevity.

The above rule (and all structural rules) only create type constraints. In addition, some rules yield constraints of another sort: *arithmetic constraints*, which take the form of semantic entailment $\mathcal{V} \;;\; \mathcal{C} \vDash \phi$ for a proposition ϕ,

asserting that ϕ holds under $\mathcal{V}$ and $\mathcal{C}$. These rules are generated for the refinement type constructors that are responsible for exchanging proofs. In what follows, we highlight a few representative rules. A full list can be found in Appendix E.

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

The ?R rule is notable for introducing both an arithmetic constraint and a type constraint. The arithmetic constraint mirrors that of the corresponding typechecking rule, and simply states that ϕ must hold. Our type constraint, in tandem with the subtyping rules, asserts that A must have the structure $?\{\psi\}.A'$ for some ψ, but ψ need not be identical to ϕ: we instead must have $\mathcal{V} \,;\, \mathcal{C} \vDash \phi \to \psi$, i.e. ϕ is *stronger* than ψ.

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

If the channel of the same type appears in our context Δ, we instead apply the ?L rule. In right rules, we see that the freshly generated type appears to the left of the subtype constraint. In contrast, for all left rules, the fresh type falls on the right due to the duality of types. When we offer a channel $(z : C)$, we want that our declared type is *broader*, or a supertype, of whatever our program necessitates— e.g. we might declare $(z : \mathsf{nat})$ even if our process only sends the zero label. Conversely, if we consume a channel $(x : A)$, we need A to be *narrower*, or a subtype, of what our program dictates, i.e. the program should be able to handle any of A's behavior. In this instance where $A = ?\{\psi\}.A'$ for some ϕ, subtyping dictates that ψ should be at least as strong as ϕ, since if A "sends" a proof of ψ, we consequently receive a proof of ϕ.

$$\frac{\mathcal{V} \,;\, \mathcal{C} \vDash e \geq 0 \qquad \mathcal{V} \,;\, \mathcal{C} \Vdash \exists n.A' <: A \qquad \mathcal{V} \,;\, \mathcal{C} \,;\, \Delta \vdash P :: (x : A')}{\mathcal{V} \,;\, \mathcal{C} \wedge n = e \,;\, \Delta \vdash \mathsf{send}\ x\ \{e\}\ ;\ P :: (x : A)}\ \exists\mathrm{R}$$

The ∃R rule again contains both an arithmetic and a type constraint. The arithmetic constraint dictates that whatever expression we send should be a natural number, i.e., $e \geq 0$ to maintain the invariant that all witnesses are natural numbers. Of special note in this particular rule is that, since we are doing inference, we cannot perform a substitution $A'[e/n]$ as in typechecking, since we lack any information about where n might occur in the fresh name A'. As a tidy solution, we instead store the substitution as an equivalence in our governing constraint $\mathcal{C}$, such that any further continuations have access to the value of n without our needing to know ahead of time.

$$\frac{\begin{array}{c}(\overline{y_i' : B_i'})_{i \in I} \vdash f[\overline{n} \mid \phi] = P_f :: (x' : A') \in \Sigma \\ \mathcal{V} \,;\, \mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash A'[\overline{e}/\overline{n}] <: A \qquad (i \in I) \quad \mathcal{V} \,;\, \mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash B_i <: B_i'[\overline{e}/\overline{n}] \\ \mathcal{V} \,;\, \mathcal{C} \,;\, \Delta, (x : A[\overline{e}/\overline{n}]) \vdash Q :: (z : C)\end{array}}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Delta, (\overline{y_i : B_i})_{i \in I} \vdash x \leftarrow f[\overline{e}]\ \overline{y}\ ;\ Q :: (z : C)}\ \mathsf{def}$$

Our final example, the def rule for spawning processes, is noteworthy for interacting with the signature: our declaration of f has A' and B'_i as type variables, which are constrained both by the body P_f of f and here by our spawning of f. Our subtyping constraints are deceptively straightforward: for each channel $(y_i : B_i)$ consumed by our spawning of f, we compare it with the corresponding channel in the declaration $(y'_i : B'_i)$ of f, and we assert that the "real" type B_i is a subtype of the "expected" type B'_i. Conversely, we also constrain the offered type A' of f such that A' is a subtype of our fresh variable A.

In practice, the def rule is especially important in that it generates the majority of our arithmetic constraints, *despite not generating any as an explicit premise*. To solve type constraints, we apply our subtyping algorithm as presented in Section 4, which consequently generates arithmetic constraints. We elaborate further on the relationship between type and arithmetic constraints, as well as the constraint solving process as a whole, in the following section.

We conclude with a statement on the soundness of our inference algorithm:

Theorem 2. *Let $z \leftarrow f[\mathcal{V}]\ \overline{x_i} = P$ be the definition of a process f. For fresh type names A_i and C such that $(z : C)$ and $\Delta = (x_i : A_i)_{\forall i}$, and if $\mathcal{V} \,;\, \top \,;\, \Delta \vdash P :: (z : C)$, then P is a well-typed process.*

6 Implementation

We implemented the type inference algorithm on top of the Rast implementation [12], which already supports lexing, parsing, and typechecking for session types with arithmetic refinements. The inference engine consists of 2,069 lines of SML code, and integrates with the z3 SMT solver [16] to model arithmetic constraints. Our general approach begins by introducing placeholder declarations for each process in the file; for every definition `proc x ← f[`$\mathcal{V}$`] x1 x2 ... xn`, we introduce the declaration

`decl f[`$\mathcal{V}$`] : (x1 :` $A_1[\overline{e_1}]$`) (x2 :` $A_2[\overline{e_2}]$`) ... (xn :` $A_n[\overline{e_n}]$`) ⊢ (x :` $A[\overline{e}]$`)`

where each type $A_1, A_2, \ldots, A_n, A$ is a fresh type variable refined with arithmetic expressions $e_i(\mathcal{V})$ over the free variables $\mathcal{V}$ of the process. Our goal is to find a *type assignment* which maps each type variable to a concrete type name declared in the program, as well as an *expression assignment* which maps each e_i to a symbolic arithmetic expression. Theorem 2 dictates that these assignments result in a well-typed program, which is validated by running the Rast type checker on the inferred types. We follow a discussion of how this inference is carried out in two stages, the practical challenges we faced, and the optimizations we introduced to address them.

Two-Stage Inference When inferring the type of a process, we form constraints of two different sorts: type constraints and arithmetic constraints. Our constraint generation rules on processes create both sorts, and our subtyping algorithm, which we use to solve type constraints, may yield additional arithmetic constraints that must also be satisfied for inference to succeed.

At first, it might be tempting to solve both sorts of contraints in a unified pass. However, this becomes infeasible in practice. To see why, first note that arithmetic constraints cannot be solved *eagerly* due to the presence of process spawns. Consider a process `foo` that calls the `two` process twice, assigning the offered channel to `y` and `z` respectively.

```
decl two: . ⊢  (x: A[e()])
proc x ← foo = y ← two; z ← two; ...
```

Both calls to `two` in the `foo` process will generate constraints on type $\mathsf{A}[e()]$, which in turn will generate multiple distinct arithmetic constraints on e. Since e must be the same in *both* constraints, it is insufficient to demonstrate the satisfiability of these two constraints independently; a solver might otherwise invalidly assign different values to e. Thus, we must first collect all arithmetic constraints and then solve them all in one large batch.

However, taking this approach and otherwise following the rules as written leads to remarkable blowup in the number of constraints we generate and expensive solver calls we must make. The primary reason for this blowup is the presence of the $\mathtt{st}_\bot$ rule, which dictates that arbitrary, even mismatched, types can be subtypes under constraint $\mathcal{C}$ as long as $\mathcal{C}$ is false. Thus, whenever we might be tempted to return false when subtyping on a structural mismatch, such as $\mathcal{V}\,;\mathcal{C} \vDash A <: B$, we must instead add to our arithmetic constraint set the constraint $\mathcal{V}\,;\mathcal{C} \vDash \bot$. Thus, in this naïve approach, we are not able to rule out *any* type assignments without making a solver call, which becomes prohibitively expensive due to the exponential number of possible assignments.

We therefore chose to restrict the programmer from writing any dead code; that is, code where $\mathcal{V}\,;\mathcal{C} \vDash \bot$. This restriction has a number of consequences, but the essential one is that we can now rule out type assignments on structure without checking whether $\mathcal{C}$ is satisfiable. Taking advantage of this restriction, we split type inference into two stages: the first stage operates solely upon type structures, where all refinements are stripped from the program, and yields a list of structurally viable type assignments; the second stage reconsiders the refinements for a specific type assignment, collecting the resulting arithmetic constraints and passing them into the z3 SMT solver. This two-stage approach has the additional benefit of drawing a clean boundary between the decidable and undecidable fragments of our algorithm: in particular, type inference *without* refinements is decidable, and so we can return determinate error messages upon encountering a type mismatch.

6.1 First Stage of Inference

The first stage collects the constraints generated during type checking and subtyping, strips off the refinements, and proceeds to compute a type assignment. Since this stage is actually decidable, our algorithm is guaranteed to terminate and either find a satisfying type assignment, or report a type error. Thus, if the program is ill-typed due to structural typing (i.e., without refinements), then it is guaranteed to be detected.

Transitivity As described, our type constraint generation rules produce a fresh "intermediate" type variable for every process expression we encounter. For instance, revisiting the `two` process and recalling the $\oplus R$ inference rule from Section 5, we will generate the following constraints:

```
decl two : . ⊢ (x : A)
proc x ← two = x.succ;          % ⊕{succ : I0} <: A
               x.succ;          % ⊕{succ : I1} <: I0
               x.zero;          % ⊕{zero : I2} <: I1
               close x          % 1 <: I2
```

We generate fresh intermediate type variables (denoted by I_n) as a placeholder for the continuation type. If we do not distinguish between these intermediate variables and the "signature" variables (e.g., A) which we introduce in our declarations, we will generate a large number of constraints, putting a heavy load on our inference engine. To make matters worse, when these types have index refinements, each such intermediate variable will also create arithmetic constraints, which will make constraint solving infeasible for z3. This imposes a serious limitation on the scalability of inference.

We address this limitation with a crucial observation: *if $A <: B$, then so is $\oplus\{k : A\} <: \oplus\{k : B\}$.* To see this in action in the above example, note the second constraint: $\oplus\{\mathbf{succ} : I_1\} <: I_0$ from which we can deduce via transitivity that $\oplus\{\mathbf{succ} : \oplus\{\mathbf{succ} : I_1\}\} <: \oplus\{\mathbf{succ} : I_0\}$. Combining this with the first constraint via transitivity, we get $\oplus\{\mathbf{succ} : \oplus\{\mathbf{succ} : I_1\}\} <: \oplus\{\mathbf{succ} : I_0\} <: A$. Now, we can eliminate the intermediate variable I_0 to obtain that $\oplus\{\mathbf{succ} : \oplus\{\mathbf{succ} : I_1\}\} <: A$. Taking this all the way, we can eliminate *all* intermediate variables to obtain $\oplus\{\mathbf{succ} : \oplus\{\mathbf{succ} : \oplus\{\mathbf{zero} : \mathbf{1}\}\}\} <: A$.

This observation leads to the most significant optimization in our inference engine, which we call *transitivity*. It partly relies on transitivity of subtyping.

Theorem 3 (Transitivity). *If $A <: B$ and $B <: C$, then $A <: C$.*

Before we generate a type assignment, we perform a transitivity pass, which eliminates intermediate variables whenever possible. This dramatically reduces the number of type and arithmetic constraints, leading to significant speedups in inference, which we evaluate in Section 7. Not all intermediate variables will be removed by transitivity though—specifically, forwarding between channels which have both previously been communicated on will result in a type constraint of the form $\mathcal{V} ; \mathcal{C} \vDash I_0 <: I_1$. In such cases, we choose I_1 to perform any appropriate substitutions by transitivity, and we leave I_0 "free", that is, unconstrained—I_0 will only appear in the continuation type of another constraint.

Once the intermediate variables are eliminated, we assign, to each remaining type variable, an initial *search space* consisting of each defined type name in the file. Then, for each type variable, we substitute the type variable with one type from its search space, and run a *partial* version of subtyping on any relevant constraints. This modified subtyping algorithm stops eagerly upon encountering two (non-identical) type names, and returns either a reduced constraint or

simply true or false. Failed candidates are removeed from the search space while successful ones remain, and might get pruned away due to refinements.

Program Reconstruction The refinement-agnostic first stage also allows us to *reconstruct* each process definition according to its assigned type, adding back in assertions and assumptions. We take the approach of *forcing calculus* [15], eagerly assuming whenever possible and lazily asserting when absolutely necessary, guaranteeing that any assertions "see" all possible assumptions. Reconstruction also means that the programmer no longer has to manually write assertions or assumptions— a labor-intensive process which significantly bloats the source code, offsetting many potential practical benefits of type inference. We also reconstruct impossible branches, in line with our restriction that the programmer cannot themselves write unreachable code. Specifically, if a label in the type of a label-set is absent from a case statement, we manually insert it and say that it is impossible. This approach, in tandem with the two-stage solution, constitutes our solution to the aforementioned impossible-blowup problem: in the type solution stage, we assume that $\mathcal{C}$ is never false, which permits us to eliminate possible type assignments on structure.

6.2 Second Stage of Inference

In the second stage, we collecting the arithmetic constraints and attempt to find a satisfying assignment. Since our first stage has already provided us with structurally sound type assignments, we generally skip over structural constructs which are not refinement-related. However, we pay extra attention to the judgmental constructs forward and spawn, since they imply subtyping relations between their operands. For these, we run the complete subtyping algorithm and collect any resulting constraints. This generation substage yields a list of arithmetic constraints which are shipped off to z3 [16], which, if they are satisfiable, returns values which we can parse into an expression assignment. If z3 fails to find a solution, i.e., either because there is none, or the problem is undecidable, or due to a timeout, we say there is no solution and move on to the next type assignment if there exists one. Note that this stage actually attempts to solve an undecidable problem, hence our algorithm is incomplete. However, because of our numerous optimizations and heuristics, it works remarkably well in practice.

Polynomial Templates The bulk of our implementation effort went towards coercing z3 into cooperating with our style of constraints and scaling it to a wide variety of challenging benchmarks. We translate our arithmetic constraints of the form $\mathcal{V}\ ;\ \mathcal{C} \vDash \phi$ into the logical formulas $\forall \mathcal{V}. \mathcal{C} \Longrightarrow \phi$, where $\mathcal{C}$ and ϕ include expression variables $e_i(\mathcal{V})$ which we want to solve for. Our initial approach was to treat e_i's as *uninterpreted functions* and rely solely upon z3's built-in uninterpreted function solver. Unfortunately, this approach turned out to be insufficient for even the most trivial examples: even when z3 did not time out, it would return a function interpretation which was piecewise or otherwise inexpressible in our language for arithmetic expressions.

Our next optimization is based on the observation that we only allow refinement expressions that are *polynomials* over the quantified variables. Taking advantage of this, we represent each expression as a multivariate polynomial of degree d, and solve for all introduced coefficients. For instance, if $d = 2$ and we have free variables m, n for an expression e, we say $e(m, n) = c_0 + c_1 m + c_2 n + c_3 m^2 + c_4 mn + c_5 n^2$. By providing this template to the solver, we greatly restrict the number of possible interpretations which z3 must consider relative to purely uninterpreted functions, thereby reducing the solver burden significantly. Although this technique practically worked mostly for degree $d = 1$, in principle, we can support non-linear refinements through this approach.

Real Arithmetic Our refinements only allow natural numbers, and do not support real values in arithmetic expressions. However, we found that z3 stalls less frequently if we solve within a real-valued logic, as opposed to an integer-valued one. Of course, with this approach we risk z3 returning non-integer values when modeling our coefficients, which must ultimately be integers, but this occurrence turns out to be much less frequent than expected. Oftentimes, z3 simply returns exact integer values, even when the same constraints would time out for an integer logic. When z3 does return a floating-point value, we fall back to trying integer logic.

7 Evaluation

Methodology We evaluate the performance and efficacy of our inference algorithm on a variety of challenging benchmarks, that are known to be well-typed. Experiments were performed on a 2021 MacBook Pro with 16GB of RAM and an 8-core M1 Pro CPU. For each benchmark, we present the execution time of both stages: type constraint solving (Stage 1) and arithmetic constraint solving (Stage 2). We evaluate the efficacy of our three key optimizations as follows:

- *Polynomial Templates*: We run our inference engine with two strategies: **uif** stands for uninterpreted functions, while **poly** stands for using polynomial templates.
- *Real Arithmetic*: Our calls to z3 also have two modes: **real** stands for real arithmetic, while **int** stands for integers.
- *Transitivity*: We enable/disable transitivity which eliminates the intermediate type variables, so that we do not need to find a satisfying type assignment for them.

Our results are summarized in Tables 1-6, each table representing one benchmark. For all experiments, each call to z3 was set to timeout after 10 seconds, and each trial, which might make multiple such calls, was run for a maximum of 60 seconds in total. All numerical results are averaged across 10 trials, rounded to the nearest millisecond.

Our algorithm yields four possible results:

- **success**: inference finds a valid type and arithmetic assignment,

Strategy	**Arithmetic**	**Transitivity**	**Result**	**Stage 1** (ms)	**Stage 2** (ms)
poly	real	true	success	0.77	303.97
poly	real	false	success	11.55	297.94
poly	int	true	timeout	n/a	n/a
poly	int	false	timeout	n/a	n/a
uif	real	true	timeout	n/a	n/a
uif	real	false	timeout	n/a	n/a
uif	int	true	timeout	n/a	n/a
uif	int	false	timeout	n/a	n/a

Table 1: Unary Nats: 60 lines of code, 3 types, 7 processes

- **timeout**: either a call to z3 for a particular type assignment fails to return within the specified 10-second limit, or the overall algorithm execution time exceeds the 60-second limit,
- **inexpressible**: when z3, under **uif** mode, returns an expression assignment that cannot be expressed by our arithmetic language, e.g., expressions contain if-then-else constructs; or in the real arithmetic case, a floating-point number is returned that cannot be converted into an integer.

Technically, there is a fourth possibility as well: if the program is ill-typed, our inference algorithm returns **unsat**. However, for our evaluation, we chose to focus on well-typed benchmarks.

7.1 Results

For each benchmark, we briefly explain its contents, present the experimental results, and discuss any notable findings.

Unary Natural Numbers This benchmark primarily contains the refined natural number based on the types

$$\mathit{type}\ \mathsf{nat}[n] = \oplus\{\mathbf{zero} : ?\{n = 0\}.\mathbf{1},\ \mathbf{succ} : ?\{n > 0\}.\mathsf{nat}[n-1]\}$$
$$\mathit{type}\ \mathsf{natpair}[m][n] = \mathsf{nat}[m] \otimes \mathsf{nat}[n] \otimes \mathbf{1}$$

The second pair type is necessary for duplicating numbers due to linearity restrictions. The module includes 4 main processes and 3 helper and test processes: **add** for adding two numbers, **clone** for making a copy of a number, **consume** that consumes a `nat` to return **1**, and **double** which doubles a number using clone and add.

Table 1 describes the results of our inference algorithm along with the lines of code, and number of type and process definitions. First, we note that inference succeeds only when we use both real arithmetic and polynomial templates. Transitivity further produces an order of magnitude speedup in Stage 1, with only a minimal increase in Stage 2. As was expected, Stage 2 takes 1-2 orders of magnitude time more than Stage 1, since it involves calls to z3. The types inferred in each case were as expected, for example

```
decl add[m][n] : (x : nat[m]) (y : nat[n]) ⊢  (z : nat[m+n])
decl double[n] : (x : nat[n]) ⊢  (y : nat[2*n])
```

Strategy	**Arithmetic**	**Transitivity**	**Result**	**Stage 1** (ms)	**Stage 2** (ms)
poly	real	true	success	0.47	118.89
poly	real	false	success	4.03	118.23
poly	int	true	timeout	n/a	n/a
poly	int	false	timeout	n/a	n/a
uif	real	true	inexpressible	0.45	22.46
uif	real	false	inexpressible	4.12	22.00
uif	int	true	inexpressible	0.48	22.69
uif	int	false	inexpressible	4.37	19.74

Table 2: Direct Nats: 47 lines of code, 3 types, 7 processes

Direct Natural Numbers An alternative representation for natural numbers is directly through refinements, via the type $\mathsf{nat} = \exists n.\mathbf{1}$. Instead of sending 'n' **succ** messages, this type produces a single natural number as a refinement and terminates. This benchmark contains all the same programs as unary naturals, but with this modified representation.

Table 2 describes the results of our experiments. Successful satisfying assignments again only occur for the specific combination of polynomial templates along with real arithmetic, regardless of transitivity. With uninterpreted functions, inference does find a solution but heavily relies upon the aforementioned if-then-else constructs to effectively case-analyze the constraints instead of finding linear solutions. For instance, if we use $\mathsf{add}[m][n]$ only as $\mathsf{add}[2][3]$ and $\mathsf{add}[4][5]$, then instead of outputting the expected expression $m + n$, z3 outputs $\mathsf{if}\ m = 2\ \mathsf{then}\ 5\ \mathsf{else}\ 9$.

Binary Numbers A more efficient representation of natural numbers is in their binary form. Instead of sending n **succ** messages, a type can send $\lceil \log_2 n \rceil$ 0's and 1's. This representation is captured in this benchark which introduce a type $\mathsf{bin}[n]$ which can send labels $b0$, $b1$, or e, representing 0, 1, and termination, respectively. We encode them in *little endian* format, i.e. the least significant bit is sent first, which makes implementations more convenient and types more intuitive. The refinement indexes their value.

$$\begin{aligned} \mathit{type}\ \mathsf{bin}[n] = \oplus\{&\mathbf{b0} : ?\{n > 0\}.\exists k.\ ?\{n = 2k\}.\mathsf{bin}[k], \\ &\mathbf{b1} : ?\{n > 0\}.\exists k.\ ?\{n = 2k+1\}.\mathsf{bin}[k], \\ &\mathbf{e} : ?\{n = 0\}.\mathbf{1}\} \end{aligned}$$

The type of $\mathsf{bin}[n]$ signifies that the provider can either send labels **b0**, **b1**, or **e**. In the case of **b0**, the provider provides a proof that indeed $n > 0$ and produces a new number k such that $n = 2k$, meaning that n is even. Analogously for **b1**, the provider asserts that $n > 0$ and is odd by producing k such that $n = 2k + 1$. Lastly, in the case of **e**, the provider proves that $n = 0$ and terminates. For this benchmark, we implement **successor** and **double** with a few helper processes.

The results are presented in Table 3. For this benchmark, the type constraints turn out to be more challenging than the arithmetic constraints primarily due to the complexity of the bin type. Hence, we observe a significant impact from the transitivity optimization— without transitivity, Stage 1 takes twice as long as Stage 2, but with transitivity, Stage 1 is faster by multiple orders of magnitude.

Strategy	Arithmetic	Transitivity	Result	Stage 1 (ms)	Stage 2 (ms)
poly	real	true	success	0.279	52.17
poly	real	false	success	88.72	49.06
poly	int	true	timeout	n/a	n/a
poly	int	false	timeout	n/a	n/a
uif	real	true	timeout	n/a	n/a
uif	real	false	timeout	n/a	n/a
uif	int	true	inexpressible	0.22	24.34
uif	int	false	inexpressible	87.95	19.55

Table 3: Binary Nats: 56 lines of code, 2 types, 4 processes

Strategy	Arithmetic	Transitivity	Result	Stage 1 (ms)	Stage 2 (ms)
poly	real	true	success	1.19	808.51
poly	real	false	success	144.04	806.00
poly	int	true	timeout	n/a	n/a
poly	int	false	timeout	n/a	n/a
uif	real	true	timeout	n/a	n/a
uif	real	false	timeout	n/a	n/a
uif	int	true	timeout	n/a	n/a
uif	int	false	timeout	n/a	n/a

Table 4: Lists: 108 lines of code, 4 types, 12 processes

Like previous examples, we need both polynomial templates and real arithmetic for scalable inference; uninterpreted functions produces inexpressible results. The types inferred are as expected and successfully typecheck:

```
decl successor[n] : (x : bin[n]) ⊢  (y : bin[n+1])
decl double[n] : (x : bin[n]) ⊢  (y : bin[2*n])
```

Lists This is another practically important data structure, particularly in the context of session types to store data. Since we currently do not support inference of polymorphic session types, we use Lisp-style lists of natural numbers, where the list is refined by its length. The type $\mathsf{list}[n]$ allows either sending the **cons** label if $n > 0$, transitioning to $\mathsf{list}[n-1]$, or the **nil** label if $n = 0$.

$$\begin{aligned} \mathit{type}\ \mathsf{list}[n] = \oplus\{&\mathbf{cons} : ?\{n > 0\}.\,\mathsf{nat} \otimes \mathsf{list}[n-1], \\ &\mathbf{nil} : ?\{n = 0\}.\mathbf{1}\} \end{aligned}$$

To focus on inference of lists, we use unrefined natural numbers (nat) as the type of elements stored inside the list. We implement 2 important (with some helper) list processes: **append**, which concatenates two lists, and **split** which splits a list in half using two mutually recursive processes that operate on even-length and odd-length lists. We also include a $\mathsf{listpair}[m][n]$ type, analogous to the aforementioned $\mathsf{natpair}$ type, to realize **split**.

Table 4 describes the results: we successfully find a solution only when using real arithmetic and polynomial templates. However, unlike unary nats, we now see a significant benefit from transitivity, which supports our hypothesis that more complex types— which beget longer, more complex programs— benefit much more from transitivity. As expected, the types inferred are as follows:

```
decl append[m][n] : (x : list[m]) (y : list[n]) ⊢  (z : list[m+n])
decl split_even[n] : (x : list[2*n]) ⊢  (y : listpair[n][n])
decl split_odd[n] : (x : list[2*n+1]) ⊢  (y : listpair[n+1][n])
```

Linear λ-calculus Our most challenging benchmark is an implementation of a linear λ-calculus of expressions on top of session types. Due to its complexity, we present this example via 2 benchmarks: the first one does not contain any refinements while the second one indexes types with their size. Expressions are represented using the following type:

$$\mathit{type}\ \mathsf{exp} = \oplus\{\mathbf{lam} : \mathsf{exp} \multimap \mathsf{exp},\ \mathbf{app} : \mathsf{exp} \otimes \mathsf{exp}\}$$

Subtyping plays an important role here because the type of values written as

$$\mathit{type}\ \mathsf{val} = \oplus\{\mathbf{lam} : \mathsf{exp} \multimap \mathsf{exp}\}$$

is a *subtype* of exp, i.e., the type of expressions: all values are also expressions. We implement processes to evaluate and normalize expressions, and we include a couple of simple programs (id, swap) in the calculus.

Table 5 contains results of the inference engine. Immediately evident is that transitivity here makes a significant difference: only those trials with transitivity enabled succeed, while the others time out. This is because of the presence of both val and exp types. In previous examples, type assignments could immediately rule out a large swath of the search space because there was only one type that would structurally fit a type variable, but here exp and val are at times interchangeable. Stage 1 involves much more work, which is exacerbated by the presence of intermediate type variables if we forego transitivity. As an example, we end up with 1024 valid type assignments after Stage 1. Of course, since we lack refinements, z3 has no constraints to solve, and so the **strategy** and **arithmetic** optimizations, which only relate to z3, have no observable effect. Since Stage 2 concerns arithmetic refinements, of which there are none in this benchmark, the calls to z3 finish in ∼20ms.

Due to subtyping, there are many possible combinations of types that our algorithm could correctly infer. Our particular implementation infers the following key declarations, prioritizing the narrower type val over its supertype exp:

```
decl apply : (e1 : val) (e2 : val) ⊢  (e : exp)
decl eval : (e : exp) ⊢  (v : val)
decl norm : (e : exp) ⊢  (n : exp)
```

Sized Linear λ-calculus Finally, our most challenging benchmark is extended with a refinement representing the size of the term, as follows:

$$\begin{aligned}\mathsf{type}\ \mathsf{exp}[n] = \oplus\{&\mathbf{lam} : ?\{n > 0\}.\forall n.\mathsf{exp}[n] \multimap \mathsf{exp}[n + n' - 1],\\ &\mathbf{app} : \forall n_1.\forall n_2.\ ?\{n = n_1 + n_2 + 1\}.\mathsf{exp}[n_1] \otimes \mathsf{exp}[n_2]\}\end{aligned}$$

We adapt the aforementioned subtype val to $\mathsf{val}[n]$. Finally, we introduce a new type, $\mathsf{boundedVal}[n] = \exists k.\ ?\{k \leq n\}.\mathsf{val}[k]$, as the new return type of our eval

Strategy	**Arithmetic**	**Transitivity**	**Result**	**Stage 1** (ms)	**Stage 2** (ms)
poly	real	true	success	38.33	23.33
poly	real	false	timeout	n/a	n/a
poly	int	true	success	39.85	20.64
poly	int	false	timeout	n/a	n/a
uif	real	true	success	38.57	21.08
uif	real	false	timeout	n/a	n/a
uif	int	true	success	39.83	20.48
uif	int	false	timeout	n/a	n/a

Table 5: Linear λ-calculus: 72 lines of code, 3 types, 8 processes

Strategy	**Arithmetic**	**Transitivity**	**Result**	**Stage 1** (ms)	**Stage 2** (ms)
poly	real	true	success	16.91	64.35
poly	real	false	timeout	n/a	n/a
poly	int	true	success	17.33	11217.74
poly	int	false	timeout	n/a	n/a
uif	real	true	timeout	n/a	n/a
uif	real	false	timeout	n/a	n/a
uif	int	true	timeout	n/a	n/a
uif	int	false	timeout	n/a	n/a

Table 6: Sized Linear λ-calculus: 95 lines of code, 4 types, 10 processes

process: although we cannot evaluate the exact size of an evaluated term, we can place an upper bound on it.

This example reflects the full power of our optimizations: like linear λ-calculus, disabling transitivity always leads to a timeout in Stage 1. Since we also have arithmetic constraints to solve for this benchmark, our polynomial template strategy proves crucial in preventing a z3 timeout. We successfully find solutions for both integer and real arithmetic with both other optimizations, but integer arithmetic takes *several orders of magnitude* longer to solve. In other words, all three of our optimizations are necessary in order to perform type inference on this benchmark in reasonable time.

Again, our implementation prioritizes the narrower type $\mathsf{val}[n]$ over $\mathsf{exp}[n]$ and returns the following key declarations:

```
decl apply[m][n] : (e1 : val[m]) (e2 : val[n]) ⊢  (e : exp[m+n+1])
decl eval : (e : exp[n]) ⊢  (v : boundedVal[n])
```

Noteworthy here is that, via the inferred type of **eval**, our algorithm deduces that the evaluation of an expression with size n yields a value whose size is no greater than n.

8 Related Works

Techniques for Inferring Type Refinements Numerous techniques have been proposed for inferring refinement types for functional programs. The earliest works on refinement type inference was in the context of ML [18] where the types were refined by a *finite* programmer-specified lattice. Finiteness was crucial for decidability of type inference, and type refinement was performed

iteratively until reaching a fixed point. Our type system is considerably more expressive with a possibly infinite set of arithmetic refinements, which makes inference undecidable. Nonetheless, they introduced this technique of inference in stages: first inferring base types followed by inferring type refinements. This style was later adopted by Liquid Types [38] where inference was reduced to Hindley-Milner type inference for base types [9], followed by liquid constraint generation and constraint solving. Type inference in this setting is decidable because they adopt a conservative but decidable notion of subtyping, where subtyping of arbitrary dependent types is reduced to a set of implication checks over base types. In contrast, our notion of subtyping is general and therefore, undecidable.

Inference of type refinements has also been carried out using abstract interpretation [32] by reducing it to computing invariants of simple, first-order imperative programs. The FUSION algorithm [8] reduces inference to finding a satisfying assignment for (nested) Horn Clause Constraints in Negation Normal Form (NNF). Hashimoto and Unno [25] reduce inference to a type optimization problem which is further reduced to a Horn constraint optimization problem. This technique can infer maximally preferred refinement types based on a user-specified preference. Pavlinovic et al. [36] propose a more general type system that is parametric both with the choice of the abstract refinement domain and context-sensitivity of control flow information. More recently, learning-based techniques using randomized testing [48] and LLMs [39] have also been proposed for refinement inference.

The most distinguishing aspect of our style of refinements is that session types follow *structural* typing, whereas the aforementioned systems follow *nominal* typing. With nominal typing where type equality and subtyping for base types depends on the name, inferring the base type for all intermediate sub-expressions becomes decidable and reduces to standard Hindley-Milner inference. On the other hand, prior work [13] shows that even though linear arithmetic as well as type equality (and therefore subtyping) for basic session types are decidable, the combination makes typing undecidable. More concretely, there is no notion of a "base type" with session types and therefore, our subtyping rules cannot be separated into subtyping for base types and refinements; they mix together naturally. As a result, even our first stage of inference differs from these works and eliminates the intermediate types instead of inferring them. More closely related to our work is the recent work on structural type refinements [3] which builds on algebraic subtyping to combine properties of nominal and structural type systems. The main difference between the two works is the design of the refinement layer. While we refine types with arithmetic expressions, they use polymorphic variants, thus supporting type application and union. Due to this, we rely on a solver for inference while they use constraint-based type inference.

Session Type Inference Also related to our system are techniques for session type inference which originated from observing communication primitives in π-calculus [23, 37] implemented in OCaml [30] and Haskell [31]. Session types have also been inferred with control flow information [7] and in a calculus of services and sessions [34]. Almeida et al. [2] propose an algorithm for inference

of FreeST [1] which is an implementation of context-free session types [42]. This technique builds on Quick Look [40] to enable inference of type annotations in polymorphic applications. Although FreeST allows non-regular protocols, arithmetic refinements are more general and can encode stronger properties based on sizes and values that can even be non-linear. Therefore, the inference algorithms are also quite different and FreeST does not require solving arithmetic constraints. To that end, none of these works support the DML-style [47] of refinements like our system.

Session Subtyping The concept of subtyping for session types has its roots in seminal work by Gay and Hole [20], which introduces both the formal notion of a type simulation and a practical subtyping algorithm for basic session types in the π-calculus. Our work builds upon theirs by introducing refinements to the language. Crucially, their algorithm is both sound and complete, but a complete subtyping algorithm is impossible in the presence of refinements; thus, our algorithm is sound, but not complete. Session subtyping has been further explored in a variety of more specific contexts: for instance, Horne and Padovani [29] develop and compare subtyping relation in both the isorecursive and equirecursive settings, and others [21, 22, 33] propose subtyping for multiparty session types. Mostrous and Yoshida [35] propose subtyping for asynchronous session types in a higher-order π-calculus, extending the idea of a type simulation into an *asynchronous* type simulation. Bravetti et al. [4] show that asynchronous subtyping is undecidable in the presence of recursion. In contrast to these works, ours is the first to explore session subtyping alongside a refinement system. Our notion of refinements is adopted from Das and Pfenning [15], but their work was restricted to type equality and did not provide a type inference algorithm.

9 Conclusion

This paper presents a type inference algorithm for structural session types with arithmetic refinements. We develop a theoretical treatment of subtyping, introduce formal inference rules for constraint generation, and implement our algorithm in the Rast language. In addition, we detail the practical optimizations necessary to solve our constraints reliably, and we demonstrate the benefits of these optimizations by evaluating the performance of our implementation on a number of examples.

The primary future direction we wish to pursue is the extension of our algorithm to other constructs in the Rast language. Specifically, Rast supports both *temporal* and *resource-aware* types [12], both of which we expect to involve relatively natural extensions of our current algorithm. Recent developments in polymorphic [17] and probabilistic [19] session types also comprise interesting new domains into which evolutions of our work might extend.

Bibliography

[1] Almeida, B., Mordido, A., Vasconcelos, V.: Freest: Context-free session types in a functional language. Electronic Proceedings in Theoretical Computer Science **291**, 12–23 (03 2019), https://doi.org/10.4204/EPTCS.291.2

[2] Almeida, B., Mordido, A., Vasconcelos, V.T.: Local type inference for context-free session types. In: Derakhshan, F., Hoffmann, J. (eds.) Proceedings 16th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, Hamilton, Canada, 4th May 2025, Electronic Proceedings in Theoretical Computer Science, vol. 420, pp. 1–11, Open Publishing Association (2025), https://doi.org/10.4204/EPTCS.420.1

[3] Binder, D., Skupin, I., Läwen, D., Ostermann, K.: Structural refinement types. In: Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development, p. 15–27, TyDe 2022, Association for Computing Machinery, New York, NY, USA (2022), ISBN 9781450394390, https://doi.org/10.1145/3546196.3550163, URL https://doi.org/10.1145/3546196.3550163

[4] Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of asynchronous session subtyping. Information and Computation **256**, 300–320 (2017), ISSN 0890-5401, https://doi.org/https://doi.org/10.1016/j.ic.2017.07.010, URL https://www.sciencedirect.com/science/article/pii/S0890540117301190

[5] Caires, L., Pfenning, F.: Session Types as Intuitionistic Linear Propositions. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, vol. 6269, pp. 222–236, Springer Berlin Heidelberg, Berlin, Heidelberg (2010), ISBN 978-3-642-15374-7 978-3-642-15375-4, https://doi.org/10.1007/978-3-642-15375-4_16, URL http://link.springer.com/10.1007/978-3-642-15375-4_16, series Title: Lecture Notes in Computer Science

[6] Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **760** (11 2014)

[7] Collingbourne, P., Kelly, P.H.J.: Inference of session types from control flow. Electron. Notes Theor. Comput. Sci. **238**(6), 15–40 (Jun 2010), ISSN 1571-0661, https://doi.org/10.1016/j.entcs.2010.06.003, URL https://doi.org/10.1016/j.entcs.2010.06.003

[8] Cosman, B., Jhala, R.: Local refinement typing. Proc. ACM Program. Lang. **1**(ICFP) (Aug 2017), https://doi.org/10.1145/3110270, URL https://doi.org/10.1145/3110270

[9] Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 207–212, POPL '82, Association for Computing Machinery, New York, NY, USA (1982), ISBN

0897910656, https://doi.org/10.1145/582153.582176, URL https://doi.org/10.1145/582153.582176

[10] Das, A., Hoffmann, J., Pfenning, F.: Parallel complexity analysis with temporal session types. Proc. ACM Program. Lang. **2**(ICFP) (Jul 2018), https://doi.org/10.1145/3236786, URL https://doi.org/10.1145/3236786

[11] Das, A., Hoffmann, J., Pfenning, F.: Work analysis with resource-aware session types. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, p. 305–314, LICS '18, Association for Computing Machinery, New York, NY, USA (2018), ISBN 9781450355834, https://doi.org/10.1145/3209108.3209146, URL https://doi.org/10.1145/3209108.3209146

[12] Das, A., Pfenning, F.: Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description). In: Ariola, Z.M. (ed.) 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), Leibniz International Proceedings in Informatics (LIPIcs), vol. 167, pp. 33:1–33:17, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020), ISBN 978-3-95977-155-9, ISSN 1868-8969, https://doi.org/10.4230/LIPIcs.FSCD.2020.33, URL https://drops.dagstuhl.de/opus/volltexte/2020/12355

[13] Das, A., Pfenning, F.: Session Types with Arithmetic Refinements pp. 18 pages, 512384 bytes (2020), ISSN 1868-8969, https://doi.org/10.4230/LIPICS.CONCUR.2020.13, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2020.13

[14] Das, A., Pfenning, F.: Session types with arithmetic refinements (2020), URL https://arxiv.org/abs/2005.05970

[15] Das, A., Pfenning, F.: Verified Linear Session-Typed Concurrent Programming. In: Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming, pp. 1–15, ACM, Bologna Italy (Sep 2020), ISBN 978-1-4503-8821-4, https://doi.org/10.1145/3414080.3414087, URL https://dl.acm.org/doi/10.1145/3414080.3414087

[16] De Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, p. 337–340, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), ISBN 3540787992

[17] DeYoung, H., Mordido, A., Pfenning, F., Das, A.: Parametric subtyping for structural parametric polymorphism. Proc. ACM Program. Lang. **8**(POPL) (Jan 2024), https://doi.org/10.1145/3632932, URL https://doi.org/10.1145/3632932

[18] Freeman, T., Pfenning, F.: Refinement types for ml. SIGPLAN Not. **26**(6), 268–277 (may 1991), ISSN 0362-1340, https://doi.org/10.1145/113446.113468, URL https://doi.org/10.1145/113446.113468

[19] Fu, Q., Das, A., Gaboardi, M.: Probabilistic refinement session types (companion report) (2025), https://doi.org/10.5281/zenodo.15185261

[20] Gay, S., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (Nov 2005), ISSN 0001-5903, 1432-0525, https://doi.org/10.1007/s00236-005-0177-z, URL http://link.springer.com/10.1007/s00236-005-0177-z

[21] Ghilezan, S., Jakšić, S., Pantović, J., Scalas, A., Yoshida, N.: Precise subtyping for synchronous multiparty sessions. Journal of Logical and Algebraic Methods in Programming **104**, 127–173 (2019), ISSN 2352-2208, https://doi.org/https://doi.org/10.1016/j.jlamp.2018.12.002, URL https://www.sciencedirect.com/science/article/pii/S2352220817302237

[22] Ghilezan, S., Pantović, J., Prokić, I., Scalas, A., Yoshida, N.: Precise subtyping for asynchronous multiparty sessions. ACM Trans. Comput. Logic **24**(2) (Nov 2023), ISSN 1529-3785, https://doi.org/10.1145/3568422, URL https://doi.org/10.1145/3568422

[23] Graversen, E.F., Harbo, J.B., Hüttel, H., Bjerregaard, M.O., Poulsen, N.S., Wahl, S.: Type inference for session types in the *pi*-calculus. In: Hildebrandt, T., Ravara, A., van der Werf, J.M., Weidlich, M. (eds.) Web Services, Formal Methods, and Behavioral Types, pp. 103–121, Springer International Publishing, Cham (2016), ISBN 978-3-319-33612-1

[24] Griffith, D., Gunter, E.L.: Liquidpi: Inferrable dependent session types. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods, pp. 185–197, Springer Berlin Heidelberg, Berlin, Heidelberg (2013), ISBN 978-3-642-38088-4

[25] Hashimoto, K., Unno, H.: Refinement type inference via horn constraint optimization. In: Blazy, S., Jensen, T. (eds.) Static Analysis, pp. 199–216, Springer Berlin Heidelberg, Berlin, Heidelberg (2015), ISBN 978-3-662-48288-9

[26] Honda, K.: Types for dyadic interaction. In: Goos, G., Hartmanis, J., Best, E. (eds.) CONCUR'93, vol. 715, pp. 509–523, Springer Berlin Heidelberg, Berlin, Heidelberg (1993), ISBN 978-3-540-57208-4 978-3-540-47968-0, https://doi.org/10.1007/3-540-57208-2_35, URL http://link.springer.com/10.1007/3-540-57208-2_35, series Title: Lecture Notes in Computer Science

[27] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems, pp. 122–138, Springer Berlin Heidelberg, Berlin, Heidelberg (1998), ISBN 978-3-540-69722-0

[28] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 273–284, POPL '08, ACM, New York, NY, USA (2008), ISBN 978-1-59593-689-9, https://doi.org/10.1145/1328438.1328472, URL http://doi.acm.org/10.1145/1328438.1328472

[29] Horne, R., Padovani, L.: A logical account of subtyping for session types. Journal of Logical and Algebraic Methods in Programming **141**, 100986 (2024), ISSN 2352-2208,

https://doi.org/https://doi.org/10.1016/j.jlamp.2024.100986, URL https://www.sciencedirect.com/science/article/pii/S2352220824000403

[30] Imai, K., Lange, J., Neykova, R.: Kmclib: Automated inference and verification of session types from ocaml programs. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 379–386, Springer International Publishing, Cham (2022), ISBN 978-3-030-99524-9

[31] Imai, K., Yuen, S., Agusa, K.: Session type inference in haskell. In: Honda, K., Mycroft, A. (eds.) Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, Paphos, Cyprus, 21st March 2010, Electronic Proceedings in Theoretical Computer Science, vol. 69, pp. 74–91, Open Publishing Association (2011), https://doi.org/10.4204/EPTCS.69.6

[32] Jhala, R., Majumdar, R., Rybalchenko, A.: Hmc: Verifying functional programs using abstract interpreters. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, pp. 470–485, Springer Berlin Heidelberg, Berlin, Heidelberg (2011), ISBN 978-3-642-22110-1

[33] Li, E., Stutz, F., Wies, T.: Deciding subtyping for asynchronous multiparty sessions. In: Weirich, S. (ed.) Programming Languages and Systems, pp. 176–205, Springer Nature Switzerland, Cham (2024)

[34] Mezzina, L.G.: How to infer finite session types in a calculus of services and sessions. In: Lea, D., Zavattaro, G. (eds.) Coordination Models and Languages, pp. 216–231, Springer Berlin Heidelberg, Berlin, Heidelberg (2008), ISBN 978-3-540-68265-3

[35] Mostrous, D., Yoshida, N.: Session typing and asynchronous subtyping for the higher-order π-calculus. Information and Computation **241**, 227–263 (2015), ISSN 0890-5401, https://doi.org/https://doi.org/10.1016/j.ic.2015.02.002, URL https://www.sciencedirect.com/science/article/pii/S0890540115000139

[36] Pavlinovic, Z., Su, Y., Wies, T.: Data flow refinement type inference. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021), https://doi.org/10.1145/3434300, URL https://doi.org/10.1145/3434300

[37] Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell, pp. 25–36, Haskell '08, Association for Computing Machinery, New York, NY, USA (2008), ISBN 9781605580647, https://doi.org/10.1145/1411286.1411290, URL https://doi.org/10.1145/1411286.1411290

[38] Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 159–169, PLDI '08, Association for Computing Machinery, New York, NY, USA (2008), ISBN 9781595938602, https://doi.org/10.1145/1375581.1375602, URL https://doi.org/10.1145/1375581.1375602

[39] Sakkas, G., Sahu, P., Ong, K., Jhala, R.: Neurosymbolic Modular Refinement Type Inference . In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), pp. 627–627, IEEE

Computer Society, Los Alamitos, CA, USA (May 2025), ISSN 1558-1225, https://doi.org/10.1109/ICSE55347.2025.00090, URL https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00090

[40] Serrano, A., Hage, J., Peyton Jones, S., Vytiniotis, D.: A quick look at impredicativity. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020), https://doi.org/10.1145/3408971, URL https://doi.org/10.1145/3408971

[41] Silva, G., Mordido, A., Vasconcelos, V.T.: Subtyping Context-Free Session Types. In: Pérez, G.A., Raskin, J.F. (eds.) 34th International Conference on Concurrency Theory (CONCUR 2023), Leibniz International Proceedings in Informatics (LIPIcs), vol. 279, pp. 11:1–11:19, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023), ISBN 978-3-95977-299-0, ISSN 1868-8969, https://doi.org/10.4230/LIPIcs.CONCUR.2023.11, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2023.11

[42] Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, p. 462–475, ICFP 2016, Association for Computing Machinery, New York, NY, USA (2016), ISBN 9781450342193, https://doi.org/10.1145/2951913.2951926, URL https://doi.org/10.1145/2951913.2951926

[43] Thiemann, P., Vasconcelos, V.T.: Label-dependent session types. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019), https://doi.org/10.1145/3371135, URL https://doi.org/10.1145/3371135

[44] Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming, p. 161–172, PPDP '11, Association for Computing Machinery, New York, NY, USA (2011), ISBN 9781450307765, https://doi.org/10.1145/2003476.2003499, URL https://doi.org/10.1145/2003476.2003499

[45] Toninho, B., Caires, L., Pfenning, F.: A decade of dependent session types. In: Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming, PPDP '21, Association for Computing Machinery, New York, NY, USA (2021), ISBN 9781450386890, https://doi.org/10.1145/3479394.3479398, URL https://doi.org/10.1145/3479394.3479398

[46] Wu, H., Xi, H.: Dependent session types. CoRR **abs/1704.07004** (2017), URL http://arxiv.org/abs/1704.07004

[47] Xi, H., Pfenning, F.: Dependent types in practical programming. In: Aiken, A. (ed.) Conference Record of the 26th Symposium on Principles of Programming Languages (POPL 1999), pp. 214–227, ACM Press, San Antonio, Texas, USA (Jan 1999)

[48] Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 400–411, ICFP 2015, Association for Computing Machinery, New York, NY, USA (2015), ISBN 9781450336697,

https://doi.org/10.1145/2784731.2784766, URL https://doi.org/10.1145/2784731.2784766

A Session Type Examples

Closing and waiting. The process `example1` has the following declaration:

```
decl example1 : (x : 1) (y : 1) ⊢ (z : 1)
```

From its type, we deduce that it must wait for x and y to close, and then it closes the channel it offers, z. A suitable definition would be:

```
proc z ← example1 x y = wait x; wait y; close z
```

Alternatively, we could wait on y before x.

Sending and receiving labels. We declare the process `example1` as follows:

```
decl negate : (x: ⊕{true : 1, false : 1}) ⊢ (y: ⊕{true : 1, false : 1})
```

This process would receive a label from x, either `true` or `false`, and then sends either `true` or `false` on y, depending on a case-analysis of what x sent. We define `negate` as:

```
proc y ← negate x = case x (
  true  ⇒ y.false; wait y; close x
  | false ⇒ y.true; wait y; close x
)
```

The types do not tell us *which* label `negate` must send when— in the given definition, we send the opposite label, but we could also just always send **false**; both programs would share the same declaration. However, the type system enforces that we cannot send on x nor receive on y a label outside of the label set $\{\mathbf{true}, \mathbf{false}\}$.

Sending and receiving channels. An equivalent of function composition in our language could be written as the process `comp` below:

```
decl comp : (x: A ⊸ B), (y: B ⊸ C) ⊢ (z: A ⊸ C)
proc z ← comp x y =
  a ← recv z;
  send x a;
  send y x;
  z ↔ y
```

We would first receive a channel a of type A from our provided channel z. We then send a to x, which changes the type of x into B. Now we send x to y to get $(y : C)$. We are left with $(y : C) \vdash (z : C)$, at which point we *forward* between z and y.

Type names and recursive types. Suppose our signature contains the type declaration $\mathsf{nat} = \oplus\{\mathbf{zero} : \mathbf{1}, \mathbf{succ} : \mathsf{nat}\}$ Consider the following process:

```
decl consume : (x: nat) ⊢ (y: 1)
proc y ← consume x = case x (
  zero ⇒ y ↔ x
  | succ ⇒ y ← consume x
)
```

Due to linearity, we can never disregard types in our context; if we wish to throw them away, we must instead transform them into a type which we know how to close. The process `consume` does just that, taking a `nat` and turning it into **1**, which the client can then easily wait on. We implement this process by first case-analyzing the input: if x sends **zero**, we can simply forward, but if x sends **succ**, it recurses back to `nat`, so we *spawn* `consume` again with x as input. By repeated respawns of `consume`, we "consume" all the labels x sends until it turns into **1**.

Assertion and assumption. Let our `nat` type now refer to the declaration in a fresh Σ that $\mathsf{nat}[n] = \oplus\{\mathbf{zero} : ?\{n = 0\}.\mathbf{1},\ \mathbf{succ} : ?\{n > 0\}.\mathsf{nat}[n-1]\}$. Suppose we have a `double` process which doubles the value of the input, declared as follows:

```
decl double[n] : (x: nat[n]) ⊢ (y: nat[2*n])
proc y ← double[n] x = case x (
  zero ⇒ assume x {n=0};
    y.zero; assert y {2*n=0};
    y ↔ x
  | succ ⇒ assume x {n>0};
      y.succ; assert y {2*n>0};
      y.succ; assert y {(2*n)-1>0};
      y ← double[n-1] x
)
```

Note the refinement parameter n in the process— when we spawn `double`, we now specify an expression in place of n, which in this case corresponds to the value of the input channel x. When we receive a **zero** label from x, we also receive a proof that $n = 0$; we use this fact to assert that $2 * n = 0$ on y after sending **zero** on y. When we instead receive **succ** on x, we get that $n > 0$; our goal is to send 2 **succ** messages on y. When we send the first, we use our assumption to assert that $2 * n > 0$, and the type of y recurses to $\mathsf{nat}[(2 * n) - 1]$ by our type definition. When we send a second **succ** label, we now must assert that $(2 * n) - 1 > 0$, which still follows from our assertion (and the fact that we work only in integers). We now have $(x : \mathsf{nat}[n - 1]) \vdash (y : \mathsf{nat}[(2 * n) - 2]$, and we conclude by respawning `double` with refinement $n - 1$ instead of n.

Witnesses. Consider a type for binary numbers that sends bits `b0` and `b1`, and terminates with `e`. If we want to express its value with a refinement, a natural thought would be to say:

$$\begin{aligned} \mathit{type}\ \mathsf{bin}[n] = \oplus\{&\mathbf{b0} : ?\{n > 0\}.\mathsf{bin}[k/2], \\ &\mathbf{b1} : ?\{n > 0\}.\mathsf{bin}[(k-1)/2], \\ &\mathbf{e} : ?\{n = 0\}.\mathbf{1}\} \end{aligned}$$

However, our arithmetic language lacks a division operator. Instead, we would use quantifiers to capture the same idea as follows:

$$\begin{aligned} \mathit{type}\ \mathsf{bin}[n] = \oplus\{&\mathbf{b0} : ?\{n > 0\}.\exists k.\ ?\{n = 2k\}.\mathsf{bin}[k], \\ &\mathbf{b1} : ?\{n > 0\}.\exists k.\ ?\{n = 2k+1\}.\mathsf{bin}[k], \\ &\mathbf{e} : ?\{n = 0\}.\mathbf{1}\} \end{aligned}$$

B Validity

We introduce three validity judgments, for signatures ($\vdash \Sigma$ valid), declarations ($\vdash_\Sigma \Sigma'$ valid), and types ($\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A$ valid).

$$\frac{\vdash_\Sigma \Sigma \text{ valid}}{\vdash \Sigma \text{ valid}}\ \Sigma V \qquad \frac{}{\vdash_\Sigma (\cdot) \text{ valid}}\ \textsc{Nil}V$$

$$\frac{\vdash_\Sigma \Sigma' \text{ valid}}{\vdash_\Sigma \Sigma', \Delta \vdash f[\overline{n}] :: (z : A) \text{ valid}}\ \textsc{ProcDecl}V$$

$$\frac{\vdash_\Sigma \Sigma' \text{ valid}}{\vdash_\Sigma \Sigma', x \leftarrow f[\overline{n}]\ \overline{y} = P \text{ valid}}\ \textsc{ProcDef}V$$

$$\frac{\vdash_\Sigma \Sigma' \text{ valid} \qquad \overline{n}\,;\top \vdash_\Sigma A \text{ valid} \qquad A \neq V'[\overline{e}]}{\vdash_\Sigma \Sigma', V[\overline{n} \mid \phi] = A \text{ valid}}\ \textsc{TpDef}V$$

$$\frac{(\forall \ell \in L) \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma A_\ell \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \oplus\{\ell : A_\ell\}_{\ell \in L} \text{ valid}}\ \oplus V \qquad \frac{(\forall \ell \in L) \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma A_\ell \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \&\{\ell : A_\ell\}_{\ell \in L} \text{ valid}}\ \& V$$

$$\frac{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \text{ valid} \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma B \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \otimes B \text{ valid}}\ \otimes V$$

$$\frac{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \text{ valid} \qquad \mathcal{V}\,;\mathcal{C} \vdash_\Sigma B \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma A \multimap B \text{ valid}}\ \multimap V \qquad \frac{}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \mathbf{1} \text{ valid}}\ \mathbf{1}V$$

$$\frac{\mathcal{V}\,;\mathcal{C} \wedge \phi \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma ?\{\phi\}.A \text{ valid}}\ ?V \qquad \frac{\mathcal{V}\,;\mathcal{C} \wedge \phi \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma !\{\phi\}.A \text{ valid}}\ !V$$

$$\frac{\mathcal{V}, n\,;\mathcal{C} \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \exists n.A \text{ valid}}\ \exists V \qquad \frac{\mathcal{V}, n\,;\mathcal{C} \vdash_\Sigma A \text{ valid}}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma \forall n.A \text{ valid}}\ \forall V$$

$$\frac{V[\overline{n} \mid \phi] \in \Sigma \qquad \mathcal{V}\,;\mathcal{C} \vDash \phi[\overline{e}/\overline{n}]}{\mathcal{V}\,;\mathcal{C} \vdash_\Sigma V[\overline{e}] \text{ valid}}\ \textsc{def}V$$

C Subtyping Rules

$$\frac{(\forall \ell \in L) \qquad \mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A_\ell <: B_\ell}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash \oplus\{\ell : A_\ell\}_{\ell\in L} <: \oplus\{m : B_m\}_{m\in M}}\ \mathsf{st}_{\oplus}$$

$$\frac{(\forall m \in M) \qquad \mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A_m <: B_m}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash \&\{\ell : A_\ell\}_{\ell\in L} <: \&\{m : B_m\}_{m\in M}}\ \mathsf{st}_{\&}$$

$$\frac{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A_1 <: B_1 \qquad \mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A_2 <: B_2}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A_1 \otimes A_2 <: B_1 \otimes B_2}\ \mathsf{st}_{\otimes}$$

$$\frac{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash B_1 <: A_1 \qquad \mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A_2 <: B_2}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A_1 \multimap A_2 <: B_1 \multimap B_2}\ \mathsf{st}_{\multimap} \qquad \frac{}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash \mathbf{1} <: \mathbf{1}}\ \mathsf{st}_{\mathbf{1}}$$

$$\frac{\mathcal{V}\,;\mathcal{C} \vDash \phi \to \psi \qquad \mathcal{V}\,;\mathcal{C} \wedge \phi\,;\Gamma \vdash A <: B}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash ?\{\phi\}.A <: ?\{\psi\}.B}\ \mathsf{st}_{?}$$

$$\frac{\mathcal{V}\,;\mathcal{C} \vDash \psi \to \phi \qquad \mathcal{V}\,;\mathcal{C} \wedge \psi\,;\Gamma \vdash A <: B}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash !\{\phi\}.A <: !\{\psi\}.B}\ \mathsf{st}_{!}$$

$$\frac{(k \text{ fresh}) \qquad \mathcal{V}, k\,;\mathcal{C}\,;\Gamma \vdash A[k/m] <: B[k/n]}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash \exists m.A <: \exists n.B}\ \mathsf{st}_{\exists}$$

$$\frac{(k \text{ fresh}) \qquad \mathcal{V}, k\,;\mathcal{C}\,;\Gamma \vdash A[k/m] <: B[k/n]}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash \forall m.A <: \forall n.B}\ \mathsf{st}_{\forall} \qquad \frac{\mathcal{V}\,;\mathcal{C} \vDash \bot}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash A <: B}\ \mathsf{st}_{\bot}$$

$$\frac{\mathcal{V}\,;\mathcal{C} \vDash e_1 = e_1' \wedge \cdots \wedge e_n = e_n'}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash V[\overline{e}] <: V[\overline{e'}]}\ \mathsf{st}_{\mathsf{refl}}$$

$$\frac{\begin{array}{c} V_1[\overline{v_1}|\phi_1] = A \in \Sigma \qquad V_2[\overline{v_2}|\phi_2] = B \in \Sigma \\ \gamma = \langle \mathcal{V}\,;\mathcal{C}\,;V_1[\overline{e_1}] <: V_2[\overline{e_2}] \rangle \\ \mathcal{V}\,;\mathcal{C}\,;\Gamma, \gamma \vdash A[\overline{e_1}/\overline{v_1}] <: B[\overline{e_2}/\overline{v_2}] \end{array}}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash V_1[\overline{e_1}] <: V_2[\overline{e_2}]}\ \mathsf{st}_{\mathsf{expd}}$$

$$\frac{\langle \mathcal{V}'\,;\mathcal{C}'\,;V_1[\overline{e_1}'] <: V_2[\overline{e_2}'] \rangle \in \Gamma \qquad \mathcal{V}\,;\mathcal{C} \vDash \exists \mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}}{\mathcal{V}\,;\mathcal{C}\,;\Gamma \vdash V_1[\overline{e_1}] <: V_2[\overline{e_2}]}\ \mathsf{st}_{\mathsf{def}}$$

D Soundness

We first provide a series of auxiliary definitions to motivate a key lemma and our main proof.

Definition 6. *For a substitution σ, we say $\mathcal{V} \,;\, \mathcal{C} \vDash \sigma$ to abbreviate that σ is a ground substitution over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$.*

Definition 7. *Given a relation $\mathcal{R}$ on valid ground types and two types A, B with $\mathcal{V} \,;\, \mathcal{C} \vdash A, B$ valid, we say $\forall \mathcal{V}.\mathcal{C} \to A <:_{\mathcal{R}} B$ if, for all substitutions σ with $\mathcal{V} \,;\, \mathcal{C} \vDash \sigma$, we get $(A[\sigma], B[\sigma]) \in \mathcal{R}$.*

We say $\forall \mathcal{V}.\mathcal{C} \to A <: B$ if there exists a type simulation $\mathcal{R}$ satisfying $\forall \mathcal{V}.\mathcal{C} \to A <:_{\mathcal{R}} B$.

Lemma 1. *Suppose $\forall \mathcal{V}'.\mathcal{C}' \to V_1[\overline{e_1}'] <:_{\mathcal{R}} V_2[\overline{e_2}']$, and assume that $\mathcal{V} \,;\, \mathcal{C} \vDash \exists \mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}$. Then it follows that $\forall \mathcal{V}.\mathcal{C} \to V_1[\overline{e_1}] <:_{\mathcal{R}} V_2[\overline{e_2}]$.*

Proof. By definition of $<:_{\mathcal{R}}$, it suffices to show that, for all substitutions σ such that $\mathcal{V} \,;\, \mathcal{C} \vDash \sigma$, we get $V_1[\overline{e_1}[\sigma]] <:_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$.

Take an arbitrary such σ: since $\vDash \mathcal{C}[\sigma]$, we can apply σ to our second assumption, yielding that $\exists \mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1}[\sigma] \wedge \overline{e_2}' = \overline{e_2}[\sigma]$. By definition, there is *some* ground substitution σ' over $\mathcal{V}'$ such that $\vDash \mathcal{C}'[\sigma']$, $\overline{e_1}'[\sigma'] = \overline{e_1}[\sigma]$, and $\overline{e_2}'[\sigma'] = \overline{e_2}[\sigma]$.

From our first assumption, for *any* such σ' with $\vDash C'[\sigma']$, we get $V_1[\overline{e_1}'[\sigma']] <:_{\mathcal{R}} V_2[\overline{e_2}'[\sigma']]$. Since $\overline{e_1}'[\sigma'] = \overline{e_1}[\sigma]$ and $\overline{e_2}'[\sigma'] = \overline{e_2}[\sigma]$, we get $V_1[\overline{e_1}[\sigma]] <:_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$.

Since this applies for any such σ, we are done as described above: by definition, $\forall \mathcal{V}.\mathcal{C} \to V_1[\overline{e_1}] <:_{\mathcal{R}} V_2[\overline{e_2}]$.

Proof of Theorem 1. From the antecedent we get a derivation $\mathcal{D}_0$ of $\mathcal{V}, \mathcal{C}, \cdot \Vdash A <: B$. Define $\mathcal{R}$ on closed valid types as:

$$\mathcal{R} = \{(A[\sigma], B[\sigma]) \mid \langle \mathcal{V} \,;\, \mathcal{C} \,;\, A <: B \rangle \in S(\mathcal{D}_0) \text{ and } \mathcal{V} \,;\, \mathcal{C} \vDash \sigma\}$$

We will show $\mathcal{R}$ is a type simulation. To do so, we consider arbitrary $(A[\sigma], B[\sigma]) \in \mathcal{R}$; by definition of $\mathcal{R}$, there must be some closure $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A <: B \rangle \in S(\mathcal{D}_0)$ and some σ with $\mathcal{V} \,;\, \mathcal{C} \vDash \sigma$.

Consider first the case where $\mathcal{V} \,;\, \mathcal{C} \vDash \bot$. It follows from the $\mathsf{st}_\bot$ rule that $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A <: B \rangle \in S(\mathcal{D}_0)$. Furthermore, $\forall \mathcal{V}.\mathcal{C} \to A <: B$ is vacuously true, and so soundness holds.

If instead $\mathcal{V} \,;\, \mathcal{C} \nvDash \bot$, it follows that there exists some ground substitution σ on $\mathcal{V}$ that satisfies $\mathcal{C}$, i.e. $\mathcal{V} \,;\, \mathcal{C} \vDash \sigma$. We proceed by case-analysis on the structure of A with an arbitrary such σ. Most cases follow by simple structural analysis; we include a subset for demonstrative purposes here.

If $A = \oplus\{\ell : A_\ell\}_{\ell \in L}$, then by enumeration of rules, we must have $B = \oplus\{m : B_m\}_{m \in M}$. It follows from $\mathsf{st}_\oplus$ that, for all $\ell \in L$, we get $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A_\ell <: B_\ell \rangle \in S(\mathcal{D}_0)$. By definition of $\mathcal{R}$, we then have $(A_\ell[\sigma], B_\ell[\sigma]) \in \mathcal{R}$. Since $A[\sigma] = \oplus\{\ell : A_\ell[\sigma]\}_{\ell \in L}$ and $B[\sigma] = \oplus\{m : B_m[\sigma]\}_{m \in M}$, we conclude that, for an arbitrary σ,

if $(A[\sigma], B[\sigma]) \in \mathcal{R}$, then $(A_\ell[\sigma], B_\ell[\sigma]) \in \mathcal{R}$ for all $\ell \in L$. Thus, we have satisfied the condition for a type simulation.

If $A = ?\{\phi\}.A'$, then by enumeration of rules, we must have $B = ?\{\psi\}.B'$. It follows from $\mathsf{st}_?$ that we get $\langle \mathcal{V} \ ; \ \mathcal{C} \wedge \phi \ ; \ A' <: B' \rangle \in S(\mathcal{D}_0)$, as well as the semantic judgment $\mathcal{V} \ ; \ \mathcal{C} \vDash \phi \to \psi$. Since our considered σ satisfies $\vDash \mathcal{C}[\sigma]$, we have two sub-cases: either $\vDash \phi[\sigma]$ or $\nvDash \phi[\sigma]$. If $\vDash \phi[\sigma]$, it must follow that $\vDash (\mathcal{C} \wedge \phi)[\sigma]$, so by definition of $\mathcal{R}$ we get $(A'[\sigma], B'[\sigma]) \in \mathcal{R}$. Furthermore, since $\vDash \phi[\sigma]$ and $\mathcal{V} \ ; \ \mathcal{C} \vDash \phi \to \psi$, we also have $\vDash \psi[\sigma]$. Since $A[\sigma] = ?\{\phi[\sigma]\}.A'$ and $B[\sigma] = ?\{\psi[\sigma]\}.B'$, we satisfy condition (1) for a type simulation. If instead $\nvDash \phi[\sigma]$, then we trivially satisfy condition (2) for a type simulation; thus, in either subcase, it holds that $\mathcal{R}$ is a type simulation.

If $A = \exists m.A'$, then by enumeration of rules, $B = \exists n.B'$. It follows from $\mathsf{st}_\exists$ that we get $\langle \mathcal{V}, k \ ; \ \mathcal{C} \ ; \ A'[k/m] <: B'[k/m] \rangle \in S(\mathcal{D}_0)$ for some fresh k. Since k is fresh, $k \notin C$, and so for any $i \in \mathbb{N}$ we get $(A'[\sigma, i/k], B'[\sigma, i/k]) \in \mathcal{R}$, thereby satisfying that $\mathcal{R}$ is a type simulation.

If $A = V_1[\overline{e}_1]$, then we have three sub-cases: either the $\mathsf{st}_{\mathsf{refl}}$ rule, the $\mathsf{st}_{\mathsf{def}}$ rule, or the $\mathsf{st}_{\mathsf{expd}}$ rule applies; we consider them in that order. If we apply the $\mathsf{st}_{\mathsf{def}}$ rule, then we do not add anything to $S(\mathcal{D}_0)$ by definition of $\mathcal{R}$, but in fact doing so is unnecessary. The first premise of the $\mathsf{st}_{\mathsf{def}}$ rule tells us that we have already seen $\langle \mathcal{V}' \ ; \ \mathcal{C}' \ ; \ V_1[\overline{e_1}'] <: V_2[\overline{e_2}'] \rangle$, implying that $\forall \mathcal{V}'.\mathcal{C}' \to V_1[\overline{e_1}'] <:_{\mathcal{R}} V_2[\overline{e_2}']$. It follows from lemma 1 and the second premise to the $\mathsf{st}_{\mathsf{def}}$ rule that $V_1[\overline{e_1}[\sigma]] <:_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$, so therefore $(V_1[\overline{e_1}[\sigma]], V_2[\overline{e_2}[\sigma]]) \in \mathcal{R}$.

We now conclude our proof. Since our derivation $\mathcal{D}_0$ must prove $\mathcal{V}_0, \mathcal{C}_0, \cdot \Vdash A <: B$, we have $\langle \mathcal{V}_0 \ ; \ \mathcal{C}_0 \ ; \ A <: B \rangle \in S(\mathcal{D}_0)$ by definition of S. Then, by definition of $\mathcal{R}$, we have $(A[\sigma], B[\sigma]) \in \mathcal{R}$ for any σ over $\mathcal{V}$ satisfying $\vDash \mathcal{C}[\sigma]$. By **??**, we then say $\forall \mathcal{V}.\mathcal{C} \to A <:_{\mathcal{R}} B$, and finally, since $\mathcal{R}$ is a type simulation, we say $\forall \mathcal{V}.\mathcal{C} \to A <: B$ and we are done.

E Constraint Generation Rules

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash \oplus\{k : B\} <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : B)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash x.k\,;\,P :: (x : A)}\ \oplus\mathrm{R}$$

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

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

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: \&\{k : A_k\} \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A_k) \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash x.k\,;\,P :: (z : C)}\ \&\,\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A_1 \otimes A_2 <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash P :: (x : A_2)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (y : A_1) \vdash \mathsf{send}\ x\ y\,;\,P :: (x : A)}\ \otimes\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: A_1 \otimes A_2 \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (y : A_1), (x : A_2) \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash y \leftarrow \mathsf{recv}\ x\,;\,Q :: (z : C)}\ \otimes\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A_1 \multimap A_2 <: A \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (y : A_1) \vdash P :: (x : A_2)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash y \leftarrow \mathsf{recv}\ x\,;\,P :: (x : A)}\ \multimap\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: A_1 \multimap A_2 \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A_2) \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A), (y : A_1) \vdash \mathsf{send}\ x\ y\,;\,Q :: (z : C)}\ \multimap\mathrm{L}$$

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

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

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

$$\frac{\begin{array}{c}(\overline{y'_i : B'_i})_{i\in I} \vdash f[\overline{n} \mid \phi] = P_f :: (x' : A') \in \Sigma \\ \mathcal{V}\,;\,\mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash A'[\overline{e}/\overline{n}] <: A \qquad (i \in I) \quad \mathcal{V}\,;\,\mathcal{C} \wedge \phi[\overline{e}/\overline{n}] \Vdash B_i <: B'_i[\overline{e}/\overline{n}] \\ \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A[\overline{e}/\overline{n}]) \vdash Q :: (z : C)\end{array}}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (\overline{y_i : B_i})_{i\in I} \vdash x \leftarrow f[\overline{e}]\ \overline{y}\,;\,Q :: (z : C)}\ \mathsf{def}$$

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

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

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

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash \phi \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash A <: !\{\phi\}.A' \qquad \mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A') \vdash Q :: (z : C) \rightsquigarrow \Xi}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ !\mathrm{L}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash e \geq 0 \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash \exists n.A' <: A \qquad \mathcal{V}\,;\,\mathcal{C} \wedge n = e\,;\,\Delta \vdash P :: (x : A')}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta \vdash \mathsf{send}\ x\ \{e\}\ ;\ P :: (x : A)}\ \exists\mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{C} \Vdash A <: \exists n.A' \qquad \mathcal{V}, n\,;\,\mathcal{C}\,;\,\Delta, (x : A') \vdash Q_n :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ Q_n :: (z : C)}\ \exists\mathrm{L}$$

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

$$\frac{\mathcal{V}\,;\,\mathcal{C} \vDash e \geq 0 \qquad \mathcal{V}\,;\,\mathcal{C} \Vdash A <: \forall n.A' \qquad \mathcal{V}\,;\,\mathcal{C} \wedge n = e\,;\,\Delta, (x : A') \vdash Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{C}\,;\,\Delta, (x : A) \vdash \mathsf{send}\ x\ \{e\}\ ;\ Q :: (z : C)}\ \forall\mathrm{L}$$