Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description)

Ankush Das
Carnegie Mellon University, Pittsburgh, PA, USA

Frank Pfenning
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Traditional session types prescribe bidirectional communication protocols for concurrent computations, where well-typed programs are guaranteed to adhere to the protocols. Recent work has extended session types with refinements from linear arithmetic, capturing intrinsic properties 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 and system provide 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. 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.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Type theory

Keywords and phrases Session Types, Resource Analysis, Refinement Types

1 Introduction

Session types [13, 14, 17] provide a structured way of statically prescribing communication protocols in message-passing programs. In this system description we introduce the Rast programming language and implementation[1] which is based on *binary session types* governing the interaction of two processes along a single channel, rather than *multi-party session types* [15] 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 [1, 18, 2]. Moreover, 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. It furthermore supports arithmetic type refinements as well as ergometric and temporal types to measure the total work and span of Rast programs. The repository also contains a number of illustrative examples that highlight various language features, some of which we briefly sketch in this system description. The theory underlying Rast has been developed in several papers, starting with the Curry-Howard interpretation of linear logic as session-typed processes [1, 2], the treatment of general equirecursive types and type equality [10],

[1] available at `https://bitbucket.org/fpfenning/rast/src/master/rast/`

asynchronous communication [11, 9], ergometric types [6], temporal types [5], indexed types and indexed type equality [12, 7].

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 queue server in the system of basic recursive session types supporting the operations of insert (enqueue) and delete (dequeue).

$$\begin{aligned}\mathsf{queue}_A = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}_A,\\ &\mathbf{del} : \oplus\{\mathbf{none} : \mathbf{1},\\ &\qquad \mathbf{some} : A \otimes \mathsf{queue}_A\}\}\end{aligned}$$

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 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. 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 [7] to express the length of a queue. The more precise type

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

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 quantifiers $\exists n.\,A$ 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 [6, 4] and temporal session types [5]. 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 indices. Rast largely follows and extends [7].

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: we respond with **none** and close the channel, or **some** followed by the element. This gives us the following type

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

which expresses that the client has to send $2n$ ergs to insert an element ($\triangleleft^{2n}$), and 2 ergs to delete an element ($\triangleleft^{2}$). The ergometric type system (described in Section 4) verifies this work bound using the potential operators as described in the type.

Temporal session types [5] capture the time complexity of session-typed 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 $\Box$. Secondly, the queue does not interact at all with the elements it contains, so they have to be of type $\Box A$ 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 $\bigcirc$. In one spot, 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:

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

We see that even though the bucket brigade requires much work for every insertion (linear in the length of the queue) it is 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 for *reconstruction* which is sound and complete on valid implicit programs.

Rast is implemented in SML, and 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, type checker, reconstruction engines, and an interpreter, with particular attention to providing precise error messages.

To summarize, our implementation makes the following contributions, most of the theory of which can be found in the aforementioned papers [1, 6, 5, 7].

- **(i)** A session-typed programming language with arithmetic refinements applied to ergometric and temporal types for parallel complexity analysis.
- **(ii)** A type equality algorithm that works well in practice despite its theoretical undecidability [7] and uses Cooper's algorithm [3] with some small improvements to decide constraints in Presburger arithmetic (and heuristics for nonlinear constraints).
- **(iii)** A type checking algorithm that is sound and complete relative to type equality.
- **(iv)** A sound and complete reconstruction algorithm for a process language where most index and ergometric constructs remain implicit.
- **(v)** An interpreter for executing session-typed programs using the recently proposed shared memory semantics [16].

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

```
type queue{n} = &{ins : A -o queue{n+1},
                  del : +{none : ?{n = 0}. 1,
                          some : ?{n > 0}. A * queue{n-1}}}
decl empty : . |- (q : queue{0})
decl elem{n} : (x : A) (t : queue{n}) |- (q : queue{n+1})

proc q <- empty =
  case q (                    % receive a label along q
    ins => x <- recv q ;      % if 'ins' receive a channel x along q
           e <- empty ;       % spawn a new empty process
           q <- elem{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{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{n+1} x t % recurse
  | 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`

rendering of their abstract syntax, except that all arithmetic expressions are enclosed in braces to make them visually easily discernible.

```
type queue{n} = &{ins : A -o queue{n+1},
                  del : +{none : ?{n = 0}. 1,
                          some : ?{n > 0}. A * queue{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{0}` along a channel named `q`. It does not *use* any other services (indicated by '`.`'), so its type is declared with

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

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

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

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

Listing 2 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 and contains two instances of `assert` to match the constraints `?{n = 0}` and `?{n > 0}` in the two possible responses to a delete request. These two lines would be omitted in implicit form 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.

3 Basic and Refined Session Types

We present the basic system of session types and its arithmetic refinement, postponing ergometric and temporal types to Section 4.

$$
\begin{array}{lrcl}
\text{Types} & A & ::= & \oplus\{\ell : A\}_{\ell\in L} \mid \&\{\ell : A\}_{\ell\in L} \mid A \otimes A \mid A \multimap A \mid \mathbf{1} \mid V\overline{[e]} \\
& & \mid & ?\{\phi\}.\, A \mid !\{\phi\}.\, A \mid \exists n.\, A \mid \forall n.\, A \\
\text{Arith. Exps.} & e & ::= & i \mid e + e \mid e - e \mid e \times e \mid v \\
\text{Arith. Props.} & \phi & ::= & e < e \mid e \leq e \mid e = e \mid e \geq e \mid e > e \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \supset \phi
\end{array}
$$

Here, i stands for a natural number, n for an arithmetic variable, L for a finite set of labels, V for a defined type, and $\overline{[e]}$ for sequence of arithmetic expressions. Arithmetic propositions could contain quantifiers, but at present the implementation only supports them at the level of types. Arithmetic expressions may be nonlinear, although a definitive outcome of type-checking is only guaranteed if they are linear and therefore lie within Presburger arithmetic.

Our implementation does not support type polymorphism but it is convenient in some of the examples. We therefore allow definitions such as $\mathsf{queue}_A[n] = \ldots$ and interpret them as a family of definitions, one for each possible type A.

We review a few basic session type operators before introducing the quantified type constructors. Table 1 overviews the session types with their continuations, their associated process terms and operational description.

The complete typing judgment for process expressions has the form of a sequent

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

where $\mathcal{V}$ are index 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. The *potential* q is explained in Section 4. We propose and maintain that the x_i and x are all distinct, and that all free index variables in $\mathcal{C}$, Δ, P, and A are contained among $\mathcal{V}$. Finally, Σ is a fixed signature containing type and process definitions (explained in Section 3.1). Because it is fixed, we elide it from the presentation of the rules. In addition we write $\mathcal{V}\ ;\ \mathcal{C} \vDash \phi$ for semantic entailment (ϕ is true assuming $\mathcal{C}$) in the constraint domain where $\mathcal{V}$ contains all arithmetic variables in $\mathcal{C}$ and ϕ.

3.1 Basic Session Types

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

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

Table 1 Basic session types with operational description

Channel Passing. The *tensor* operator $A \otimes B$ prescribes that the provider of $x : A \otimes B$ sends a channel 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 using the term $y \leftarrow \mathsf{recv}\ x\ ;\ Q$, binding it to variable y and continuing to execute Q. The dual operator $A \multimap B$ allows the provider to receive a channel of type A and continue to provide type B. Finally, the type $\mathbf{1}$ indicates *termination*, operationally denoting that the provider sends a *close* message and terminates the communication.

A 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. Operationally, we refer to it as *forwarding*.

Process Definitions. Typed process definitions have the form $\Delta \vdash^{q} f\overline{[n]} = P :: (x : A)$ where f is the name of the process and P its definition. In addition, $\overline{n}$ is a sequence of arithmetic variables that Δ, q, P, and A can refer to. Note that in the implementation a typed definition is split up into a declaration and a simple definition

```
decl f{n1}...{nk} : (x1 : A1) ... (xm : Am) |- (x : A)
proc x <- f{n1}...{nk} x1 ... xm = P
```

All types and processes may be mutually recursive.

A new instance of a defined process f can be spawned with the expression $x \leftarrow f\overline{[e]}\ \ \overline{y}\ ;\ Q$ where $\overline{y}$ is a sequence of channels matching the antecedents Δ and $\overline{[e]}$ is a sequence of arithmetic expression matching the variables $\overline{[n]}$. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q. The declaration of f is looked up in the signature Σ, and $\overline{e}$ is substituted for $\overline{n}$ and $\overline{y}$ for Δ. Sometimes a process invocation is a tail call, written without a continuation as $x \leftarrow f\overline{[e]}\ \overline{y}$.

Type Definitions. We allow (possibly mutually recursive) type definitions $V\overline{[n]} = A$, or, in concrete syntax

```
type v{n1}...{nk} = A
```

in the signature Σ. Here, $\overline{[n]}$ again denotes a sequence of arithmetic variables. We also require A to be *contractive* [10] meaning A should not itself be a type name. Our type definitions are *equirecursive* so we can silently replace type names $V\overline{[e]}$ indexed with arithmetic refinements by $A[\overline{e}/\overline{n}]$ during type checking.

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

Table 2 Refined session types with operational description

All types in a signature must be *valid* which requires that all free arithmetic variables of $\mathcal{C}$ and A are contained in $\mathcal{V}$, and that for each arithmetic expression e in A we can prove $\mathcal{V}' \ ;\ \mathcal{C}' \vdash e : \mathsf{nat}$ for the constraints $\mathcal{C}'$ known at the occurrence of e (implying $e \geq 0$).

3.2 The Refinement Layer

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

Quantification. The provider of $(c : \exists n.\, A)$ should send a witness e along channel c and then continue as $A[e/n]$. 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 $\mathcal{V} \ ;\ \mathcal{C} \vdash e : \mathsf{nat}$.

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

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

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.

Constraints. Refined session types also allow constraints over index variables. 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. Statically, it is the provider's responsibility to ensure that ϕ holds, while the client is permitted to assume that ϕ is true.

Thus, the typing rules for this new type constructor are

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

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

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 remaining issue is how to type-check a branch that is impossible due to unsatisfiable constraints. A special impossibility construct is used to handle this situation (dead branches).

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

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. In practice, we almost never write this construct since reconstruction will fill in missing branches, whose impossibility is then verified by the type checker.

Example: Binary Numbers. As a second 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 labels **b1**, **b0**, **b1**, **b1**, **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. Because (linear) arithmetic contains no division operator, we express the type $\mathsf{bin}[n]$ 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.

4 Ergometric and Temporal Session Types

An important application of refinement types is complexity analysis. Prior work on resource-aware session types [6, 5, 4] crucially rely on arithmetic refinements to express work and time bounds. In this section, we review these type systems. 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.

4.1 Ergometric Types

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

Two dual type constructors $\triangleright^r A$ and $\triangleleft^r A$ are used to exchange potential. The provider of $x : \triangleright^r A$ must *pay* r units of potential along x using process term $(\mathsf{pay}\, x\, \{r\}\ ;\ P)$, and continue to provide A by executing P. These r units are deducted from the potential stored inside the sender. Dually, the client must receive the r units of potential using the term $(\mathsf{get}\, x\, \{r\}\ ;\ Q)$ 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 $\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash^q_\Sigma P :: (x : A)$. We allow potential q to refer to index variables in $\mathcal{V}$ to capture variable potential. The typing rules for $\triangleright^r A$ are

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

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

In both cases, we check that the exchanged potential in the expression and type matches ($r_1 = r_2$), and while paying, we ensure that the sender has sufficient potential to pay. The dual type $\triangleleft^r A$ enables the provider to receive potential that is sent by its client. Since the sent or received potential must match the one prescribed by the type, our reconstruction algorithm can insert the pay and get actions in a sound and complete way (get as soon as possible and pay as late as possible).

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 `linlam-reds.rast` that counts the number of reductions necessary for the evaluation of an expression in the linear λ-calculus.

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

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

Example: Queue Revisited. 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{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                               some : ?{n > 0}. A * queue{n-1}}}

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

Interestingly, the exact code of Listing 2 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, greatly aided by the linearity of the language. 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{n} = &{enq : <{6}| nat -o queue{n+1},
                  deq : <{4}| +{none : ?{n = 0}. 1,
                               some : ?{n > 0}. nat * queue{n-1}}}
```

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

4.2 Temporal 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 r units on the left of the context using $[A]^t_L$, and on the right by $[A]^t_R$. Briefly, $[\bigcirc^t A]^{-t}_L = [\bigcirc^t A]^{-t}_R = A$.

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

Always A**.** A process providing $x : \Box A$ 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 $x : \Box A$ 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 $\Delta\ \mathsf{delayed}^{\Box}$ which requires each antecedent to have the form $y_i : \bigcirc^{n_i} \Box B_i$.

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

Rast also has its dual modality $\Diamond A$, which communicates at some indeterminate future time. This is used when the time (span) of a computation is unpredictable or not expressible within the constraints of the language

Since all temporal operators ultimately model time, they interact with each other and the temporal displacement operator used in $\bigcirc LR$ rule becomes more complicated. 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.

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

Example: Queue Revisited. 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{n} = [] {enq : () A -o ()()()queue{n+1},
                    deq : ()+{none: () ?{n = 0}. 1,
                              some: () ?{n > 0}. A * ()queue{n-1}}}
decl empty : . |- (q : ()()queue{0})
decl elem{n} : (x : A) (r : ()()queue{n}) |- (q : queue{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.

Abstract Types	Concrete Types	Abstract Syntax	Concrete Syntax
$\oplus\{l : A, \ldots\}$	`+{l : A, ...}`	$x.k$	`x.k`
$\&\{l : A, \ldots\}$	`&{l : A, ...}`	case x $(\ell \Rightarrow P)_{\ell \in L}$	`case x (l => P | ...)`
$A \otimes B$	`A * B`	send x w	`send x w`
$A \multimap B$	`A -o B`	$y \leftarrow$ recv x	`y <- recv x`
$\mathbf{1}$	`1`	close x	`close x`
		wait x	`wait x`
$\exists n.\, A$	`?n. A`	send x $\{e\}$	`send x {e}`
$\forall n.\, A$	`!n. A`	$\{n\} \leftarrow$ recv x	`{n} <- recv x`
$?\{n = 0\}.\, A$	`?{n = 0}. A`	assert x $\{n = 0\}$	`assert x {n = 0}`
$!\{n = 0\}.\, A$	`!{n = 0}. A`	assume x $\{n = 0\}$	`assume x {n = 0}`
$\triangleright^r A$	`|{r}> A`	pay x $\{r\}$	`pay x {r}`
$\triangleleft^r A$	`<{r}| A`	get x $\{r\}$	`get x {r}`
$\bigcirc^t A$	`({t}) A`	delay t	`delay {t}`
$\Box A$	`[] A`	when x	`when x`
$\Diamond A$	`<> A`	now x	`now x`
$V\overline{[e]}$	`V{e1}...{ek}`		

Table 3 Abstract and Corresponding Concrete Syntax for Types and Expressions

5 Implementation

We have implemented a prototype for resource-aware session types in Standard ML (6700 lines of code). This implementation contains a lexer and parser (1355 lines), an arithmetic solver (1083 lines), a type checker (2852 lines), pretty printer (375 lines), reconstruction engine (880 lines), and interpreter (155 lines). The source code is well-documented and available open-source.

Syntax. Table 3 describes the abstract and concrete syntax for types and their corresponding process expressions. 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{n} = A
decl f : (x1 : A1) ... (xn : An) |- (x : A)
proc x <- f x1 ... xn = P
```

Listing 2 Top-Level Declarations

The first line is a *type definition*, where v is the name with index variable 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 offered channel is x of type A. Finally, the last line is a *process definition* for the same process f defined using the process expression P. We use a hand-written lexer and shift-reduce parser to read an input file and generate the corresponding abstract syntax tree of the program. 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. To simplify and improve the efficiency of the type equality algorithm, we also assign internal names to type subexpressions parameterized over their free 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 [7] 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, leaving reconstruction of quantifiers and temporal constructs to future work.

The implementation takes some care to provide good 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 $V\overline{[e]}$ with $V\overline{[n]} = A$ to $A[\overline{e}/\overline{n}]$ we record a reverse mapping from $A[\overline{e}/\overline{n}]$ to $V\overline{[e]}$. 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.

Type Equality. At the core of type checking lies type equality, defined coinductively [10]. With arithmetic refinements this equality is undecidable, but have found what seems to be a practical approximation [7], incrementally constructing a bisimulation closed under reflexivity. This algorithm always terminated, but may fail to establish an equality if the coinductive invariant is not general enough. Rast therefore allows the programmer to assert an arbitrary number of additional types equalities with the construct

```
eqtype V{e1}...{en} = V'{e1'}...{ek'}
```

These are then checked one by one, assuming all other asserted equalities. The default construction of the bisimulation 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 [3] for Presburger arithmetic. We found a small number of optimizations were necessary, but the resulting algorithm has been quite efficient and robust.

- **(i)** 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.
- **(ii)** We exploit that we are working over natural numbers so all solutions have a natural lower bound.

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.

(i) To check $e_1 = e_2$, we normalize $e_1 - e_2$ and check that each coefficient of the normal form is 0.

(ii) To check $e_1 \geq e_2$, we normalize $e_1 - e_2$ and check that each coefficient is non-negative.

(iii) 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 $y \geq 0$.

(iv) 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 current set of heuristics beyond Presburger arithmetic.

Interpreter. The current version of the interpreter pursues a sequential schedule following a prior proposal [16]. We only execute programs that have no free 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) or all number in the range $[2, 257]$ takes 27.172 milliseconds using MLton during our experiments (see machine specifications below).

6 Examples

We present several different kinds of example from varying domains illustrating different features of the type system and algorithms. Table 4 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.

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

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

(iii) **linlam**: expressions in the linear λ-calculus indexed by their size.

(iv) **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.

(v) **primes**: the sieve of Eratosthenes to classify numbers as prime or composite.

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

(vii) **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.

(viii) **theorems**: processes representing valid circular [8] proofs of simple theorems such as $n(k+1) = nk + n, n + 0 = n, n * 0 = 0$, etc.

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

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 4 Case Studies

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

Lists with Potential. The type $\mathsf{list}[n, p]$ is the type of lists of length n where each element carries potential p. This potential is transferred to the client of the list before each element. We use elements of type nat to stand in for an arbitrary type.

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

Standard processes on lists such as append or reverse and their work analysis can be found in the repository (file `examples/list-work.rast`).

Map: In a linear setting, a *mapper* from type A to type B is a process that provides the choice between two labels, **next** and **done**. When receiving **next** it then receives an element of type A, responds with an element of type B and recurses to wait for the next label. If it receives **done**, it terminates. Below is one possible type that illustrates nonlinear constraints. We could also distribute the cost of the mapper $q + 1$ to the individual list elements

```
type mapper{q} = &{ next : <{q+1}| A -o B * mapper{q},
                    done : <{1}| 1 }
decl map{n}{p}{q} : (k : list{n}{p+4}) (m : mapper{q})
                        |{3+(n*(q+1)+1)}- (l : list{n}{p})
```

The temporal type for the same mapper is given below where `‘A` (pronounced “tick A”) stands for `()A` conveying that it represents the unavoidable minimal delay due to a send or receive.

```
type listA{n}{r} = +{cons: ‘?{n > 0}. ([]A * ‘({r+5}) listA{n-1}{r}),
                     nil: ‘?{n = 0}. 1}
type mapper{r} = &{next : ‘A -o ‘({r}) ([] B * ‘({4}) mapper{r}),
                   done : ‘1}
decl map{n}{r} : (l : listA{n}{r}) (m : ({2}) mapper{r})
                      |- (k : ({r+5}) listB{n}{r})
```

Folding a list can be done in a similar fashion.

Linear λ-Calculus. We demonstrate an implementation of the (untyped) linear λ-calculus, including evaluation, in which the index objects track the size of the expression. Type-checking verifies that that the result of evaluating a linear λ-term is no larger than

the original term. Our representation uses linear higher-order abstract syntax (see file `examples/linlam-size.rast`).

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

An expression of size n is either a λ (the label **lam**) or an application (label **app**). In case of **lam**, after proving $n > 0$, it expects an expression of size n_1 as an argument and then behaves like the body of the λ-abstraction of size $n_1 + n - 1$. In case of **app**, it sends two expressions of size n_1 and n_2 such that $n = n_1 + n_2 + 1$.

Interestingly, the result type of evaluation contains an existential quantifier since we do not know the precise size of the value—we just know it is bounded by n. Also, as exemplified in the type of `val{n}`, a value can only be a λ-expression (label **app** missing).

Trie. We illustrate the data structure of a trie to maintain multisets of binary numbers. There is a fair amount of parallelism since consecutive requests to insert numbers into the trie can be carried out concurrently. We start with binary numbers where each bit carries potential p.

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

Trie Interface. A trie is represented by the type $\mathsf{trie}[n]$ where n is the number of elements in the current multiset. When inserting a number it updates to $\mathsf{trie}[n+1]$. When we delete a number x from the trie we delete all copies of x and return its multiplicity. If m is the multiplicity of the number, then after deletion the trie will have $\mathsf{trie}[n-m]$ elements. This requires the constraint that $m \leq n$: the multiplicity of an element cannot be greater than the total number of elements in the multiset.

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

The way we insert the binary number is starting at the root with the least significant bit and recursively insert the number into the left or right subtrie, depending on whether the bit is **b0** or **b1**. When we reach the end of the sequence of bits (**e**) we increase the multiplicity at the leaf we have reached.

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

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

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

References

1 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory*, CONCUR'10, pages 222–236, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1887654.1887670.

2 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 760, 11 2014. doi:10.1017/S0960129514000218.

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

4 Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts, 2019. arXiv:1902.06056.

5 Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.*, 2(ICFP):91:1–91:30, July 2018. URL: http://doi.acm.org/10.1145/3236786, doi:10.1145/3236786.

6 Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 305–314, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3209108.3209146, doi:10.1145/3209108.3209146.

7 Ankush Das and Frank Pfenning. Session types with arithmetic refinements and their application to work analysis, 2020. arXiv:2001.04439.

8 Farzaneh Derakhshan and Frank Pfenning. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *arXiv e-prints*, page arXiv:1908.01909, Aug 2019. arXiv:1908.01909.

9 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.

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

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

12 Dennis Griffith and Elsa L. Gunter. Liquidpi: Inferrable dependent session types. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, pages 185–197, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

13 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

14 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

15 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1328438.1328472, doi:10.1145/1328438.1328472.

16 Klaas Pruiksma and Frank Pfenning. A shared-memory semantics for mixed linear and non-linear session types. 2018.

17 Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52 – 70, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0890540112001022, doi:https://doi.org/10.1016/j.ic.2012.05.002.

18 Philip Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, September 2012. URL: https://doi.org/10.1145/2398856.2364568, doi:10.1145/2398856.2364568.