RAST: A LANGUAGE FOR RESOURCE-AWARE SESSION TYPES

ANKUSH DAS [a,*] AND FRANK PFENNING [b]

[a] Amazon, Cupertino, CA, USA
  e-mail address: ankushd@cs.cmu.edu

[b] Carnegie Mellon University
  e-mail address: fp@cs.cmu.edu

ABSTRACT. Traditional session types prescribe bidirectional communication protocols for concurrent computations, where well-typed programs are guaranteed to adhere to the protocols. However, simple session types cannot capture properties beyond the basic type of the exchanged messages. In response, recent work has extended session types with refinements from linear arithmetic, capturing intrinsic attributes of processes and data. These refinements then play a central role in describing sequential and parallel complexity bounds on session-typed programs. The Rast language provides an open-source implementation of session-typed concurrent programs extended with arithmetic refinements as well as ergometric and temporal types to capture work and span of program execution. To further support generic programming, Rast also enhances arithmetically refined session types with recently developed nested parametric polymorphism. Type checking relies on Cooper's algorithm for quantifier elimination in Presburger arithmetic with a few significant optimizations, and a heuristic extension to nonlinear constraints. Rast furthermore includes a reconstruction engine so that most program constructs pertaining the layers of refinements and resources are inserted automatically. We provide a variety of examples to demonstrate the expressivity of the language.

1. INTRODUCTION

Session types [Hon93, HVK98, Vas12] provide a structured way of statically prescribing communication protocols in message-passing programs. In this paper, we introduce the Rast programming language and implementation which is based on *binary session types* governing the interaction of two processes along a single channel, rather than *multiparty session types* [HYC08] which take a more global view of computation. Nevertheless, during the execution of a Rast program, complex networks of interacting processes arise. Recent work has placed binary session types without general recursion on a strong logical foundation by exhibiting a Curry-Howard isomorphism with linear logic [CP10, Wad12, CPT16]. Due

to this correspondence, the cut reduction properties of linear logic entail type safety of session typed processes and guarantee *freedom from deadlocks* (global progress) and *session fidelity* (type preservation) ensuring adherence to the communication protocols at runtime.

The Rast programming language is based on session types derived from intuitionistic linear logic, extended with equirecursive types and recursive process definitions. Rast also supports full parametric polymorphism enabling definition of polymorphic data structures. It furthermore supports arithmetic type refinements as well as ergometric and temporal types to measure the total work and span of Rast programs. The theory underlying Rast has been developed in several papers, including the Curry-Howard interpretation of linear logic as session-typed processes [CP10, CPT16], the treatment of general equirecursive types and type equality [GH05], asynchronous communication [GV10, DCPT12], ergometric types [DHP18b], temporal types [DHP18a], indexed types [GG13, DP20c], indexed type equality [DP20b], and nested polymorphism [DDMP21].

We focus on key aspects of language design and implementation, not the underlying theory which can be found in the cited papers. A notable exception is subtyping for the full language, including nested polymorphic types, whose properties are the subject of ongoing research. We present Rast in layers, using typing rules, an operational semantics, and examples to explain and illustrate increasingly advanced features via their type structure. All language layers satisfy the properties of preservation (session fidelity) and progress (deadlock-freedom), with slightly different statements depending on the semantic properties under consideration. For example, in the presence of ergometric types the sum of potential and total work expended remains constant. This paper is a significantly revised and extended version of a system description [DP20a], including the formal definition of typing and computation, nested parametric polymorphism, subtyping, and additional examples.

We begin with motivation and a brief overview of the main features of the language using a concurrent queue data structure as a running example. The following type specifies the interface to a polymorphic queue server in the system of basic recursive session types storing elements of type A and supporting the operations of insert and delete.

$$\mathsf{queue}[A] = \&\{\mathbf{ins} : A \multimap \mathsf{queue}[A],$$
$$\mathbf{del} : \oplus\{\mathbf{none} : \mathbf{1},$$
$$\mathbf{some} : A \otimes \mathsf{queue}[A]\}\}$$

The *external choice* operator $\&$ dictates that the process providing this data structure accepts either one of two messages: the labels **ins** or **del**. In the case of **ins**, it receives an element of type A denoted by the $\multimap$ operator, and then the type recurses back to $\mathsf{queue}[A]$. On receiving a **del** request, the process can respond with one of two labels (**none** or **some**), indicated by the *internal choice* operator $\oplus$. If the queue is empty, it responds with **none** and then *terminates* (indicated by $\mathbf{1}$). If the queue is nonempty, it responds with **some** followed by the element of type A (expressed with the $\otimes$ operator) and recurses. The type $\mathsf{queue}[A]$ denotes the type name queue instantiated with the session type A.

However, the simple session type does not express the conditions under which the **none** and **some** branches must be chosen, which requires tracking the length of the queue. Rast extends session types with arithmetic refinements [DP20b, DP20c] which can be used to express the length of a queue. The more precise type

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

uses the index refinement n to indicate the number of elements in the queue. In addition, the *type constraint* $?\{\phi\}.\,A$ read as "*there exists a proof of* ϕ" is analogous to the *assertion* of ϕ in imperative languages. Conceptually, the process providing the queue must provide a proof of $n = 0$ after sending **none**, and a proof of $n > 0$ after sending **some** respectively. It is therefore constrained in its choice between the two branches based on the value of the index n. Since the constraint domain is decidable and the actual form of a proof is irrelevant to the outcome of a computation, in the implementation no proof is actually sent.

As is standard in session types, the dual constraint to $?\{\phi\}.\,A$ is $!\{\phi\}.\,A$ (*for all proofs of ϕ*, analogous to the *assumption* of ϕ). We also add explicit quantifier type constructors $\exists n.\,A$ and $\forall n.\,A$ that send and receive natural numbers, respectively.

Arithmetic refinements are instrumental in expressing *sequential* and *parallel complexity bounds*. These are captured with ergometric [DHP18b, DBH+21] and temporal session types [DHP18a]. They rely on index refinements to express, for example, the size of lists, stacks, and queue data structures, or the height of trees and express work and time bounds as a function of these refinements.

Ergometric session types [DHP18b] capture the sequential complexity of programs, often called the *work*. Revisiting the queue example, consider an implementation where each element in the queue corresponds to a process. Then insertion acts like a bucket brigade, passing the new element one by one to the end of the queue. Among multiple cost models provided by Rast is one where each *send* operation requires 1 unit of work (erg). In this cost model, such a bucket brigade requires $2n$ ergs because each process has to send **ins** and then the new element. On the other hand, responding to the **del** request requires only 2 ergs: the provider responds with **none** and closes the channel, or **some** followed by the element. This gives us the following type

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

which expresses that the client has to send $2n$ ergs of potential to insert an element ($\triangleleft^{2n}$), and 2 ergs to delete an element ($\triangleleft^{2}$). The ergometric type system (described in Section 6) verifies this work bound using the potential operators as described in the type.

Temporal session types [DHP18a] capture the time complexity of programs assuming maximal parallelism on unboundedly many processors, often called the *span*. How does this work out in our example? We adopt a cost model where each send and receive action takes one unit of time (tick). First, we note that a use of a queue is at the client's discretion, so should be available at *any* point in the future, expressed by the type constructor $\square$. Secondly, the queue does not interact at all with the elements it contains, so they have to be of type $\square A$ for an arbitrary A. Since each interaction takes 1 tick, the next interaction requires at least 1 tick to elapse, captured by the next-time operator $\bigcirc$. During insertion, we need more time than this: a process needs 2 ticks to pass the element down the queue, so it takes 3 ticks overall until it can receive the next insert or delete request after an insertion. This reasoning yields the following temporal type:

$$\mathsf{queue}[A]\{n\} = \square \,\&\, \{\mathbf{ins} : \bigcirc(\square A \multimap \bigcirc^{3}\mathsf{queue}[A]\{n+1\}),$$
$$\mathbf{del} : \bigcirc \oplus \{\mathbf{none} : \bigcirc ?\{n = 0\}.\,\mathbf{1},$$
$$\mathbf{some} : \bigcirc ?\{n > 0\}.\,\square A \otimes \bigcirc \mathsf{queue}[A]\{n-1\}\}\}$$

We see that even though the bucket brigade requires much work for every insertion (linear in the length of the queue), it has a lot of parallelism because there are only a constant number of required delays between consecutive insertions or deletions.

Rast follows the design principle that bases an *explicit language* directly on the correspondence with the sequent calculus for the underlying logic (such as linear logic, or temporal or ergometric linear logic), extended with recursively defined types and processes. Programming in this fully explicit form tends to be unnecessarily verbose, so Rast also provides an *implicit language* in which most constructs related to index refinements and amortized work analysis are omitted. Explicit programs are then recovered by a proof-theoretically motivated algorithm [DP20c] for *reconstruction* which is sound and complete on valid implicit programs.

Rast is implemented in SML and available as an open-source repository [DDP19]. It allows the user to choose explicit or implicit syntax and the exact cost models for work and time analysis. The implementation consists of a lexer, parser, reconstruction engine, arithmetic solver, type checker, and an interpreter, with particular attention to providing precise error messages. The repository also contains a number of illustrative examples that highlight various language features, some of which we briefly sketch in this paper.

To summarize, Rast makes the following contributions:

(1) A session-typed programming language with arithmetic refinements applied to ergometric and temporal types for parallel complexity analysis.
(2) An extension with full parametric polymorphism to enable generic programming.
(3) A subtyping algorithm that works well in practice despite its theoretical undecidability [DP20b] and uses Cooper's algorithm [Coo72] with some small improvements to decide constraints in Presburger arithmetic (and heuristics for nonlinear constraints).
(4) A type checking algorithm that is sound and complete relative to subtyping.
(5) A sound and complete reconstruction algorithm for a process language where most index and ergometric constructs remain implicit.
(6) An interpreter for executing session-typed programs using the recently proposed shared memory semantics [PP20].

2. Example: An Implementation of Queues

We use the implementation of queues as sketched in the introduction as a first example program, starting with the indexed version. The concrete syntax of types is a straightforward rendering of their abstract syntax (Table 6).

```
type queue[A]{n} = &{ins : A -o queue[A]{n+1},
                      del : +{none : ?{n = 0}. 1,
                              some : ?{n > 0}. A * queue[A]{n-1}}}
```

Each channel has exactly two endpoints: a *provider* and a *client*. Session fidelity ensures that provider and client always agree on the type of the channel and carry out complementary actions. The type of the channel evolves during communication, since it has to track where the processes are in the protocol as they exchange messages.

In our example, we need two kinds of processes: an *empty* process at the end of the queue, and an *elem* process that holds an element x. The empty process *provides* an empty queue, that is, a service of type `queue[A]{0}` along a channel named `q`. It does not *use* any channels (indicated by an empty context '.'), so its type is declared with

```
type queue[A]{n} = &{ins : A -o queue[A]{n+1},
                      del : +{none : ?{n = 0}. 1,
                              some : ?{n > 0}. A * queue[A]{n-1}}}

decl empty[A] : . |- (q : queue[A]{0})
decl elem[A]{n} : (x : A) (t : queue[A]{n}) |- (q : queue{n+1})

proc q <- empty[A] =
  case q (                            % receive a label along q
   ins => x <- recv q ;               % if 'ins', receive a channel x along q
          e <- empty[A] ;             % spawn a new empty process (at type A)
          q <- elem[A]{0} x e         % continue as an elem holding x
 | del => q.none ;                    % if 'del', respond with label 'none'
          assert q {0=0} ;            % assert that (n = 0)[0/n]
          close q )                   % terminate by closing q

proc q <- elem[A]{n} x t =
  case q (                            % receive a label along q
   ins => y <- recv q ;               % if 'ins', receive a channel y along q
          t.ins ;                     % send label 'ins' along t
          send t y ;                  % send the channel y along t
          q <- elem[A]{n+1} x t       % continue as elem at index n+1
 | del => q.some ;                    % if 'del', respond with label 'some'
          assert q {n+1>0} ;          % assert that (n > 0)[n+1/n]
          send q x ;                  % send x along q
          q <-> t )                   % identify q with t and terminate
```

Listing 1: Declaration and definition of queue processes, file `examples/list.rast`

```
  decl empty[A] : . |- (q : queue[A]{0})
```

Polymorphism is explicit in Rast, requiring the programmer to specify the free type variables. The declaration `empty[A]` denotes that the type `A` may occur freely in its type and definition.

An *elem* process *provides* a service of type `queue[A]{n+1}` along a channel named `q` and *uses* a queue of type `queue[A]{n}` along a channel named `t`. In addition, it holds ("owns") an element `x` of type `A`.

```
  decl elem[A]{n} : (x : A) (t : queue[A]{n}) |- (q : queue[A]{n+1})
```

The turnstile '`|-`' separates the channels used from the channel that is provided (which is always exactly one, analogous to a value returned by a function). The notation `elem[A]{n}` indicates that the type `A` and natural number `n` are parameters of this process.

Listing 1 shows the implementation of the two forms of processes in Rast. Comments, starting with a `%` character and extending to the end of the line, provide a brief explanation for the actions of each line of code. This code is in *explicit* form, both in its refinements and polymorphism. Thus, the type and natural number parameters to a process need to be

specified explicitly (e.g., `elem[A]{0}` or `elem[A]{n+1}`). Moreover, the code contains two instances of `assert` to match the constraints `?{n = 0}` and `?{n > 0}` in the two possible responses to a delete request. Rast also provides the programmer with an option to write code in *implicit* form where the two asserts would be omitted since they can be read off the type at the corresponding place in the protocol. Of course, the type checker verifies that the assertion is justified and fails with an error message if it is not, whether the construct is explicit or implicit. However, even in the implicit form, parameters (both type and natural numbers) to a process need to be provided to spawn a new instance of it. The full implementation can be found in `tests/queues-quant.rast` in the repository.

Concealing Queue Size. As a final pair of illustrations, we describe how to wrap a sized queue in an unsized one, and vice-versa. First, we define the type of an unsized queue (introduced in Section 1).

```
type uqueue[A] = &{ins : A -o uqueue[A],
                    del : +{none : 1,
                            some : A * uqueue[A]}}
```

Next, we define a `conceal` process that uses a sized queue and provides an unsized queue, thus hiding, but *not discarding* the queue size.

```
decl conceal[A]{n} : (q : queue[A]{n}) |- (u : uqueue[A])


proc u <- conceal[A]{n} q =
  case u (                            % receive label along unsized queue u
   ins => x <- recv q ;               % if 'ins', receive channel x along u
          q.ins ;                     % send label 'ins' along sized queue q
          send q x ;                  % send channel x along q
          u <- conceal[A]{n+1} q      % continue as conceal at index n+1
 | del => q.del ;                     % if 'del', send 'del' to q
          case q (                    % case on label received from q
            none => u.none ;          % if 'none', send 'none' on u
                    u <-> q           % identify and terminate
          | some => x <- recv q ;     % receive channel x from q
                    u.some ;          % send 'some' on u
                    send u x ;        % send channel x on u
                    u <- conceal[A]{n-1} q
          )
  )
```

The `conceal` process case analyzes on the label received on `u`. If it receives an `ins` request on `u`, it forwards the same on `q` and recurses at index `n+1` Similarly, it forwards the `del` request to `q` and forwards on the response from `q` to `u`. In the `none` branch, it identifies the channels `u` and `q` and terminates. In the `some` branch, it forwards the element `x` received from `q` on to `u` and recurses at index `n+1`.

We can also define a dual `display` process that takes an unsized queue as input and provides a sized queue.

```
decl display[A] : (u : uqueue[A]) |- (q : ?n. queue[A]{n})
```

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

Table 1: Basic session types with operational description

In this case, we use existential quantifier to express that there exists an `n` such that the unsized queue has size `n`. The `display` process will first send an index natural number `n` and then behave as a queue of size `n`. We omit the process definition which also exists in the file `tests/queues-quant.rast` in the repository.

3. Basic System of Session Types

The underlying system of session types is derived from a Curry-Howard interpretation [CP10, CPT16] of intuitionistic linear logic [GL87]. The key idea is that an intuitionistic linear sequent $A_1 \; A_2 \; \ldots \; A_n \vdash A$ is interpreted as the interface to a process P. We label each of the antecedents with a channel name x_i and the succedent with channel name z. The x_i's are *channels used by* P and z is the *channel provided by* P.

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

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

$$\Delta \vdash_\Sigma P :: (x : A)$$

where Δ represents the linear antecedents $x_i : A_i$, P is the process expression and $x : A$ is the linear succedent. Additionally, Σ is a fixed valid signature containing type and process definitions (explained later). Because it is fixed, we elide it from the presentation of the typing rules. We will extend the typing judgment in the subsequent sections as we introduce polymorphism (Section 4), refinements (Section 5), ergometric (Section 6), and temporal session types (Section 7).

At runtime, a program is represented using a multiset of semantic objects denoting processes and messages defined as a *configuration*.

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

We formalize the operational semantics as a system of *multiset rewriting rules* [CS09]. We introduce semantic objects $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$ which mean that process P or message

M provide along channel c. A process configuration is a multiset of such objects, where any two provided channels are distinct.

In this section, we briefly review the structural type formers that constitute the base fragment of Rast. The type grammar is defined as

$$\text{Types} \quad A, B, C \quad ::= \quad \oplus\{\ell : A\}_{\ell \in L} \mid \&\{\ell : A\}_{\ell \in L} \mid A \otimes B \mid A \multimap B \mid \mathbf{1} \mid V$$

Table 1 overviews the types, their associated process terms, their continuation (both in types and terms) and operational description. For each type, the first row describes the provider's viewpoint, while the second row describes the client's matching but dual viewpoint.

Choice Operators. The _internal choice_ type constructor $\oplus\{\ell : A_\ell\}_{\ell \in L}$ is an n-ary labeled generalization of the additive disjunction $A \oplus B$. Operationally, it requires the provider of $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ to send a label label $k \in L$ on channel x and continue to provide type A_k. The corresponding process term is written as $(x.k \ ; \ P)$ where the continuation P provides type $x : A_k$. Dually, the client must branch based on the label received on x using the process term $\mathsf{case} \ x \ (\ell \Rightarrow Q_\ell)_{\ell \in L}$ where Q_ℓ is the continuation in the ℓ-th branch.

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

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

Communication is asynchronous, so that the provider $(c.k \ ; \ P)$ sends a message k along c and continues as P without waiting for it to be received. As a technical device to ensure that consecutive messages on a channel arrive in order, the sender also creates a fresh continuation channel c' so that the message k is actually represented as $(c.k \ ; \ c \leftrightarrow c')$ (read: send k along c and continue along c'). This formulation has the advantage that a message is just a special form of process only with a different semantic symbol. The reason we distinguish processes and messages in the semantics is that messages, unlike processes, are only allowed to interact with other processes, not spontaneously create messages. For instance, in the $\oplus S$ rule below, if we used proc to represent the message, we could apply the same rule recursively to create new messages. When the message k is received along c, we select branch k and also substitute the continuation channel c' for c. Rules $\oplus S$ and $\oplus C$ below describe the operational behavior of the provider and client respectively.

$(\oplus S) : \mathsf{proc}(c, c.k \ ; \ P) \mapsto \mathsf{proc}(c', P[c'/c]), \ \mathsf{msg}(c, c.k \ ; \ c \leftrightarrow c') \qquad (c' \text{ fresh})$

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

The _external choice_ constructor $\&\{\ell : A_\ell\}_{\ell \in L}$ generalizes additive conjunction and is the _dual_ of internal choice reversing the role of the provider and client. Thus, the provider branches on the label $k \in L$ sent by the client. The typing rules are as follows

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

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

Semantics rules $\& S$ and $\& C$ express the operational behavior at runtime.

$(\& S) : \mathsf{proc}(d, c.k \; ; \; Q) \;\mapsto\; \mathsf{msg}(c', c.k \; ; \; c' \leftrightarrow c), \; \mathsf{proc}(d, Q[c'/c]) \qquad (c' \text{ fresh})$

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

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

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

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

Operationally, the provider $(\mathsf{send} \; c \; d \; ; \; P)$ sends the channel d and the continuation channel c' along c as a message and continues with executing P. The client receives the channel d and continuation channel c' and substitutes d for x and c' for c.

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

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

The dual operator $A \multimap B$ allows the provider $(y \leftarrow \mathsf{recv} \; x \; ; \; P)$ to receive a channel of type A and continue to provide type B with process term P. The client of $A \multimap B$, on the other hand, sends channel w of type A and continues to use B using term $(\mathsf{send} \; x \; w \; ; \; Q)$.

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

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

The semantics rules are the exact dual to $\otimes$.

$(\multimap S) : \mathsf{proc}(e, \mathsf{send} \; c \; d \; ; \; Q) \;\mapsto\; \mathsf{msg}(c', \mathsf{send} \; c \; d \; ; \; c' \leftrightarrow c), \; \mathsf{proc}(e, Q[c'/c]) \qquad (c' \text{ fresh})$

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

Termination. The type $\mathbf{1}$ indicates *termination* requiring that the provider of $x : \mathbf{1}$ send a *close* message, formally written as $(\mathsf{close} \; x)$ followed by terminating the communication. Correspondingly, the client of $x : \mathbf{1}$ uses the term $(\mathsf{wait} \; x \; ; \; Q)$ to wait for the close message before continuing with executing Q. Linearity enforces that the provider does not use any channels, as indicated by the empty context in $\mathbf{1}R$.

$$\frac{}{\cdot \vdash (\mathsf{close} \; x) :: (x : \mathbf{1})} \; \mathbf{1}R \qquad\qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, (x : \mathbf{1}) \vdash (\mathsf{wait} \; x \; ; \; Q) :: (z : C)} \; \mathbf{1}L$$

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

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

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

Forwarding. A forwarding process $(x \leftrightarrow y)$ identifies the channels x and y so that any further communication along either x or y will be along the unified channel. Its typing rule corresponds to the logical rule of identity.

$$\frac{}{y : A \vdash (x \leftrightarrow y) :: (x : A)} \text{ id}$$

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

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

$M(c)$ indicates that c occurs in message M ensuring that M is the sole client of c.

Process Definitions. Process definitions have the form $\Delta \vdash f = P :: (x : A)$ where f is the name of the process and P its definition, with Δ being the channels used by f and $x : A$ being the provided channel. All definitions are collected in a fixed global signature Σ. For a *valid signature*, we require that $\Delta \vdash P :: (x : A)$ for every definition, thereby allowing definitions to be mutually recursive. A new instance of a defined process f can be spawned with the expression $x \leftarrow f \ \overline{y} \ ; \ Q$ where $\overline{y}$ is a sequence of channels matching the antecedents Δ. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q. The following def rule describes the typing of a spawn.

$$\frac{\overline{y' : B'} \vdash f = P_f :: (x' : B) \in \Sigma \qquad \Delta' = \overline{(y : B')} \qquad \Delta, (x : B) \vdash Q :: (z : C)}{\Delta, \Delta' \vdash_\Sigma (x \leftarrow f \ \overline{y} \ ; \ Q) :: (z : C)} \text{ def}$$

The declaration of f is looked up in the signature Σ (first premise), and the channel types in Δ are matched to the signature $\overline{B'}$ (second premise). Similarly, the freshly created channel x has type B from the signature. The corresponding semantics rule $\text{def}C$ also performs a similar substitution.

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

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

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

Type Definitions. Session types can be defined recursively, departing from a strict Curry-Howard interpretation of linear logic, analogous to the way pure ML or Haskell depart from a pure interpretation of intuitionistic logic. A type definition for a type name V is of the form $V = A$, where A is a type expression. The signature Σ contains all type definitions, that are possibly mutually recursive. For a well-formed signature, we require A to be *contractive*, i.e., A itself must not be a type name. Our type definitions are *equirecursive* so we can silently replace type names V by A during type checking, and no explicit rules for recursive types are needed. Because both process definitions and type definitions may be recursive, processes in our language may not be terminating.

4. Polymorphic Session Types

In this section, we describe the modifications to the Rast language required to realize nested polymorphism [DDMP21] to support general-purpose programming. First, due to the presence of type variables, the formal typing judgment is extended with $\mathfrak{v}$ and written as

$$\mathfrak{v} \; ; \; \Delta \vdash_\Sigma P :: (x : A)$$

where $\mathfrak{v}$ stores the type variables (which we denote by α). We presuppose and maintain that all free type variables in Δ, P, and A are contained in $\mathfrak{v}$. The signature Σ is still fixed but the process and type definitions are now (possibly) parameterized by type variables.

To support polymorphism, we need two primary additions. First, we need to update the form of process and type definitions. Secondly, we add explicit quantifiers to allow exchange of types at runtime. Thus, the extended type grammar is

$$\text{Types} \quad A \quad ::= \quad \ldots \mid \alpha \mid V[\overline{A}] \mid \exists \alpha. \, A \mid \forall \alpha. \, A$$

Type Definitions. A type name V is now defined according to a definition $V[\overline{\alpha}] = A$ that is parameterized by a sequence of *distinct type variables* $\overline{\alpha}$ that its definition A can refer to. We can use type names in an expression using $V[\overline{B}]$. Type expressions can also refer to parameter α available in scope. The *free variables* in type A refer to the set of type variables that occur freely in A. Since types are *equirecursive*, the type $V[\overline{B}]$ is considered equivalent to its unfolding $A[\overline{B}/\overline{\alpha}]$. All type names V occurring in a valid signature must be defined, and all type variables defined in a valid definition must be distinct. Furthermore, for a valid definition $V[\overline{\alpha}] = A$, the free variables occurring in A must be contained in $\overline{\alpha}$.

Process Definitions. Process definitions now have the form $\Delta \vdash f[\overline{\alpha}] = P :: (x : A)$ where f is the name of the process and P its definition, with Δ being the channels used by f and $x : A$ being the offered channel. In addition, $\overline{\alpha}$ is a sequence of type variables that Δ, P and A can refer to. These type variables are implicitly universally quantified at the outermost level. The spawn expression $(x \leftarrow f[\overline{A}] \; \overline{y} \; ; \; Q)$ now takes a sequence of types $\overline{A}$ matching the type variables $\overline{\alpha}$, making the polymorphism explicit. The def rule is updated to reflect this modification.

$$\frac{\overline{y' : B'} \vdash f[\overline{\alpha}] = P_f :: (x' : B) \in \Sigma \qquad \begin{array}{c} \\ \Delta' = \overline{(y : B')}[\overline{A}/\overline{\alpha}] \qquad \mathfrak{v} \; ; \; \Delta, (x : B[\overline{A}/\overline{\alpha}]) \vdash Q :: (z : C) \end{array}}{\mathfrak{v} \; ; \; \Delta, \Delta' \vdash (x \leftarrow f[\overline{A}] \; \overline{y} \; ; \; Q) :: (z : C)} \; \text{def}$$

Note that $\overline{A}$ is substituted for $\overline{\alpha}$ while matching the types in Δ' and $\overline{y}$ (second premise). Similarly, the provided channel x has type B from the signature with $\overline{A}$ substituted for $\overline{\alpha}$.

Explicit Quantifiers. To support full parametric polymorphism, Rast also provides two dual type constructors, $\exists \alpha. \, A$ and $\forall \alpha. \, A$ to exchange types between processes. Table 2 describes these types along with their operational behavior. The existential type $\exists \alpha. \, A$ requires the provider to send a valid type B and continue to provide $A[B/\alpha]$. The term used to send a type is $(\text{send } x \; [B] \; ; \; P)$ where P is the continuation. Dually, the client receives a type using the expression $([\alpha] \leftarrow \text{recv } x \; ; \; Q)$, binds it to variable α, and then continues to execute Q using the session type A which possibly contains the free variable α. We introduce an auxiliary judgment $\mathfrak{v} \vdash B$ valid to define that all free variables of B are contained in $\mathfrak{v}$. The corresponding typing rules are

Type	Cont.	Process Term	Cont.	Description
$c : \exists \alpha.\, A$	$c : A[B/\alpha]$	send $c\ [B]$; P	P	provider sends type B along c
		$[\alpha] \leftarrow$ recv c ; Q_α	$Q_\alpha[B/\alpha]$	client receives type B along c
$c : \forall \alpha.\, A$	$c : A[B/\alpha]$	$[\alpha] \leftarrow$ recv c ; P_α	$P_\alpha[B/\alpha]$	provider receives type B along c
		send $c\ [B]$; Q	Q	client sends type B along c

Table 2: Explicitly quantified session types with operational description

$$\frac{\mathfrak{v} \vdash B\ \mathsf{valid} \qquad \mathfrak{v} \ ;\ \Delta \vdash P :: (x : A[B/\alpha])}{\mathfrak{v} \ ;\ \Delta \vdash (\mathsf{send}\ x\ [B]\ ;\ P) :: (x : \exists \alpha.\, A)}\ \exists R$$

$$\frac{\mathfrak{v}, \alpha \ ;\ \Delta, (x : A) \vdash Q :: (z : C)}{\mathfrak{v} \ ;\ \Delta, (x : \exists \alpha.\, A) \vdash ([\alpha] \leftarrow \mathsf{recv}\ x\ ;\ Q) :: (z : C)}\ \exists L$$

The provider checks whether type B is valid and continues to provide type $A[B/\alpha]$. The client receives a type, binding it to α which is ensured by adding α to $\mathfrak{v}$.

Operationally, the provider (send $x\ [B]$; P) sends the type B and the continuation channel c' along c and continues executing P. The client receives the type B and continuation channel c' and substitutes B for α and c' for c in Q.

$(\exists S) : \mathsf{proc}(c, \mathsf{send}\ c\ [B]\ ;\ P) \mapsto \mathsf{proc}(c', P[c'/c]),\ \mathsf{msg}(c, \mathsf{send}\ c\ [B]\ ;\ c \leftrightarrow c') \qquad (c'\ \mathrm{fresh})$

$(\exists C) : \mathsf{msg}(c, \mathsf{send}\ c\ [B]\ ;\ c \leftrightarrow c'),\ \mathsf{proc}(e, [\alpha] \leftarrow \mathsf{recv}\ c\ ;\ Q) \mapsto \mathsf{proc}(e, Q[c', B/c, \alpha])$

Dually, a provider with a universally typed session $\forall \alpha.\, A$ receives an arbitrary type, binds it to α, and proceeds to provide session prescribed by type A, possibly referring α. On the other hand, the client sends a valid type B and continues with session $A[B/\alpha]$. The formal rules for $\forall \alpha.\, A$ are

$$\frac{\mathfrak{v}, \alpha \ ;\ \Delta \vdash P :: (x : A)}{\mathfrak{v} \ ;\ \Delta \vdash ([\alpha] \leftarrow \mathsf{recv}\ x\ ;\ P) :: (x : \forall \alpha.\, A)}\ \forall R$$

$$\frac{\mathfrak{v} \vdash B\ \mathsf{valid} \qquad \mathfrak{v} \ ;\ \Delta, (x : A[B/\alpha]) \vdash Q :: (z : C)}{\mathfrak{v} \ ;\ \Delta, (x : \forall \alpha.\, A) \vdash (\mathsf{send}\ x\ [B]\ ;\ Q) :: (z : C)}\ \forall L$$

$(\forall S) : \mathsf{proc}(d, \mathsf{send}\ c\ [B]\ ;\ P) \mapsto \mathsf{msg}(c', \mathsf{send}\ c\ [B]\ ;\ c' \leftrightarrow c),\ \mathsf{proc}(d, P[c'/c]) \qquad (c'\ \mathrm{fresh})$

$(\forall C) : \mathsf{proc}(c, [\alpha] \leftarrow \mathsf{recv}\ c\ ;\ Q),\ \mathsf{msg}(c', \mathsf{send}\ c\ [B]\ ;\ c' \leftrightarrow c) \mapsto \mathsf{proc}(c', Q[c', B/c, \alpha])$

Since polymorphism is parametric, it is possible to avoid explicitly sending types at runtime if this optimization is desired and does not interfere with other lower-level aspects of an implementation such as dynamic monitoring.

Example: Context-Free Languages. Recursive session types capture the class of regular languages [TV16]. However, in practice, many useful languages are beyond regular. As an illustration, suppose we would like to express a balanced parentheses language, also known as the Dyck language [Dyc82] with the end-marker $. We use **L** to denote an opening symbol, and **R** to denote a closing symbol (in a session-typed mindset, **L** can represent client request and **R** is server response). We need to enforce that each **L** has a corresponding closing **R** and they are properly nested. To express this, we need to track the number of **L**'s in the output with the session type. However, this notion of *memory* is beyond the expressive power of regular languages, so mere recursive session types will not suffice.

We utilize the expressive power of nested types to express this behavior.

```
type T[x] = +{L : T[T[x]], R : x}
type D = +{L : T[D], $ : 1}
```

The nested type `T[T[x]]` takes `x` as a type parameter and either outputs **L** and continues with `T[T[x]]`, or outputs **R** and continues with `x`. The type `D` either outputs **L** and continues with `T[D]`, or outputs $ and terminates. The type `D` expresses a Dyck word [KH66].

The key idea here is that the number of `T`'s in the type of a word tracks the number of unmatched **L**'s in it. Whenever the type `T[x]` outputs **L**, it recurses with `T[T[x]]` incrementing the number of `T`'s in the type by 1. Dually, whenever the type outputs **R**, it recurses with x decrementing the number of `T`'s in the type by 1. The type `D` denotes a balanced word with no unmatched **L**'s. Moreover, since we can only output $ (or **L**) at the type `D` and *not* **R**, we obtain the invariant that any word of type `D` must be balanced. If we imagine the parameter x as the symbol stack, outputting an **L** pushes `T` on the stack, while outputting **R** pops `T` from the stack. The definition of `D` ensures that once an **L** is outputted, the symbol stack is initialized with `T[D]` indicating one unmatched **L**. The file `polytests/dyck.rast` in the repository contains the complete code.

5. Refinement Session Types

In this section, we index types with arithmetic refinements that describe intrinsic attributes of corresponding channels. In addition to the type constructors arising from the connectives of intuitionistic linear logic ($\oplus$, $\&$, $\otimes$, $\mathbf{1}$, $\multimap$) and type variables arising from polymorphism, we index type names by a sequence of arithmetic expressions $V[\overline{A}]\{\overline{e}\}$, existential and universal quantification over natural numbers ($\exists n.\, A$, $\forall n.\, A$) and existential and universal constraints ($?\{\phi\}.\, A$, $!\{\phi\}.\, A$). The type grammar is extended as follows (we write i for constant and n for variable natural numbers).

$$
\begin{array}{lll}
\text{Types} & A,B,C \;\; ::= & \ldots \mid V[\overline{A}]\{\overline{e}\} \mid ?\{\phi\}.\, A \mid !\{\phi\}.\, A \mid \exists n.\, A \mid \forall n.\, A \\[4pt]
\text{Arith. Exps.} & e \;\; ::= & i \mid e + e \mid e - e \mid i \times e \mid n \\[4pt]
\text{Arith. Props.} & \phi \;\; ::= & e = e \mid e > e \mid \top \mid \bot \mid \phi \wedge \phi \\
& \mid & \phi \vee \phi \mid \neg\phi \mid \exists n.\, \phi \mid \forall n.\, \phi
\end{array}
$$

To account for refinements, the typing judgment is further extended and has the form

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

where $\mathcal{V}$ are arithmetic variables n, $\mathcal{C}$ are constraints over these variables expressed as a single proposition, Δ are the linear antecedents $x_i : A_i$, P is a process expression, and $x : A$ is the linear succedent. We presuppose and maintain that all free index variables in $\mathcal{C}$, Δ, P, and A are contained among $\mathcal{V}$. As described in Section 4, the process and type definitions in signature Σ are now also parameterized by arithmetic variables. In addition we write $\mathcal{V} \;;\; \mathcal{C} \vDash \phi$ for semantic entailment (proving ϕ assuming $\mathcal{C}$) in the constraint domain where $\mathcal{V}$ contains all arithmetic variables in $\mathcal{C}$ and ϕ.

We now describe quantifiers ($\exists n.\, A$, $\forall n.\, A$) and constraints ($?\{\phi\}.\, A$, $!\{\phi\}.\, A$). An overview of the types, process expressions, their continuation, and operational meaning can be found in Table 3.

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

Table 3: Refined session types with operational description

Quantification. The provider of $(c : \exists n.\, A)$ should send a witness i along channel c and then continue as $A[i/n]$. The witness is specified by an arithmetic expression e which, since it must be closed at runtime, can be evaluated to a number i (following standard evaluation rules of arithmetic). From the typing perspective, we just need to check that the expression e denotes a natural number, using only the permitted variables in $\mathcal{V}$. This is represented with the auxiliary judgment $\mathcal{V}$; $\mathcal{C} \vdash e : \mathsf{nat}$ (implicitly proving $e \geq 0$ under constraint $\mathcal{C}$).

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

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

Statically, the $\exists L$ rule adds n to $\mathcal{V}$ ensuring that continuations Q and A can refer to variables in $\mathcal{V}, n$. Operationally, the provider sends the arithmetic expression with the continuation channel as a message that the client receives and appropriately substitutes.

$(\exists S) : \mathsf{proc}(c, \mathsf{send}\ c\ \{e\}\ ;\ P) \ \mapsto\ \mathsf{proc}(c', P[c'/c]),\ \mathsf{msg}(c, \mathsf{send}\ c\ \{e\}\ ;\ c \leftrightarrow c') \qquad (c'\ \text{fresh})$

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

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

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

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

$(\forall S) : \mathsf{proc}(d, \mathsf{send}\ c\ \{e\}\ ;\ P) \ \mapsto\ \mathsf{msg}(c', \mathsf{send}\ c\ \{e\}\ ;\ c' \leftrightarrow c),\ \mathsf{proc}(d, P[c'/c]) \qquad (c'\ \text{fresh})$

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

Constraints. Refined session types also allow constraints over index variables. As we have already seen in the examples, these critically govern permissible messages. From the message-passing perspective, the provider of $(c : ?\{\phi\}.\, A)$ should send a proof of ϕ along c and the client should receive such a proof. However, since the index domain is decidable

and future computation cannot depend on the form of the proof (what is known in type theory as *proof irrelevance*) such messages are not actually exchanged. Instead, it is the provider's responsibility to ensure that ϕ holds, while the client is permitted to assume that ϕ is true. Therefore, and in an analogy with imperative languages, we write assert c $\{\phi\}$; P for a process that *asserts* ϕ for channel c and continues with P, while assume c $\{\phi\}$; Q *assumes* ϕ and continues with Q.

Thus, the typing rules for this new type constructor are

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

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

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

Operationally, the provider creates a message containing the constraint (which simply evaluates to $\top$) that is received by the client. Since the constraints exchanged at runtime are always trivial, we can skip the communication entirely for constraints. However, in the formal semantics we still require communication for uniformity with other type constructors.

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

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

In well-typed configurations (which arise from executing well-typed processes) the constraint ϕ in these rules will always be closed and true so there is no need to check this explicitly.

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

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

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

$(!S) : \mathsf{proc}(d, \mathsf{assert}\ c\ \{\phi\}\ ;\ P) \mapsto \mathsf{msg}(c', \mathsf{assert}\ c\ \{\phi\}\ ;\ c' \leftrightarrow c),\ \mathsf{proc}(d, P[c'/c]) \quad (c'\ \text{fresh})$

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

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

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

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

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

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

```
case c ( none ⇒ assume c {0 = 0} ; P₁
       | some ⇒ assume c {0 > 0} ; impossible)
```

In implicit syntax, we can omit the **some** branch altogether and it would be reconstructed in the form shown above. Abstracting away from this example, the typing rule for impossibility simply checks that the constraints are indeed unsatisfiable

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

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

Example: Binary Numbers. As another example, consider natural numbers in binary representation. The idea is that, for example, the number 13 in binary $(1101)_2$ form is represented as a sequence of messages $(\textbf{b1}, \textbf{b0}, \textbf{b1}, \textbf{b1}, \textbf{e}, close)$ sent or received on a given channel with the least significant bit first. Here **e** represents 0 (the empty sequence of bits), while **b0** and **b1** represent bits 0 and 1, respectively.

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

We can then index binary numbers with their value. Because (linear) arithmetic contains no division operator, we express the type $\textsf{bin}\{n\}$ of binary numbers with value n using existential quantification, with the concrete syntax ?k. A for $\exists k. A$.

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

The constraint that $n > 0$ in the case of **b0** ensures the representation is unique and there are no leading zeros; the same constraint for **b1** is in fact redundant. The `examples/arith.rast` contains several examples of processes over binary numbers like addition, multiplication, predecessor, equality and conversion to and from numbers in unary form.

6. Ergometric Session Types

An important application of refinement types is complexity analysis. Prior works on resource-aware session types [DHP18b, DHP18a, DBH+21] crucially rely on arithmetic refinements to express work and time bounds. The design principle we followed is that they should be *conservative* over the basic and indexed session types, so that previously defined programs and type-checking rules do not change. In this section, we review the ergometric type system that computes work intuitively defined as the total operations executed by the system.

The key idea is that *processes store potential* and *messages carry potential*. This potential can either be consumed to perform *work* or exchanged using special messages. The type system provides the programmer with the flexibility to specify what constitutes work. Thus, programmers can choose to count the resource they are interested in, and the type system provides the corresponding upper bound. Our current examples assign unit cost to message sending operations (exempting those for index objects or potentials themselves) effectively counting the total number of "real" messages exchanged during a computation.

Two dual type constructors $\rhd^r A$ and $\lhd^r A$ are used to exchange potential. Table 4 contains a description of these types and their operational behavior. The provider of $x : \rhd^r A$ must *pay* r units of potential along x using process term $(\textsf{pay } x \ \{r\} \ ; \ P)$, and continue to

Type	Cont.	Process Term	Cont.	Description
$c : \rhd^r A$	$c : A$	$\mathsf{pay}\, c\, \{r\}\ ;\ P$	P	provider pays r potential units on channel c
		$\mathsf{get}\, c\, \{r\}\ ;\ Q$	Q	client gets r potential units on c
$c : \lhd^r A$	$c : A$	$\mathsf{get}\, c\, \{r\}\ ;\ P$	P	provider gets r potential units on channel c
		$\mathsf{pay}\, c\, \{r\}\ ;\ Q$	Q	client pays r potential units on c

Table 4: Ergometric session types with operational description

provide A by executing P. These r units are deducted from the potential stored inside the sender. Dually, the client must receive the r units of potential using the term $(\mathsf{get}\, x\, \{r\}\ ;\ Q)$ and add this to its internal stored potential. Finally, since processes are allowed to store potential, the typing judgment records the potential available to a process above the turnstile $\mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta \vdash^{q}_{\Sigma} P :: (x : A)$. We allow potential q to refer to index variables in $\mathcal{V}$ to capture variable potential. The typing rules for $\rhd^r A$ are

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash q \geq r_1 = r_2 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta \vdash^{q-r_1} P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta \vdash^{q} \mathsf{pay}\, x\, \{r_1\}\ ;\ P :: (x : \rhd^{r_2} A)}\ \rhd R$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash r_1 = r_2 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta, (x : A) \vdash^{q+r_1} Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta, (x : \rhd^{r_2} A) \vdash^{q} \mathsf{get}\, x\, \{r_1\}\ ;\ Q :: (z : C)}\ \rhd L$$

In both cases, we check that the exchanged potential in the expression and type matches $(r_1 = r_2)$, and while paying, we ensure that the sender has sufficient potential to pay. We use distinct variables r_1 and r_2 to illustrate that the process and type expressions can use syntactically different but semantically equal annotations (e.g. $r_1 = n + n, r_2 = 2 * n$). On the other hand, the receiver adds the r_1 units to its process potential.

We extend the semantic objects with work counters: $\mathsf{proc}(c, w, P)$ (resp. $\mathsf{msg}(c, w, M)$) denotes a process (resp. message) providing channel c, executing P (resp. M) and having performed work w so far. The semantics rules for $\rhd$ are expressed as follows.

$(\rhd S) : \mathsf{proc}(c, w, \mathsf{pay}\, c\, \{r\}\ ;\ P) \ \mapsto\ \mathsf{proc}(c', w, P[c'/c]),\ \mathsf{msg}(c, 0, \mathsf{pay}\, c\, \{r\}\ ;\ c \leftrightarrow c')\quad (c' \text{ fresh})$

$(\rhd C) : \mathsf{msg}(c, w', \mathsf{pay}\, c\, \{r\}\ ;\ c \leftrightarrow c'),\ \mathsf{proc}(d, w, \mathsf{get}\, c\, \{r\}\ ;\ Q)\ \mapsto\ \mathsf{proc}(d, w + w', Q[c'/c])$

The freshly created message in rule $\rhd S$ has not performed any work so far, therefore has $w = 0$. In the $\rhd C$ rule, the work performed by the message is absorbed by the receiving process. Thus, the total work done is conserved, and no work is dropped by the system. Note that even though the message was created with $w = 0$, it can interact with forwarding processes and absorb the work performed by them. Thus, when they interact with the receiver process, they may have a different work annotation w'. We follow the same approach for all the rules of operational semantics so far.

The dual type $\lhd^r A$ enables the provider to receive potential that is sent by its client. Its rules are the exact inverse of $\rhd$.

$$\frac{\mathfrak{v}\ ;\ \mathcal{C} \vDash r_1 = r_2 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta \vdash^{q+r_1} P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta \vdash^{q} \mathsf{get}\, x\, \{r_1\}\ ;\ P :: (x : \lhd^{r_2} A)}\ \lhd R$$

$$\frac{\mathfrak{v}\ ;\ \mathcal{C} \vDash q \geq r_1 = r_2 \qquad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta, (x : A) \vdash^{q-r_1} Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \mathfrak{v}\ ;\ \Delta, (x : \lhd^{r_2} A) \vdash^{q} \mathsf{pay}\, x\, \{r_1\}\ ;\ Q :: (z : C)}\ \lhd L$$

$(\lhd S) : \mathsf{proc}(d, w, \mathsf{pay}\, c\, \{r\}\ ;\ P)\ \mapsto\ \mathsf{msg}(c', 0, \mathsf{pay}\, c\, \{r\}\ ;\ c' \leftrightarrow c),\ \mathsf{proc}(d, w, P[c'/c])\quad (c'\ \mathrm{fresh})$

$(\lhd C) : \mathsf{proc}(c, w, \mathsf{get}\, c\, \{r\}\ ;\ Q),\ \mathsf{msg}(c', w', \mathsf{pay}\, c\, \{r\}\ ;\ c' \leftrightarrow c)\ \mapsto\ \mathsf{proc}(c', w + w', Q[c'/c])$

We use a special expression $\mathsf{work}\,\{r\}\ ;\ P$ to *perform work*. Usually, work actions are inserted by the Rast compiler based on a cost model selected by the programmer, such as paying one erg just before every send operation. The programmer can also select a model where all operations are free and manually insert calls to $\mathsf{work}\,\{r\}$. An example of this is given in the file `examples/linlam-reds.rast` that counts the number of reductions necessary for the evaluation of an expression in the linear λ-calculus (described in Section 10).

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

At runtime, executing the $\mathsf{work}\,\{r\}$ construct increments the work counter by r.

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

Work is *precise*, that is, before terminating a process must have 0 potential, which can be achieved by explicitly consuming any remaining potential.

Example: Ergometric Queue. We have already seen the ergometric types of queues as a bucket brigade in the introduction. We show it now in concrete syntax, where `<{p}|` receives potential `p`.

```
type queue[A]{n} = &{ins : <{2*n}| A -o queue[A]{n+1},
                      del : <{2}| +{none : ?{n = 0}. 1,
                                    some : ?{n > 0}. A * queue[A]{n-1}}}

decl empty[A] : . |- (q : queue[A]{0})
decl elem[A]{n} : (x : A) (r : queue[A]{n}) |- (q : queue[A]{n+1})
```

Interestingly, the exact code of Listing 1 will check against this more informative type (see file `examples/list-work.rast`). The cost model will insert the appropriate $\mathsf{work}\,\{r\}$ action and reconstruction will insert the actions to pay and get potential.

For a queue implemented internally as two stacks we can perform an amortized analysis. Briefly, the queue process maintains two lists: one (*in*) to store messages when they are enqueued, and a reversed list (*out*) from which they are dequeued. When the client wishes to dequeue an element and the *out* list is empty, the provider reverses the *in* list to serve as the new *out* list. A careful analysis shows that if this data structure is used linearly, both insert and delete have constant amortized time. More specifically we obtain the type

```
type queue[A]{n} = &{enq : <{6}| A -o queue[A]{n+1},
                      deq : <{4}| +{none : ?{n = 0}. 1,
                                    some : ?{n > 0}. A * queue[A]{n-1}}}
```

The program can be found in the file `examples/list-work.rast` in the repository.

7. Temporal Session Types

Rast also supports *temporal modalities next* ($\bigcirc A$), *always* ($\Box A$), and *eventually* ($\Diamond A$), interpreted over a linear model of time. To model computation time, we use the syntactic form delay which advances time by one tick. A particular cost semantics is specified by

taking an ordinary, non-temporal program and adding delays capturing the intended cost. For example, if only the blocking operations should cost one unit of time, a delay is added before the continuation of every receiving construct. For type checking, the delay construct subtracts one $\bigcirc$ operator from every channel it refers to. We denote consuming t units on the left of the context using $[A]_L^{-t}$, and on the right by $[A]_R^{-t}$. Briefly, $[\bigcirc^t A]_L^{-t} = [\bigcirc^t A]_R^{-t} = A$. As we will explain soon, consuming time units on the left and the right differ due to the $\square$ and $\lozenge$ modalities.

$$\frac{\mathcal{V} \;;\; \mathcal{C} \vDash t \geq 0 \qquad \mathcal{V} \;;\; \mathcal{C} \;;\; \mathfrak{v} \;;\; [\Delta]_L^{-t} \vdash^{\underline{q}} Q :: (x : [A]_R^{-t})}{\mathcal{V} \;;\; \mathcal{C} \;;\; \mathfrak{v} \;;\; \Delta \vdash^{\underline{q}} \mathsf{delay}\,(t) \;;\; P :: (x : A)} \;\bigcirc LR$$

To express the semantics, we now use $\mathsf{proc}(c, w, t, P)$ to denote a process P at local clock t. This local clock advances by r as the process executes a $\mathsf{delay}\,(r)$.

$(\bigcirc C) : \mathsf{proc}(c, w, t, \mathsf{delay}\,(r) \;;\; P) \;\mapsto\; \mathsf{proc}(c, w, t + r, P)$

Always A***.*** A process providing $x : \square A$ promises to be available at any time in the future, including now. When the client would like to use this provider it (conceptually) sends a message now! along x and then continues to interact according to type A.

A process P providing $x : \square A$ must be able to wait indefinitely. But this is only possible if all the channels that P uses can also wait indefinitely. This is enforced in the rule by the condition $\Delta\ \mathsf{delayed}^{\square}$ which requires each antecedent to have the form $y_i : \bigcirc^{n_i} \square B_i$.

$$\frac{\Delta\ \mathsf{delayed}^{\square} \quad \Delta \vdash P :: (x : A)}{\Delta \vdash (\mathsf{when?}\,(x) \;;\; P) :: (x : \square A)} \;\square R \qquad\qquad \frac{\Delta, x : A \vdash Q :: (z : C)}{\Delta, x : \square A \vdash (\mathsf{now!}\,(x) \;;\; Q) :: (z : C)} \;\square L$$

The corresponding semantics rules are

$(\square S) : \mathsf{proc}(d, w, t, \mathsf{now!}\,(c) \;;\; P) \;\mapsto\; \mathsf{msg}(c', 0, t, \mathsf{now!}\,(c) \;;\; c' \leftrightarrow c),\ \mathsf{proc}(d, w, t, P[c'/c])$

$(\square C) : \mathsf{proc}(c, w, s, \mathsf{when?}\,(c) \;;\; Q), \mathsf{msg}(c', w', t, \mathsf{now!}\,(c) \;;\; c' \leftrightarrow c) \;\mapsto$
$$\mathsf{proc}(c', w + w', t, Q[c'/c]) \qquad (s \leq t)$$

Eventually A***.*** The dual of $\square A$ is $\lozenge A$. A process providing $\lozenge A$ promises to provide A eventually. When a process offering $x : \lozenge A$ is ready, it will send a now! message along x and then continue at type A. Conversely, the client of $x : \lozenge A$ will have to be ready and waiting for the now! message to arrive along x and then continue at type A. We use $(\mathsf{when?}\,(c) \;;\; Q)$ for the corresponding client. The typing rules for now! and when? are somewhat subtle.

$$\frac{\Delta \vdash P :: (x : A)}{\Delta \vdash (\mathsf{now!}\,(x) \;;\; P) :: (x : \lozenge A)} \;\lozenge R \qquad \frac{\Delta\ \mathsf{delayed}^{\square} \quad \Delta, x : A \vdash Q :: (z : C) \quad A\ \mathsf{delayed}^{\lozenge}}{\Delta, x{:}\lozenge A \vdash (\mathsf{when?}\,(x) \;;\; Q) :: (z : C)} \;\lozenge L$$

The predicate $C\ \mathsf{delayed}^{\lozenge}$ means that C must have the form $\bigcirc^n \lozenge C'$ (for some n) requiring that C may be delayed a fixed finite number of time steps and then must be allowed to communicate at an arbitrary time in the future. The semantics rules for $\lozenge$ are exact inverse of $\square$.

$(\lozenge S) : \mathsf{proc}(c, w, t, \mathsf{now!}\,(c) \;;\; P) \;\mapsto\; \mathsf{proc}(c', w, t, P[c'/c]),\ \mathsf{msg}(c, 0, t, \mathsf{now!}\,(c) \;;\; c \leftrightarrow c')$

$(\lozenge C) : \mathsf{msg}(c, 0, t, \mathsf{now!}\,(c) \;;\; c \leftrightarrow c'),\ \mathsf{proc}(d, w, s, \mathsf{when?}\,(c) \;;\; P) \;\mapsto$
$$\mathsf{proc}(d, w + w', t, P[c'/c]) \qquad (s \leq t)$$

Type	Cont.	Process Term	Cont.	Description
$c : \Diamond A$	$c : A$	now! (c) ; P	P	provider sends now! on channel c
		when? (c) ; Q	Q	client waits for now! on c
$c : \Box A$	$c : A$	when? (c) ; P	P	provider waits for now! on channel c
		now! (c) ; Q	Q	client sends now! on c
$c : \bigcirc^r A$	$c : A$	delay (r) ; P	P	provider delays for r time units

Table 5: Temporal session types with operational description

Table 5 provides a formal description of the temporal types. Since all temporal operators ultimately model time, they interact with each other and the temporal displacement operator used in $\bigcirc LR$ rule needs to be generalized as described below. We define $[A]^0 = A$ and $[A]^{-(t+1)} = [[A]^{-1}]^{-t}$. Below S denotes a non-temporal type. When the displacement is undefined, the $\bigcirc LR$ rule cannot be applied. More details can be found in prior work [DHP18a].

$$
\begin{array}{lclcccll}
[\bigcirc A]_L^{-1} &=& A & \qquad [\bigcirc A]_R^{-1} &=& A & \qquad [x : A]_L^{-1} &=& x : [A]_L^{-1} \\
[\Box A]_L^{-1} &=& \Box A & \qquad [\Box A]_R^{-1} &=& \textit{undefined} & \qquad [x : A]_R^{-1} &=& x : [A]_R^{-1} \\
[\Diamond A]_L^{-1} &=& \textit{undefined} & \qquad [\Diamond A]_R^{-1} &=& \Diamond A & \qquad [\cdot]_L^{-1} &=& \cdot \\
[S]_L^{-1} &=& \textit{undefined} & \qquad [S]_R^{-1} &=& \textit{undefined} & \qquad [\Delta, \Delta']_L^{-1} &=& [\Delta]_L^{-1}, [\Delta']_L^{-1}
\end{array}
$$

Example: Temporal Queue. We have already foreshadowed the temporal type of a queue, implemented as a bucket brigade. We show it now in concrete syntax, where () is the $\bigcirc$ modality and [] represents $\Box$. We also show the types of the `empty` and `elem` processes (see file `examples/time.rast`).

```
type queue[A]{n} = [] {enq : () A -o ()()()queue[A]{n+1},
                       deq : ()+{none: () ?{n = 0}. 1,
                                 some: () ?{n > 0}. A * ()queue[A]{n-1}}}

decl empty[A] : . |- (q : ()()queue[A]{0})
decl elem[A]{n} : (x : A) (r : ()()queue[A]{n}) |- (q : queue[A]{n+1})
```

Because Rast currently does not have reconstruction for time we have to update the program with the five temporal actions presented in this section (two instances of `delay`, two of `when`, and one of `now`). A key observation here is that in the case of `elem` the process r does not need to be ready instantaneously, but can be ready after a delay of 2 ticks, because that is how long it takes to receive the **ins** label and the element along q. This slack is also reflected in the type of `empty` because it becomes then back of a new element when the end of the queue is reached.

8. Subtyping

A late addition to Rast was the introduction of subtyping as a generalization of type equality. Declaratively, in the system of basic session types (Section 3), we have just two rules to

express subtyping

$$\frac{\Delta \vdash P :: (x : A) \quad A \leq A'}{\Delta \vdash P :: (x : A')} \ \mathsf{sub}R \qquad\qquad \frac{\Delta, (x : A') \vdash P :: (z : C) \quad A \leq A'}{\Delta, (x : A) \vdash P :: (z : C)} \ \mathsf{sub}L$$

plus the rules for subtyping $A \leq B$. The latter follow Gay and Hole [GH05] by defining it coinductively as the largest type simulation. This includes the standard notion of depth and width subtyping for internal and external choice, except that our relation happens to be exactly reversed from theirs due to our intuitionistic framework. We introduce fresh internal definitions for all intermediate type subexpressions, which allows us a practically efficient implementation of their algorithm that incrementally constructs this simulation. For type-checking purposes, we can restrict the uses of subtyping to forwarding (rule id), spawn (rule def), and channel passing (rules $\otimes R$ and $\multimap L$). Our implementation of the linear λ-calculus in Section 10 exploits subtyping by observing that all values (type $\mathtt{val}$) are also expressions (type $\mathtt{exp}$), that is, $\mathtt{val} \leq \mathtt{exp}$. This means we can pass a channel of type $\mathtt{val}$ to a process expecting a channel of type $\mathtt{exp}$, which is used in the implementation of β-reduction.

The algorithms for subtyping become increasingly more complicated with the addition of indexed types [DP20b] (which is in fact undecidable) and nested polymorphic types (the subject of ongoing research, generalizing type equality [DDMP21]). Nevertheless, the basic structure of incrementally constructing a simulation remains intact, just recognizing that a new pair is already in the partial simulation becomes more difficult. Our example suite shows that even in the undecidable cases, type checking (including subtyping) does not become a significant bottleneck.

9. Implementation

We have implemented a prototype for Rast in Standard ML (8100 lines of code). This implementation contains a lexer and parser (1200 lines), reconstruction engine (900 lines), an arithmetic solver (1200 lines), a type checker (2500 lines), pretty printer (400 lines), and an interpreter (200 lines). The source code is well-documented and available open-source [DDP19].

Syntax. Table 6 describes the syntax for Rast programs. Each row presents the abstract and concrete representation of a session type, and its corresponding providing expression. A program contains a series of mutually recursive type and process declarations and definitions.

```
type v[a]...{n}... = A
decl f[a]...{n}... : (x1 : A1) ... (xn : An) |- (x : A)
proc x <- f [a]...{n}... x1 ... xn = P
```

Listing 2: Top-Level Declarations

The first line is a *type definition*, where v is the name with type parameters $\overline{a}$ and index variables $\overline{n}$ and A is its definition. The second line is a *process declaration*, where f is the process name, $(x_1 : A_1) \ldots (x_n : A_n)$ are the used channels and corresponding types, while the provided channel is x of type A. Finally, the last line is a *process definition* for the same process f defined using the process expression P. In addition, f can be parameterized by type variables $\overline{a}$ and index variables $\overline{n}$. We use a hand-written lexer and shift-reduce parser to read an input file and generate the corresponding abstract syntax tree of the program.

Abstract Types	Concrete Types	Abstract Syntax	Concrete Syntax
$\oplus\{l : A, \ldots\}$	`+{l : A, ...}`	$x.k$	`x.k`
$\&\{l : A, \ldots\}$	`&{l : A, ...}`	$\text{case } x\ (\ell \Rightarrow P)_{\ell \in L}$	`case x (l => P | ...)`
$A \otimes B$	`A * B`	$\text{send } x\ w$	`send x w`
$A \multimap B$	`A -o B`	$y \leftarrow \text{recv } x$	`y <- recv x`
$\mathbf{1}$	`1`	$\text{close } x$	`close x`
		$\text{wait } x$	`wait x`
$\exists a.\, A$	`?[a]. A`	$\text{send } x\ [B]$	`send x [B]`
$\forall a.\, A$	`![a]. A`	$[a] \leftarrow \text{recv } x$	`[a] <- recv x`
$\exists n.\, A$	`?n. A`	$\text{send } x\ \{e\}$	`send x {e}`
$\forall n.\, A$	`!n. A`	$\{n\} \leftarrow \text{recv } x$	`{n} <- recv x`
$?\{\phi\}.\, A$	`?{phi}. A`	$\text{assert } x\ \{\phi\}$	`assert x {phi}`
$!\{\phi\}.\, A$	`!{phi}. A`	$\text{assume } x\ \{\phi\}$	`assume x {phi}`
$\triangleright^r A$	`|{r}> A`	$\text{pay } x\ \{r\}$	`pay x {r}`
$\triangleleft^r A$	`<{r}| A`	$\text{get } x\ \{r\}$	`get x {r}`
$\bigcirc^t A$	`({t}) A`	$\text{delay}\ (t)$	`delay {t}`
$\Box A$	`[] A`	$\text{when?}\ (x)$	`when x`
$\Diamond A$	`<> A`	$\text{now!}\ (x)$	`now x`
$V[\overline{A}]\{\overline{e}\}$	`V[A]...{e}...`		
		$x \leftrightarrow y$	`x <-> y`
		$x \leftarrow f\ x_1 \ldots x_n$	`x <- f x1 ... xn`

Table 6: Abstract and Corresponding Concrete Syntax for Types and Expressions

The reason to use a hand-written parser instead of a parser generator is to anticipate the most common syntax errors that programmers make and respond with the best possible error messages.

Validity Checking. Once the program is parsed and its abstract syntax tree is extracted, we perform a validity check on it. We check that all index refinements, potentials, and delay operators are non-negative. We also check that all index expressions are closed with respect to the the index variables in scope, and similarly for type expressions. To simplify and improve the efficiency of the subtyping algorithm, we also assign internal names to type subexpressions [DP20c, DDMP21] parameterized over their free type and index variables. These internal names are not visible to the programmer.

Cost Model. The cost model defines the execution cost of each construct. Since our type system is parametric in the cost model, we allow programmers to specify the cost model they want to use. Although programmers can create their own cost model (by inserting work or delay expressions in the process expressions), we provide three custom cost models: send, recv, and recvsend. If we are analyzing work (resp. time), the send cost model inserts a work{1} (resp. delay{1}) before (resp. after) each send operation. Similarly, recv model assigns a cost of 1 to each receive operation. The recvsend cost model assigns a cost of 1 to each send and receive operation.

Reconstruction and Type Checking. The programmer can use a flag in the program file to indicate whether they are using *explicit* or *implicit* syntax. If the syntax is explicit, the reconstruction engine performs no program transformation. However, if the syntax is implicit, we use the implicit type system to approximately type-check the program. Once completed, we use the forcing calculus, introduced in prior work [DP20c] to insert assert, assume, pay, get and work constructs. The core idea here is simple: insert assume or get constructs eagerly, i.e., as soon as available on a channel, and insert assert and pay lazily, i.e., just before communicating on that channel. The forcing calculus proves that this reconstruction technique is sound and complete in the absence of certain forms of quantifier alternations (which are checked before reconstruction is performed). We only perform reconstruction for proof constraints and ergometric types; reconstruction of type and index quantifiers and temporal constructs is left to future work.

The implementation takes some care to provide constructive and precise error messages, in particular as session types (not to mention arithmetic refinements, ergometric types, and temporal types) are likely to be unfamiliar. One technique is staging: first check approximate type correctness, ignoring index, ergometric, and temporal types, and only if that check passes perform reconstruction and strict checking of type. Another particularly helpful technique has been *type compression*. Whenever the type checker expands a type $V[\overline{A}]\{\overline{e}\}$ with $V[\overline{\alpha}]\{\overline{n}\} = B$ to $B[\overline{A}/\overline{\alpha}, \overline{e}/\overline{n}]$, we record a reverse mapping back to $V[\overline{A}]\{\overline{e}\}$. When printing types for error messages this mapping is consulted, and complex types may be compressed to much simpler forms, greatly aiding readability of error messages. This is feasible in part because all intermediate subexpressions have an explicit (internal) definition, simplifying the lookup. Finally, our implementation uses a bi-directional [DP20c, DDMP21] type checking algorithm which reconstructs intermediate types for each channel. This helps localize the source of the error message as the program point where reconstruction fails. We designed the abstract syntax tree to also contain the relevant source code location information which is utilized while generating the error message.

Subtyping. At the core of type checking lies subtyping, defined coinductively [GH05]. In the presence of arithmetic refinements, subtyping and also type equality are undecidable, but we have found what seems to be a practical approximation [DP20b, DDMP21], incrementally constructing a simulation closed under reflexivity. The data structures are rather straightforward, emphasizing simplicity over efficiency since subtyping tends to be fast in practice. There are several places where the translation from the underlying theory to the implementation are not straightforward. After the file has been read and the validity of types has been verified, we compute the *variance* of all type constructors in all type arguments (covariant, contravariant, nonvariant, or bivariant) by a greatest fixed point computation, starting from the assumption that all arguments are nonvariant. The variance information is then used when determining if a new subtyping goal is implied by the partial simulation constructed so far. The algorithm employs syntactic *matching* (allowing the type variables in the simulation to be instantiated) without consulting the partial simulation again (which could lead to nontermination). It also calls upon the constraint solver to determine if the constrained pairs in the partial simulation contain enough information to entail the constraints in the goal. The theory establishing soundness of this subtyping algorithm in the presence of nested polymorphic types under their coinductive interpretation is currently under development.

This algorithm always terminates because we impose a bound on how many distinct inequalities $V[\overline{A}]\{\overline{e}\} \leq V'[\overline{A'}]\{\overline{e'}\}$ we consider for any given pair V, V'. Note that variables also include the internal names created during elaboration. However, it may fail to establish an inequality if the coinductive invariant is not general enough. Rast therefore allows the programmer to assert an arbitrary number of additional type equalities or inequalities with the constructs

```
eqtype V[A]...{e}... = V'[A']...{e'}...
eqtype V[A]...{e}... <= V'[A']...{e'}...
```

These are abstracted over their free variables, then checked one by one, assuming all other asserted equalities and inequalities. The default construction of the simulation is currently strong enough so that this feature has not been needed for any of our standard examples.

Arithmetic Solver. To determine the validity of arithmetic propositions that is used by our refinement layer, we use a straightforward implementation of Cooper's decision procedure [Coo72] for Presburger arithmetic. We found a small number of optimizations were necessary, but the resulting algorithm has been quite efficient and robust.

(1) We eliminate constraints of the form $x = e$ (where x does not occur in e) by substituting e for x in all other constraints to reduce the total number of variables.
(2) We exploit that we are working over natural numbers so all solutions have a natural lower bound, i.e., 0.

We also extend our solver to handle non-linear constraints. Since non-linear arithmetic is undecidable, in general, we use a normalizer which collects coefficients of each term in the multinomial expression.

(1) To check $e_1 = e_2$, we normalize $e_1 - e_2$ and check that each coefficient of the normal form is 0.
(2) To check $e_1 \geq e_2$, we normalize $e_1 - e_2$ and check that each coefficient is non-negative.
(3) If we know that $x \geq c$, we substitute $y + c$ for x in the constraint that we are checking with the knowledge that the fresh $y \geq 0$.
(4) We try to find a quick counterexample to validity by plugging in 0 and 1 for the index variables.

If the constraint does not fall in the above two categories, we print the constraint and trust that it holds. A user can then view these constraints manually and confirm their validity. At present, all of our examples pass without having to trust unsolvable constraints with our set of heuristics beyond Presburger arithmetic.

Interpreter. The current version of the interpreter pursues a sequential schedule following a prior proposal [PP20]. We only execute programs that have no free type or index variables and only one externally visible channel, namely the one provided. When the computation finishes, the messages that were asynchronously sent along this distinguished channel are shown, while running processes waiting for input are displayed simply as a dash '-'.

The interpreter is surprisingly fast. For example, using a linear prime sieve to compute the status (prime or composite) of all numbers in the range $[2, 257]$ takes 27.172 milliseconds using MLton during our experiments (see machine specifications below).

Module	iLOC	eLOC	#Defs	R (ms)	T (ms)
arithmetic	395	619	29	0.959	5.732
integers	90	125	8	0.488	0.659
linlam	88	112	10	0.549	1.072
list	341	642	37	3.164	4.637
primes	118	164	11	0.289	4.580
segments	48	76	8	0.183	0.225
ternary	270	406	20	0.947	140.765
theorems	79	156	13	0.182	1.095
tries	243	520	13	2.122	6.408
Total	**1672**	**2820**	**149**	**8.883**	**165.173**

Table 7: Case Studies

10. FURTHER EXAMPLES

We present several different kinds of examples from varying domains illustrating different features of the type system and algorithms. Table 7 describes the results: iLOC describes the lines of source code in implicit syntax, eLOC describes the lines of code after reconstruction (which inserts implicit constructs), #Defs shows the number of process definitions, R (ms) and T (ms) show the reconstruction and type-checking time in milliseconds respectively. Note that reconstruction is faster than type-checking since reconstruction does not involve solving any arithmetic propositions. The experiments were run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory.

(1) **arithmetic**: natural numbers in unary and binary representation indexed by their value and processes implementing standard arithmetic operations.
(2) **integers**: an integer counter represented using two indices x and y with value $x - y$.
(3) **linlam**: expressions in the linear λ-calculus indexed by their size.
(4) **list**: lists indexed by their size, and some standard operations such as *append, reverse, map, fold*, etc. Also provides and implementation of stacks and queues using lists.
(5) **primes**: the sieve of Eratosthenes to classify numbers as prime or composite.
(6) **segments**: type $\mathsf{seg}[n] = \forall k.\mathsf{list}[k] \multimap \mathsf{list}[n+k]$ representing partial lists with a constant-work append operation.
(7) **ternary**: natural numbers and integers represented in balanced ternary form with digits $0, 1, -1$, indexed by their value, and a few standard operations on them. This example is noteworthy since it is the only one stressing the arithmetic decision procedure.
(8) **theorems**: processes representing valid circular [DP19] proofs of simple theorems such as $n(k + 1) = nk + n, n + 0 = n, n * 0 = 0$, etc.
(9) **tries**: a trie data structure to store multisets of binary numbers, with constant amortized work insertion and deletion verified with ergometric types.

We highlight interesting examples from some case studies showcasing the invariants that can be proved using arithmetic refinements and nested polymorphism.

Linear λ-Calculus. We implemented the linear λ-calculus with evaluation (weak head normalization) of terms. We use higher-order abstract syntax, representing linear abstraction in the object language by a process receiving a message corresponding to its argument. This is inspired by Milner's call-by-name encoding of the (nonlinear) λ-calculus in the

π-calculus [Mil92]. We expand on it by considering typing in metalanguage, and also provide a static analysis of size of normal forms and number of reductions.

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

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

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

Rast can infer that val is a subtype of exp. We can derive constructors *apply* for expressions and *lambda* for values (we do not need the corresponding constructor for expressions).

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


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

As a simple example, we follow the representation of a combinator that swaps the arguments to a function.

```
(* swap = \f. \x. \y. (f y) x *)
decl swap : . |- (e : exp)
proc e <- swap =
  e.lam ; f <- recv e ;
  e.lam ; x <- recv e ;
  e.lam ; y <- recv e ;
  fy <- apply f y ;
  e <- apply fy x
```

Evaluation is now the following very simple process.

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

If e sends a **lam** label, we just rebuild the expression as a value. If e sends an **app** label then e represents a linear application $e_1\,e_2$ and the continuation has type $\mathsf{exp} \otimes \mathsf{exp}$. This means we *receive* a channel representing e_1 and the continuation (still called e) behaves like e_2. We note this with a comment in the source. We then evaluate e_1 which exposes a λ-expression along the channel v_1. We send e along v_1, carrying out the reduction via communication. The result of this (still called v_1) is evaluated to yield the final value v. This program is available in the repository at `examples/linlam.rast`.

We would now like to prove that the value of a linear λ-expression is smaller than or equal to the original expression. At the same time we would like to rule out a class

of so-called *exotic terms* in the representation, which are possible due to the presence of recursion in the metalanguage. We achieve this by indexing the types exp and val with their *size*. For an application, this is easy: the size is one more than the sum of the sizes of the subterms.

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

The size $n_2 + 1$ of a λ-expression is one more than the size n_2 of its body, but what is that in our higher-order representation? The body of a linear function takes an expression of size n_1 and then behaves like an expression of size $n_1 + n_2$. Solving for n_2 then gives use the following type definitions and types for the constructor processes.

```
type exp{n} = +{lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1},
                app : ?n1. ?n2. ?{n = n1+n2+1}. exp{n1} * exp{n2}}

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

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

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

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

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

```
proc v <- eval{n} e =
  case e ( lam => send v {n} ;
                  v <- lambda{n-1} e
         | app => {n1} <- recv e ;
                  {n2} <- recv e ;
                  e1 <- recv e ;
                  v1 <- eval{n1} e1 ;
                  {k2} <- recv v1 ;
                  case v1 ( lam => send v1 {n2} ;
                                   send v1 e ;
                                   v2 <- eval{n2+k2-1} v1 ;
                                   {l} <- recv v2 ;
                                   send v {l} ; v <-> v2))
```

Type-checking now verifies that if evaluation terminates, the resulting value is smaller than the expression (or of equal size). The repository contains the implementation in the file examples/linlam-size.rast.

Remarkably, ergometric session types can bound the number of reductions using an amortized analysis of work! For this, we assign 1 erg (unit of potential) to each λ-expression. Our cost model is that all operations are free, except the equivalent of a β-reduction which

costs 1 erg. Because transfer of potential is reconstructed, the program is very close to the original, size-free program.

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


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

The |> in the **lam** branch denotes a unit potential associated with an λ-expression. The definitions for *apply* and *lambda* are unchanged.

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


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

The |{1}- denotes a unit potential on the *lambda* process. This potential is consumed to send 1 erg after sending the **lam** label. Note that reconstruction enables the programmer to skip such work constructs.

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

Type-checking here verifies that the reduction of a given expression with n λ-abstractions to a value performs at most $k < n$ β-reductions, with a potential of $n - k$ for further reductions remaining in the value. This means that there are exactly $n - k$ λ-abstractions remaining in the result. The only change in the program is the work construct to realize the cost model of only counting the reductions. Rast allows programmers to choose from one of the existing cost models or provide their own custom cost model. Again, the remaining constructs (e.g. paying and getting potential) are automatically inserted by the reconstruction engine. The full code can be found in the file `examples/linlam-reds.rast` in the repository.

Trie Data Structure. We now implement multisets of natural numbers (in binary form), as introduced at the end of Section 5. One of the key questions is how to maintain linearity in the design of the data structure and interface. For example, should we be able to delete an element from the trie, not knowing a priori if it is even *in* the trie? To avoid exceedingly complex types to account for these situations, the process maintaining a trie offers an interface with two operations: insert (label **ins**) and delete (label **del**). We index the type `trie{n}` with the number of elements in the trie, so inserting an element always increases n by 1. If the element is already present, we just add 1 to its multiplicity. Deleting an element actually removes all copies of it and returns its multiplicity m. If the element is *not* in the

trie, we just return a multiplicity of $m = 0$. In either case, the trie contains $n - m$ elements afterwards.

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

This type requires universal quantification over k, (written `!k`) which is the value of the number inserted into or deleted from the trie on each interaction (which is arbitrary).

The basic idea of the implementation is that each bit in the number `x : bin{k}` addresses a subtrie: if it is **b0** we descend into the left subtrie, if it is **b1** we descent into the right subtrie. If it is **e** we have found (or constructed) the node corresponding to x and we either increase its multiplicity (for insert), or respond with its multiplicity and set the new multiplicity to zero (for delete). We have two forms of processes: a *leaf* with zero elements and an interior node with $n_0 + m + n_1$ elements (where n_0 and n_1 and the number of elements in the left and right subtries, and m is the multiplicity of the number corresponding to this node in the trie).

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

The code is somewhat repetitive, so we only show the code for inserting an element into an interior node.

```
proc t <- node{n0}{m}{n1} l c r =
  case t (
    ins => {k} <- recv t ;
           x <- recv t ;
           case x ( b0 => {k'} <- recv x ;
                          l.ins ; send l {k'} ; send l x ;
                          t <- node{n0+1}{m}{n1} l c r
                  | b1 => {k'} <- recv x ;
                          r.ins ; send r {k'} ; send r x ;
                          t <- node{n0}{m}{n1+1} l c r
                  | e => wait x ;
                         c.inc ;
                         t <- node{n0}{m+1}{n1} l c r )
  | del => ...)
```

What does type-checking verify in this case? It shows that the number of elements in the trie increases and decreases as expected for each insert and delete operation. On the other hand, it does not verify that the *correct* multiplicities are incremented or decremented, which is beyond the reach of the current type system. The source code is available at `examples/trie-work.rast`.

Expression Server. Nested polymorphism in Rast can also be employed to ensure interesting invariants. As an example, we adapt the example of an arithmetic expression from prior work on context-free session types [TV16]. The type of the server is defined as

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

```
type tm[K] = +{ const : bin * K,
                add : tm[tm[K]],
                double : tm[K] }
```

The type `bin` represents a binary number as before, except we dropped the index in order to focus on nested polymorphism.

An arithmetic term, parameterized by continuation type K can have one of three forms: a constant, the sum of two terms, or the double of a term. Consequently, the type `tm[K]` ensures that a process providing `tm[K]` is a *well-formed term*: it either sends the `const` label followed by sending a constant binary number of type `bin` and continues with type K; or it sends the `add` label and continues with `tm[tm[K]]`, where the two terms denote the two summands; or it sends the `double` label and continues with `tm[K]`. In particular, the continuation type `tm[tm[K]]` in the `add` branch enforces that the process must send exactly two summands for sums.

As a first illustration, consider two binary constants a and b, and suppose that we want to create the expression $a + 2b$. We can issue commands to the expression server in a *prefix notation* to obtain $a + 2b$, as shown in the following `exp[K]` process, which is parameterized by a continuation type K.

```
decl exp[K] : (a : bin) (b : bin) (k : K) |- (e : tm[K])
proc e <- exp[K] a b k =
  e.add ;                  % (a:bin) (b:bin) (k:K) |- (e : tm[tm[K]])
  e.const ; send e a ;     % (b:bin) (k:K) |- (e : tm[K])
  e.double ;               % (b:bin) (k:K) |- (e : tm[K])
  e.const ; send e b ;     % (k:K) |- (e : K)
  e <-> k
```

In prefix notation, $a + 2b$ would be written $+ (a) (2\ b)$, which is exactly the form followed by the `exp` process: The process sends `add`, followed by `const` and the number `a`, followed by `double`, `const`, and `b`. Finally, the process continues at type K by forwarding `k` to `e`. To assist the reader, we describe the intermediate typing contexts on the right that are automatically reconstructed by the Rast type checker.

The type `tm[K]` ensures that any process offering it must be a *well-formed term*. In particular, the continuation type `tm[tm[K]]` in the `add` branch enforces that once the `add` command is issued, the process must send exactly two summands. Similar properties hold of the `const` and `double` branches.

To evaluate a term, we can define an `eval` process, parameterized by type K:

```
decl eval[K] : (t : tm[K]) |- (v : bin * K)
```

The `eval` process uses channel `t : tm[K]` as argument, and offers `v : bin * K`. The process evaluates term `t` and sends its binary value along `v`.

```
decl eval[K] : (t : tm[K]) |- (v : bin * K)
proc v <- eval[K] t =
  case t (
    const =>                 % (t : bin * K) |- (v : bin * K)
      n <- recv t ;          % (n : bin) (t : K) |- (v : bin * K)
      send v n ; v <-> t
  | add =>                   % (t : tm[tm[K]]) |- (v : bin * K)
```

```
        v1 <- eval[tm[K]] t ; % (v1 : bin * tm[K]) |- (v : bin * K)
        n1 <- recv v1 ;       % (n1 : bin) (v1 : tm[K]) |- (v : bin * K)
        v2 <- eval[K] v1 ;    % (n1 : bin) (v2 : bin * K) |- (v : bin * K)
        n2 <- recv v2 ;       % (n1 : bin)(n2 : bin)(v2 : K) |- (v : bin * K)
        n <- plus n1 n2 ;     % (n : bin) (v2 : K) |- (v : bin * K)
        send v n ; v <-> v2
    | double =>               % (t : tm[K]) |- (v : bin * K)
        v1 <- eval[K] t ;     % (v1 : bin * K) |- (v : bin * K)
        n1 <- recv v1 ;       % (n1 : bin) (v1 : K) |- (v : bin * K)
        n <- double n1 ;      % (n : bin) (v1 : K) |- (v : bin * K)
        send v n ; v <-> v1
    )
```

Intuitively, the process evaluates term `t` and sends its binary value along `v`. If `t` is a constant, then `eval[K]` receives the constant n, sends it along v and forwards.

The interesting case is the `add` branch. We evaluate the first summand by spawning a new `eval[K]` process on `t`. Note that since the type of `t` (indicated on the right) is `tm[tm[K]]` and hence, the recursive call to `eval` is at parameter `tm[K]`. We store the value of the first summand at channel `n1 : bin`. Then, we continue to evaluate the second summand by calling `eval[K]` on `t` again and storing its value in `n2 : bin`. Finally, we add `n1` and `n2` by calling the `plus` process (which has a straightforward definition), and send the result `bin` along `v`. We follow a similar approach for the `double` branch.

11. Related Work

The literature on session types is by now vast, so we focus our review of related work on *binary session types* (rather than multiparty session types) with *implementations* (rather than theoretical foundations). Among them, we can distinguish those that offer a library or embedding to a pre-existing language, and those that may be considered stand-alone language designs.

Libraries. There are a number of libraries for session types. Such libraries tend to have a very different flavor from Rast because they focus on practical usability in the context of a general-purpose language. As such, the challenge usually is how to encode session types so programs can be statically checked against them and how to achieve the expected dynamic behavior. Among them we find libraries for Haskell [LM16a, OY16], Scala [SY16], OCaml [Pad17], and Rust [JML15]. Noteworthy is the embedding of session types in ATS [XRWB16] because, unlike the others, ATS supports arithmetic indexing similar to Rast. The most recent library for Rust [CB20] is perhaps the closest to Rast in that it extends the exact basic system of session types from Section 3 with shared types [BP17]. While some of these libraries permit limited polymorphism, none of them support ergometric or temporal types.

Languages. Designing complete languages like Rast frees the researcher from the limitations and idiosyncrasies of the host language as they explore the design space. A relatively early effort was the object-oriented language MOOL [Vas11] which distinguishes linear and nonlinear channels.

A different style of language is SePi [BMV12, FV13] based on the π-calculus. It supports *linear* refinements in terms of uninterpreted propositions (which may reference integers) in addition to assert and assume primitives on them. They are not intended to capture internal properties of data structures of processes; instead, they allow the programmer to express some security properties.

The CO_2 middleware language [BCPP15, BCM$^+$15] supports *binary timed session types.* The notion of time here is *external.* As such, it does not measure work or span based on a cost model like Rast, but specifies interaction time windows for processes that can be enforced dynamically via monitors.

Concurrent C0 [WPP16] is an implementation of linear and shared session types as an extension of C0, a small type-safe and memory-safe subset of C. It integrates the basic session types from Section 3 with shared session type [BP17] in the context of an imperative language. Relatedly, the Nomos language [DBH$^+$21] integrates linear and shared ergometric session types with a functional language to aid smart contract programming. Although Nomos does not support temporal types and polymorphism, it embeds a linear programming solver to automatically infer the exact potential annotations.

Links [LM16b, LM17, FLMD19] is a language aimed at developing web applications. While based on a different foundations, it is related to SILL [TCP13, Gri16] in that both integrate traditional functional types with linear session types. As such, they can express many (nonlinear) programs that Rast cannot, but they support neither arithmetic refinements nor ergometric or temporal types.

Context-free session types [TV16, AMV20] generalize ordinary session types with sequential composition as well as permitting some polymorphism. The linear sublanguage of context-free session types can be modeled in Rast with nested polymorphism [DDMP21]. Several dependent extensions for session types in prior work including proof exchanges [TCP11], subtyping and constraint relations based on ATS [WX17], and equality based on $\beta\eta$-congruences [TY18]. However, none of them formally investigate type equality, nor provide a type checking algorithm realized in an implementation.

12. Conclusion

This paper describes the Rast programming language. In particular, we focused on the concrete syntax, type checking and subtyping, parametric polymorphism [DDMP21], the refinement layer [DP20b, DP20c], and its applicability to work [DHP18b] and span analysis [DHP18a]. The refinements rely on an arithmetic solver based on Cooper's algorithm [Coo72]. The interpreter uses the shared memory semantics introduced in recent work [PP20]. We concluded with several examples demonstrating the efficacy of the refined type system in expressing and verifying properties about data structure sizes and values. We also illustrated the work and span bounds for several examples, all of which have been verified with our system, and are available in an open-source repository [DDP19].

In the future, we plan to address some limitations of the Rast language. One goal of Rast was to explore the boundaries of purely linear programming with general recursion. Often, this imposes a certain programming discipline and can be inconvenient if we need to drop

or duplicate channels. Recent work on adjoint logic [PP19] uniformly integrates different logical layers into a unified language by assigning modes to communication. We plan to utilize this adjoint formulation to support shared [BP17] and unrestricted channels. Prior work on the SILL [Gri16] and Nomos [DBH$^+$21] have demonstrated such an integration is helpful in general-purpose programming.

In the direction of parametric polymorphism, we plan to develop the theory of subtyping. Our initial investigation suggests that subtyping is undecidable, and thus would like to explore the boundaries of our current subtyping algorithm. Relatedly, we also plan to explore if we can extend the ideas of reconstruction to explicit quantifiers.

With respect to refinements, we intend to pursue richer constraint domains such as non-linear arithmetic, particularly SMT. We would also like to support reconstruction for the temporal fragment of Rast. Unfortunately, the $\bigcirc$ operator affects all connected channels at once and its proof-theoretic properties are not as uniform as those of polymorphism, proof constraints, or ergometric types, posing a significant challenge. We also plan to explore dependent session type systems to express data-dependent distributed protocols.

References

[AMV20]   Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Deciding the bisimilarity of context-free session types. In A. Biere and D. Parker, editors, *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, pages 39–56, Dublin, Ireland, April 2020. Springer LNCS 12079.

[BCM$^+$15]   Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sabastian Podda, and Livio Pompianu. A contract-oriented middleware. In C. Braga and P. Ölveczky, editors, *Formal Aspects of Component Software (FACS 2015)*, pages 86–104. Springer LNCS 9539, 2015.

[BCPP15]   Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia Allessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In S. Graf and M. Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2015)*, pages 161–177, Grenoble, France, June 2015. Springer LNCS 9039.

[BMV12]   Pedro Baltazar, Dimitris Mostrous, and Vasco T. Vasconcelos. Linearly refined session types. In S. Alves and I. Mackie, editors, *International Workshop on Linearity (LINEARITY 2012)*, pages 38–49, Tallinn, Estonia, April 2012. EPTCS 101. `doi:10.4204/eptcs.101.4`.

[BP17]   Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. In *International Conference on Functional Programming (ICFP)*, pages 37:1–37:29. ACM, September 2017. Extended version available as Technical Report CMU-CS-17-106R, June 2017.

[CB20]   Ruofei Chen and Stephanie Balzer. Ferrite: A judgmental embedding of session types in Rust. *CoRR*, 2020. `arXiv:2009.13619`.

[Coo72]   David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7(91-99):300, 1972.

[CP10]   Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

[CPT16]   Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 726(3):367–423, 2016. `doi:10.1017/S0960129514000218`.

[CS09]   Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044 – 1077, 2009. Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006). `doi:10.1016/j.ic.2008.11.006`.

[DBH$^+$21]   Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-Aware Session Types for Digital Contracts. In *34th IEEE Computer Security Foundations Symposium, CSF*, 2021. To appear.

[DCPT12] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Annual Conference on Computer Science Logic (CSL 2012)*, pages 228–242, Fontainebleau, France, September 2012. LIPIcs 16.

[DDMP21] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Nested Session Types. In *30th European Symposium on Programming, ESOP*, 2021. To appear.

[DDP19] Ankush Das, Farzaneh Derakhshan, and Frank Pfenning. Rast Implementation. `https://bitbucket.org/fpfenning/rast/src/master/`, 2019. Accessed: 2019-11-11.

[DHP18a] Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. In M. Flatt, editor, *Proceedings of International Conference on Functional Programming (ICFP 2018)*, pages 91:1–91:30, St. Louis, Missouri, USA, September 2018. ACM.

[DHP18b] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In A. Dawar and E. Grädel, editors, *Proceedings of 33rd Symposium on Logic in Computer Science (LICS 2018)*, pages 305–314, Oxford, UK, July 2018.

[DP19] Farzaneh Derakhshan and Frank Pfenning. Circular proofs as session-typed processes: A local validity condition. *CoRR*, August 2019. `arXiv:abs/1908.01909`.

[DP20a] Ankush Das and Frank Pfenning. Rast: Resource-aware session types with arithmetic refinements (system description). In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 33:1–33:17. LIPIcs 167, 2020. `doi:10.4230/LIPIcs.FSCD.2020.33`.

[DP20b] Ankush Das and Frank Pfenning. Session types with arithmetic refinements. In I. Konnov and L. Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171, pages 13:1–13:18. LIPIcs 171, 2020. `doi:10.4230/LIPIcs.CONCUR.2020.13`.

[DP20c] Ankush Das and Frank Pfenning. Verified linear session-typed concurrent programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming (PPDP 2020)*, pages 7:1–7:15, Bologna, Italy, September 2020. ACM.

[Dyc82] Dyck. Gruppentheoretische Studien. (mit drei lithographirten Tafeln.). *Mathematische Annalen*, 20:1–4, 1882. URL: `http://eudml.org/doc/157013`.

[FLMD19] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types. In *Proceedings of the 46th Symposium on Programming Languages (POPL 2019)*, pages 28:1–28:29, Cascais, Portugal, January 2019. ACM Press.

[FV13] Juliana Franco and Vasco T. Vasconcelos. A concurrent programming language with refined session types. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods (SEFM 2013)*, pages 15–28, Madrid, Spain, September 2013. Springer LNCS 8368.

[GG13] Dennis Griffith and Elsa L. Gunter. LiquidPi: Inferrable dependent session types. In *Proceedings of the NASA Formal Methods Symposium*, pages 186–197. Springer LNCS 7871, 2013.

[GH05] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, Nov 2005. `doi:10.1007/s00236-005-0177-z`.

[GL87] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

[Gri16] Dennis Griffith. *Polarized Substructural Session Types*. PhD thesis, University of Illinois at Urbana-Champaign, April 2016.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

[Hon93] Kohei Honda. Types for dyadic interaction. In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR 1993)*, pages 509–523. Springer LNCS 715, 1993.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, pages 122–138. Springer LNCS 1381, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In G. Necula and P. Wadler, editors, *Proceedings of the 35th Symposium on Principles of*

Programming Language (POPL 2008), pages 273–284, San Francisco, California, USA, January 2008. ACM.

[JML15] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In P. Bahr and S. Erdweg, editors, *Proceedings of the 11th Workshop on Generic Programming (WGP 2015)*, pages 13–22, Vancouver, Canada, August 2015. ACM.

[KH66] Allen J Korenjak and John E Hopcroft. Simple deterministic languages. In *7th Annual Symposium on Switching and Automata Theory (SWAT 1966)*, pages 36–46. IEEE, 1966.

[LM16a] Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In G. Mainland, editor, *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*, pages 133–145, Nara, Japan, September 2016. ACM.

[LM16b] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 434–447. ACM, 2016. `doi:10.1145/2951913.2951921`.

[LM17] Sam Lindley and J. Garrett Morris. Lightweight functional session types. In S. Gay and A. Ravara, editors, *Behavioural Types: from Theory to Tools*, chapter 12, pages 265–286. River Publishers, June 2017. `doi:https://doi.org/10.13052/rp-9788793519817`.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[OY16] Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In R. Bodik and R. Majumdar, editors, *Proceedings of the 43rd Annual Symposium on Principles of Programming Languages (POPL 2016)*, pages 568–581, St. Petersburg, Florida, January 2016. ACM.

[Pad17] Luca Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27(e4), 2017.

[PP19] Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. In F. Martins and D. Orchard, editors, *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*, pages 60–79, Prague, Czech Republic, April 2019. EPTCS 291.

[PP20] Klaas Pruiksma and Frank Pfenning. Back to futures, February 2020. `arXiv:2002.04607`.

[SY16] Alceste Scalas and Nobuko Yoshida. Lightweight session programming in Scala. In *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016)*, pages 21:1–21:28, Rome, Italy, July 2016. LICIcs 56.

[TCP11] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, page 161–172, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/2003476.2003499`.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M. Felleisen and P. Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 350–369, Rome, Italy, March 2013. Springer LNCS 7792.

[TV16] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *Proceedings of the 21st International Conference on Functional Programming (ICFP 2016)*, pages 462–475, Nara, Japan, September 2016. ACM.

[TY18] Bernardo Toninho and Nobuko Yoshida. Depending on session-typed processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 128–145, Cham, 2018. Springer International Publishing.

[Vas11] Vasco T. Vasconcelos. Session, from types to programming languages. *Bulletin of the EATCS*, 103:53–73, 2011.

[Vas12] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012. `doi:10.1016/j.ic.2012.05.002`.

[Wad12] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012)*, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.

[WPP16] Max Willsey, Rokhini Prabhu, and Frank Pfenning. Design and implementation of Concurrent C0. In *Fourth International Workshop on Linearity*, pages 73–82. EPTCS 238, June 2016.

[WX17]   Hanwen Wu and Hongwei Xi. Dependent session types. *CoRR*, abs/1704.07004, 2017. `arXiv:`
            `1704.07004`.
[XRWB16] Hongwei Xi, Zhiqiang Ren, Hanwen Wu, and William Blair. Session types in a linearly typed
            multi-threaded lambda-calculus. *CoRR*, 2016. `arXiv:1603.03727`.