Rast: Resource-Aware Session Types with Arithmetic Refinements

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.

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 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], 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).

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

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 \( \rightarrow \) operator, and then the type recurses back to \( \text{queue}_A \). 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 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] which can be used to express the length of a queue. The more precise type

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

uses the index refinement \( n \) to indicate the number of elements in the queue. In addition, the type constraint \(?\{\phi\}.A\) read as “there exists a proof of \( \phi \)” is analogous to the assertion of \( \phi \) 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 \( \phi \), analogous to the assumption of \( \phi \)). 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 prior work on session types with arithmetic refinements [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 \textit{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 \texttt{ins} and then the new element. On the other hand, responding to the \texttt{del} request requires only 2 ergs: we respond with \texttt{none} and close the channel, or \texttt{some} followed by the element. This gives us the following type

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

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

Temporal session types \cite{10.5555/1062112.1062128} capture the time complexity of session-typed programs assuming maximal parallelism on unboundedly many processors, often called the \textit{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 $\odot$. 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:

\[
\text{queue}_A[n] = \Box \& \{ \text{ins} : \odot(\Box A \rightarrow \odot^{3}\text{queue}_A[n+1]), \\
\text{del} : \odot \oplus \{ \text{none} : \odot ?\{n = 0\}.1, \\
\text{some} : \odot ?\{n > 0\}.\Box A \otimes \odot \text{queue}_A[n-1] \}
\]

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 \textit{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 \textit{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 \textit{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.

(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 \cite{10.5555/1857596.1857624} and uses Cooper’s algorithm \cite{10.5555/935220.935222} with some small improvements to decide constraints in Presburger arithmetic (and heuristics for nonlinear constraints).
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 (
    ins => x <- recv q ;
    e <- empty ;
    q <- elem{0} x e
    | del => q. none
    assert q {0=0} ;
    close q )

proc q <- elem{n} x t =
case q (
    ins => y <- recv q ;
    t.ins ;
    send t y ;
    q <- elem{n+1} x t
    | del => q.some
    assert q {n+1>0} ;
    send q x ;
    q <-> t )

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

(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
rendering of their abstract syntax (Table 3), except that all arithmetic expressions are
enclosed in braces to make them visually easily discernible.

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
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`.

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 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 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.
in Section 3.1). Because it is fixed, we elide it from the presentation of the rules. In addition, we write $V : C \models \phi$ for semantic entailment ($\phi$ is true assuming $C$) in the constraint domain where $V$ contains all arithmetic variables in $C$ and $\phi$.

### 3.1 Basic Session Types

#### External and Internal Choice.

The external choice type constructor $\& \{ \ell : A \}_L$ is an $n$-ary labeled generalization of the additive conjunction $A \& B$. Operationally, it requires the provider of $x : \& \{ \ell : A \}_L$ 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 $\text{case } x (\ell \Rightarrow P)_L$. 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 $\oplus \{ \ell : A \}_L$ is the dual of external choice requiring the provider to send one of the labels $k \in L$ that the client must branch on.

#### Channel Passing.

The tensor operator $A \otimes B$ prescribes that the provider of $x : A \otimes B$ sends a channel $w$ of type $A$ and continues to provide type $B$. The corresponding process term is $\text{send } x w ; P$ where $P$ is the continuation. Correspondingly, its client must receive a channel using the term $y \leftarrow \text{recv } x ; Q$, binding it to variable $y$ and continuing to execute $Q$. The dual operator $A \rightarrow B$ allows the provider to receive a channel of type $A$ and continue to provide type $B$. Finally, the type $\mathbf{1}$ 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 and Type Definitions.

Process definitions (possibly mutually recursive) have the form $\Delta \vdash f[n] = P :: (x : A)$ where $f$ is the name of the process and $P$ its definition. In addition, $\pi$ is a sequence of
Table 2

<table>
<thead>
<tr>
<th>Type</th>
<th>Cont.</th>
<th>Process Term</th>
<th>Cont.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c : \exists n . A )</td>
<td>( c : A[i/n] )</td>
<td>send ( c { e } ; P )</td>
<td>( P )</td>
<td>provider sends the value ( i ) of ( e ) along ( c )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { n } \leftarrow \text{recv} c ; Q )</td>
<td>( Q[i/n] )</td>
<td>client receives number ( i ) along ( c )</td>
</tr>
<tr>
<td>( c : \forall n . A )</td>
<td>( c : A[i/n] )</td>
<td>( { n } \leftarrow \text{recv} c ; P )</td>
<td>( P[i/n] )</td>
<td>provider receives number ( i ) along ( c )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send ( c { e } ; Q )</td>
<td>( Q )</td>
<td>client sends value ( i ) of ( e ) along ( c )</td>
</tr>
<tr>
<td>( c : ?{ \phi } . A )</td>
<td>( c : A )</td>
<td>assert ( c { \phi } ; P )</td>
<td>( P )</td>
<td>provider asserts ( \phi ) on channel ( c )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assume ( c { \phi } ; Q )</td>
<td>( Q )</td>
<td>client assumes ( \phi ) on ( c )</td>
</tr>
<tr>
<td>( c : !{ \phi } . A )</td>
<td>( c : A )</td>
<td>assume ( c { \phi } ; P )</td>
<td>( P )</td>
<td>provider assumes ( \phi ) on channel ( c )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assert ( c { \phi } ; Q )</td>
<td>( Q )</td>
<td>client asserts ( \phi ) on ( c )</td>
</tr>
</tbody>
</table>

Refined session types with operational description

arithmetic variables that \( \Delta, q, P \), and \( A \) can refer to. Note that in the implementation a typed definition is split up into a declaration and a simple definition

\[
\text{decl } f{n1} \ldots {nk} : (x1 : A1) \ldots (xm : Am) \vdash (x : A) \\
\text{proc } x \leftarrow f{n1} \ldots {nk} \ x1 \ldots xm = P
\]

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

We allow (possibly mutually recursive) type definitions \( V[e] = A \), or, in concrete syntax

\[
\text{type } v{n1} \ldots {nk} = A
\]

in the signature \( \Sigma \). Here, \([n]\) 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 \( V[e] \) indexed with arithmetic refinements by \( A[\overline{e/\overline{n}}] \) during type checking.

All types in a signature must be valid which requires that all free arithmetic variables of \( C \) and \( A \) are contained in \( \mathcal{V} \), and that for each arithmetic expression \( e \) in \( A \) we can prove \( \mathcal{V} \vdash e : \text{nat} \) for the constraints \( C' \) 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 \( \mathcal{V} \).

\[
\frac{\mathcal{V} ; C \vdash e : \text{nat} \quad \mathcal{V} ; C ; \Delta \vdash P :: (x : A[e/n])}{\mathcal{V} ; C ; \Delta \vdash \text{send} \ x \{ e \} ; P :: (x : \exists n . A)} \quad \exists R
\]
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 $\phi$ along $c$ and the client should receive such a proof. Statically, it is the provider’s responsibility to ensure that $\phi$ holds, while the client is permitted to assume that $\phi$ is true. The dual operator $!\{\phi\}. A$ reverses the role of provider and client. The provider of $c : !\{\phi\}. A$ may assume the truth of $\phi$, while the client must verify it. The typing rules for the $?$ type constructor are

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

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).

$$
\frac{\forall \Delta. (x : A) \vdash^? Q :: (z : C)}{\forall \Delta, (x : ?\{\phi\}. A) \vdash^? \text{assert} \ x \ \{\phi\} ; \ P :: (x : ?\{\phi\}. A)} \quad ?R
$$

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

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 $b_1, b_0, b_1, b_1, e$, close sent or received on a given channel with the least significant bit first. Here $e$ represents 0 (the empty sequence of bits), while $b_0$ and $b_1$ represent bits 0 and 1, respectively. Because (linear) arithmetic contains no division operator, we express the type $\text{bin}\{n\}$ of binary numbers with value $n$ using existential quantification, with the concrete syntax $?k. A$ for $\exists k. A$.

```rasta
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 $b_0$ ensures the representation is unique and there are no leading zeros; the same constraint for $b_1$ 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 \((\text{pay} \ x \ r \ \{r\}; \ P)\), 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 \((\text{get} \ x \ r \ \{r\}; \ Q)\) 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 \( V ; C ; \Delta \Gamma^\Sigma P :: (x : A) \). We allow potential \( q \) to refer to index variables in \( V \) to capture variable potential. The typing rules for \( \triangleright^r A \) are

\[
\frac{V ; C \models q \geq r_1 = r_2}{V ; C ; \Delta \Gamma^\Sigma \text{pay} \ x \ r_1 \ ; \ P :: (x : \triangleright^r A)} \quad \triangleright^R
\]

\[
\frac{V ; C \models r_1 = r_2}{V ; C ; \Delta \Gamma^\Sigma \text{get} \ x \ r_1 \ ; \ Q :: (z : C)} \quad \triangleright^L
\]

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 \( \text{work} \ r \ ; \ P \) 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 \( \text{work} \ r \). An example of this is given in the file \textit{linlam-reds.rast} that counts the number of reductions necessary for the evaluation of an expression in the linear \( \lambda \)-calculus.

\[
\frac{V ; C \models q \geq r}{V ; C ; \Delta \Gamma^\Sigma \text{work} \ r \ ; \ P :: (x : A)} \quad \text{work}
\]
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`.

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

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

```rast
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 (◊A)`, `always (□A)`, and `eventually (♦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 ◊ operator from every channel it refers to. We denote consuming `r` units on the left of the context using `[A]_L^r$, and on the right by `[A]_R^r$. Briefly, `[◊A]_L^{-t} = [◊A]_R^{-t} = A$.

\[
\frac{\forall \; \mathcal{C} \models t \geq 0 \quad \mathcal{V} : \mathcal{C} ; [\Delta]_L^{-t} \models^\mathcal{V} Q \quad (x : [A]_R^{-t})}{\mathcal{V} : \mathcal{C} ; \Delta \models^\mathcal{V} delay(t) ; P :: (x : A) \quad ◊ LR}
\]

**Always A.**

A process providing `x : □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 \( \Delta \text{ delayed} \Box \) which requires each antecedent to have the form \( y_i : O^{n_i} \Box B_i \).

\[
\Delta \text{ delayed} \Box \Delta \vdash P :: (x : A) \quad \Delta, x : A \vdash Q :: (z : C) \quad \Delta, x : \Box A \vdash (\text{when} \? (x) ; P) :: (x : \Box A) \quad \Box R
\]

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 (more details in prior work [5]).

**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 \( \Box \) represents \( \Box \). We also show the types of the empty and elem processes (see file examples/time.rast).

```plaintext
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.

## 5 Implementation

We have implemented a prototype for Rast 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 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.

```plaintext
| 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
Abstract Types | Concrete Types | Abstract Syntax | Concrete Syntax
---|---|---|---
⊕{l : A, ...} | +{l : A, ...} | x.k | x.k
∧{l : A, ...} | &{l : A, ...} | case x (ℓ ⇒ P)∈L | case x (l ⇒ P | ...)
A ⊗ B | A * B | send x w | send x w
A →o B | A →o B | y ← recv x | y ← recv x
1 | 1 | close x | close x
wait x | | wait x | |
∃n. A | ?n. A | send x {e} | send x {e}
∀n. A | !n. A | {n} ← 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}
▷A | | pay x {r} | pay x {r}
◁A | < {r} | get x {r} | get x {r}
◇A | (t) | delay t | delay {t}
□A | [] A | when x | when x
◊A | <> A | now x | now x
V[e] | V{e1}...{ek} | x ← y | x <-> y
x ← f x₁...xₙ | x ← f x₁...xₙ
Table 3 Abstract and Corresponding Concrete Syntax for Types and Expressions

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 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 $V[e]$ with $V[n] = A$ to $A[\pi/\pi]$ we record a reverse mapping from $A[\pi/\pi]$ to $V[e]$. 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 terminates, 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 type equalities with the construct

\[ \text{eqtype } V\{e1\}...\{en\} = V'\{e1'\}...\{ek'\} \]

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, 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.
(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 (which can be improved in the future).

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 examples 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 $\lambda$-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 $\text{seg}[n] = \forall k. \text{list}[k] \rightarrow \text{list}[n+k]$ 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.
The table below presents the results for various modules, including the number of definitions (#Defs), the mean runtime (R), and the total runtime (T) in milliseconds.

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

Table 4: Case Studies

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

**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 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).

```plaintext
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 \( \lambda \) (the label \( \text{lam} \)) or an application (label \( \text{app} \)). In case of \( \text{lam} \), after proving \( n > 0 \), it expects an expression of size \( n_1 \) as an argument and then behaves like the body of the \( \lambda \)-abstraction of size \( n_1 + n - 1 \). In case of \( \text{app} \), 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 \( \text{val}{n} \), a value can only be a \( \lambda \)-expression (label \( \text{app} \) missing).

**Trie Data Structure.**

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 also obtain a good characterization of the necessary work—the data structure is quite efficient (in theoretical terms). We start with binary numbers where each bit carries potential \( p \).

```plaintext
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 } 
```

A trie is represented by the type \( \text{trie}[n] \) where \( n \) is the number of elements in the current multiset. When inserting a number it updates to \( \text{trie}[n+1] \). 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 \( \text{trie}[n - m] \) 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 \( b_0 \) or \( b_1 \). When we reach the end of the sequence of bits \( e \) we increase the multiplicity at the leaf we have reached. As we traverse the trie, we need to construct new intermediate nodes in case we encounter a leaf. These operations require 4 messages per bit, so the input number should have potential of 4 per bit. For deletion, we need one more because we need to communicate the answer back to the client, so 5 units per bit. For simplicity, we therefore uniformly require 5 units of potential per bit when adding a number to the trie and “burn” the extra unit during insertion.

\[
\text{type trie } \{n\} =
&\{\text{ins}: \langle\{\text{4}\}\rangle \mid !k. \text{bin}(k)\{5\} -o \text{trie}\{n+1\},
\quad \text{del}: \langle\{\text{5}\}\rangle \mid !k. \text{bin}(k)\{6\} -o ?m. ?\{m\leq n\}. \text{bin}(m)\{0\} * \text{trie}\{n-m\}\}
\]

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

\[
\text{decl leaf : . |- (t : trie\{0\})}
\]
\[
\text{decl node\{n0\}\{m\}\{n1\}}:
\quad (l : \text{trie}\{n0\}) (c : \text{ctr}\{m\}) (r : \text{trie}\{n1\}) \mid - (t : \text{trie}\{n0+m+n1\})
\]

The source code is available at examples/trie-work.rast.

## 7 Conclusion

This paper describes the Rast programming language. In particular, we focused on the concrete syntax, type checker and equality, the refinement layer \([7]\), and its applicability to work \([6]\) and time analysis \([5]\). The refinements rely on an arithmetic solver based on Cooper’s algorithm \([3]\). The interpreter uses the shared memory semantics introduced in recent work \([16]\). 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 time bounds for several examples, all of which have been verified with our system, and are available in an open-source repository.

## References


