# Work Analysis with Resource-Aware Session Types

Ankush Das, Jan Hoffmann and Frank Pfenning

Carnegie Mellon University, Pittsburgh, PA, USA

**Abstract.** While there exist several successful techniques for supporting programmers in deriving static resource bounds for sequential code, analyzing the resource usage of message-passing concurrent processes poses additional challenges. To meet these challenges, this article presents an analysis for statically deriving worst-case bounds on the total work performed by message-passing processes. To decompose interacting processes into components that can be analyzed in isolation, the analysis is based on novel resource-aware session types, which describe protocols and resource contracts for inter-process communication. A key innovation is that both messages and processes carry potential to share and amortize cost while communicating. To symbolically express resource usage in a setting without static data structures and intrinsic sizes, resource contracts describe bounds that are functions of interactions between processes. Resource-aware session types combine standard binary session types and type-based amortized resource analysis in a linear type system. This type system is formulated for a core session-type calculus of the language SILL and proved sound with respect to a multiset-based operational cost semantics that tracks the total number of messages that are exchanged in a system. The effectiveness of the analysis is demonstrated by analyzing standard examples from amortized analysis and the literature on session types and by a comparative performance analysis of different concurrent programs implementing the same interface.

## 1 Introduction

In the past years, there has been increasing interest in supporting developers to statically reason about the resource usage of their code. There are different approaches to the problem that are based on type systems [30,37,18,15,27], abstract interpretation [25,5,16], recurrence relations [20,4,36], termination analysis [46,11,8,31], and other techniques [14,19]. Among the applications of this research we find the prevention of side channels that leak secret information [38,6,35], identification of complexity bugs [39], support of scheduling decisions [1], and help in profiling [26].

While there has been great progress in analyzing sequential code, relatively little research has been done on analyzing the resource consumption of concurrent and distributed programs [22,3,2]. The lack of analysis tools is in sharp contrast to the need of programming language support in this area: concurrent and distributed programs programs are both increasingly pervasive and particularly difficult to analyze. For shared memory concurrency, we need to precisely predict scheduling to account for synchronization cost. Similarly, the interactive nature of message-passing systems makes it difficult to decompose the system into

components that can be analyzed in isolation. After all, the resource usage of each component crucially depends on its interactions with the world.

In this paper, we study the foundations of worst-case resource analysis for message-passing processes. A key idea of our approach is to rely on *resource-aware session types* to describe structure, protocols, and resource bounds for inter-process communication that we can use to perform a compositional and precise amortized analysis. *Session types* [32,33,12,13,45] prescribe bidirectional communication protocols for message-passing processes. *Binary session types* govern the interaction of two processes along a single channel, prescribing complementary send and receive actions for the processes at the two endpoints of a channel. We use such protocols as the basis of resource usage contracts that not only specify the type but also the potential of a message that is sent along a channel. The potential (in the sense of classic amortized analysis [43]) may be spent sending other messages as part of the network of interacting processes, or maintained locally for future interactions. Resource analysis is static, using the type system, and the runtime behavior of programs is not affected.

We focus on bounds on the total work that is performed by a system, counting the number of messages that are exchanged. While this alone does not account yet for the concurrent nature of message-passing programs it constitutes a significant and necessary first step. The bounds we derive are also useful in their own right. For example, the information can be used in scheduling decisions, to bound the number of messages that are sent along a specific channel, or to statically decide whether we should spawn a new thread of control or execute sequentially when possible. Additionally, bounds on the work of a process can also serve as input to a Brent-style theorem [10] that relates the complexity of the execution of a program on a $k$-processor machine to the program's work (the focus of this paper) and span (the resource usage if we assume an unlimited number of processors). We are working on a companion paper for deriving bounds on the span, which is both conceptually and technically quite different.

Our analysis is based on a linear type system that combines standard binary session types as available in the SILL language [44,40], and type-based amortized resource analysis [30,27]. Both techniques are based on linear or affine type systems, making their combination natural. Each session type constructor is decorated with a natural number that declares a potential that must be transferred (conceptually!) along with the corresponding message. Since the interface to a process is characterized entirely by the resource-aware session types of the channels it can interact with, this design provides for a compositional resource specification. For closed programs (which evolve into a closed network of interacting processes) the bound then becomes a single constant. In addition to the natural compositionality of type systems we also preserve the good support for deriving resource annotations via LP solving which is a key feature of type-based amortized analysis. While we have not yet implemented a type inference algorithm, we designed the system with support for type inference in mind. Moreover, resource-aware session types integrate well with type-based amortized analysis for functional programs because they are based on compatible logical foundations

(intuitionistic linear logic and intuitionistic logic, respectively), as exemplified in the design of SILL [44,40] that combines them monadically.

A conceptual challenge of the work is to express symbolic bounds in a setting without static data structures and intrinsic sizes. Our innovation is that resource-aware session types describe bounds that are functions of interactions (i.e., messages sent) along a channel. A major technical challenge is to account for the global number of messages sent with local derivation rules: Operationally, local message counts are forwarded to a parent process when a sub-process terminates. As a result, local message counts are incremented by sub-processes in a rather non-local fashion. Our solution is that both messages and processes carry potential to share and amortize the cost of a terminating sub-process proactively as a side-effect of the communication.

Our main contributions are as follows. We present the first session type system for deriving parametric bounds on the resource usage of message-passing processes. We prove the nontrivial soundness of type system with respect to an operational cost semantics that tracks the total number of messages exchanged in a network of communicating process. We demonstrate the effectiveness of the technique by deriving tight bounds for some standard examples from amortized analysis and the literature on session types. We also show how resource-aware session types can be used to specify and compare the performance characteristics of different implementations of the same protocol. The analysis is currently manual, with automation left for future work, as is a companion type system for deriving the *span* of concurrent computations in the same language.

## 2 Overview

In this section, we motivate and informally introduce our resource-aware session types and show how they can be used to analyze the resource usage of message-passing processes. We start with building some intuition about session types.

*Session Types.* Session types have been introduced by Honda [32] to describe the structure of communication just like standard data types describe the structure of data. We follow the approach and syntax of SILL [44,40] which is based on a Curry-Howard isomorphism between intuitionistic linear logic and session types, extended by recursively defined types and processes. In the intuitionistic approach, every channel has a *provider* and a *client*. We view the session type as describing the communication from the provider's point of view, with the client having to perform matching actions.

As a first simple example, we consider natural numbers in binary form. A process *providing* a natural number sends a stream of bits starting with the least significant bit. These bits are represented by messages $\mathsf{zero}$ and $\mathsf{one}$, eventually terminated by \$. Because the provider chooses which messages to send, we call this an *internal choice*, which is written as

$$\mathsf{bits} = \oplus\{\mathsf{zero} : \mathsf{bits}, \mathsf{one} : \mathsf{bits}, \$ : \mathbf{1}\} \ .$$

Here, $\oplus\{l_1 : A_1, \ldots, l_n : A_n\}$ is an n-ary, labelled generalization of $A \oplus B$ of linear logic, and $\mathbf{1}$ is the multiplicative unit of linear logic. Operationally, $\mathbf{1}$ means the provider has to send an *end* message, closing the channel and terminating the communication session. For example, the number $6 = (110)_2$ would be represented by the sequence of messages $\mathsf{zero}, \mathsf{one}, \mathsf{one}, \$, \mathit{end}$.

The session type does not prescribe any particular implementation only the interface to a process. In this example, a client of a channel $c : \mathsf{bits}$ has to branch on whether it receives $\mathsf{zero}$, $\mathsf{one}$, or \$. Note that as we proceed in a session, the type of a channel must change according to the protocol. For example, if a client receives the message \$ along $c : \mathsf{bits}$ then $c$ must afterwards have type $\mathbf{1}$. The next message along $c$ must be *end* and we have to wait for that after receiving \$ so the session is properly closed.

As a second example we describe the interface to a counter. As a client, we can repeatedly send $\mathsf{inc}$ messages to a counter, until we want to read its value and send $\mathsf{val}$. At that point the counter will send a stream of bits representing its value as prescribed by the type $\mathsf{bits}$. From the provider's point of view, a counter presents an *external choice*, since the client chooses between $\mathsf{inc}$ or $\mathsf{val}$.

$$\mathsf{ctr} = \&\{\mathsf{inc} : \mathsf{ctr}, \mathsf{val} : \mathsf{bits}\}$$

The type former $\&\{l_1 : A_1, \ldots, l_n : A_n\}$ is an n-ary labelled generalization of $A \& B$ of linear logic. Operationally, the provider must branch based on which of the labels $l_i$ it receives. After receiving $l_k$ along a channel $c : \&\{l_1 : A_1, \ldots, l_n : A_n\}$, communication along $c$ proceeds at type $A_k$.

Such type formers can be arbitrarily nested to allow more complex bidirectional protocols. Consider for example the store protocol, which is defined by the following type.

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

A provider of a channel $c$ of type $\mathsf{store}_A$ either accepts an $\mathsf{ins}$ or $\mathsf{del}$ message. If it receives the $\mathsf{ins}$ message, the type of $c$ is now $A \multimap \mathsf{store}_A$, the implication of linear logic. It means the provider now receives a *channel* of type $A$ along $c$ and then behaves again like a store. If it receives a $\mathsf{del}$ message, it responds with either the message $\mathsf{none}$ or the message $\mathsf{some}$ (an internal choice, $\oplus$). If it sends $\mathsf{none}$ it next must send an *end* message and terminate. If it sends $\mathsf{some}$, the type of $c$ is now $A \otimes \mathsf{store}_A$. This corresponds to the multiplicative conjunction of linear logic and, operationally, requires the provider to now send a channel of type $A$ and then behave again as $\mathsf{store}_A$. Linearity of the type system guarantees that the channels retrieved from a store of this type are some permutation of the channels inserted. It may, for example, behave as a stack or a queue (as explained in Section 7).

*Modeling a binary counter.* We will now describe an implementation of a counter and use our resource-aware session types to analyze its resource usage. Like in the rest of the paper, the resource we are interested in is the total number of messages sent along all channels in the system.

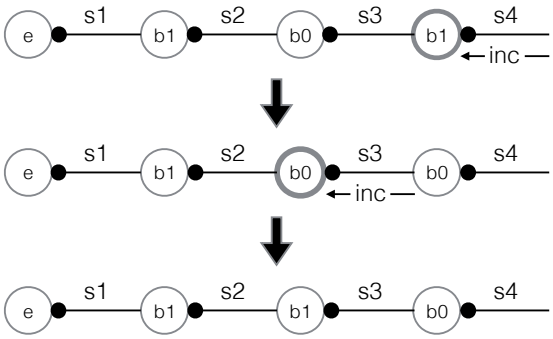


**Fig. 1.** A binary counter system representing $5 = (101)_2$ with the messages triggered when an inc message is received along $s_4$.

A well-known example of amortized analysis counts the number of bits that must be flipped to increment a counter. It turns out the amortized cost per increment is 2, so that $n$ increments require at most $2n$ bits to be flipped. We can see this by introducing a potential of 1 for every bit that is 1 and using the potential to *pay* for the expensive case in which and increment triggers many flips. When the lowest bit is zero, we flip it to one (costing 1) and also store a remaining potential of 1 with this bit. When the lowest bit is one we use the stored potential to flip the bit back to zero (with no stored potential) and use the remaining potential of 2 for incrementing the higher bits.

We model a binary counter as a chain of processes where each process represents a single bit (process $b0$ or $b1$) with a final process $e$ at the end. Each of the processes in the chain *provides* a channel of the ctr type, and each (except the last) also *uses* a channel of this type representing the higher bits. For example, in the first chain in Figure 1, the process $b0$ offers along channel $s_3$ and uses channel $s_2$. In our notation, we would write this as

$$\begin{aligned} \cdot &\vdash e :: (s_1 : \mathsf{ctr}) \\ s_1 : \mathsf{ctr} &\vdash b1 :: (s_2 : \mathsf{ctr}) \\ s_2 : \mathsf{ctr} &\vdash b0 :: (s_3 : \mathsf{ctr}) \\ s_3 : \mathsf{ctr} &\vdash b1 :: (s_4 : \mathsf{ctr}) \end{aligned}$$

We see that, logically, parallel composition with a private shared channel corresponds to an application of the cut rule. We do not show here the *definitions* of $e$, $b0$, and $b1$, which can be found in Figure 3. The only channel visible to an outside client (not shown) is $s_4$. Figure 1 shows the messages triggered if an increment message is received along $s_4$.

*Expressing resource bounds.* Our basic approach is that *messages carry potential* and *processes store potential*. This means the sender has to pay not just 1 unit for sending the message, but whatever additional units to amortize future costs. In the amortized analysis of the counter each bit flip corresponds exactly to an inc message, because that is what triggers a bit to be flipped. Our cost model

focuses on messages as prescribed by the session type and does not count other operations, such as spawning a new process or terminating a process. This choice is not essential to our approach, but convenient here.

To capture the informal analysis we need to express *in the type* that we have to send 1 unit of potential with the label $\mathsf{inc}$. We do this using a superscript indicating the required potential with the label, postponing for now the discussion of $\mathsf{val}$.

$$\mathsf{ctr} = \&\{\mathsf{inc}^1 : \mathsf{ctr}, \mathsf{val}^? : \mathsf{bits}\} \ .$$

When we assign types to the processes, we now use these more expressive types. We also indicate the potential stored in a particular process as a superscript on the turnstile.

$$t : \mathsf{ctr} \stackrel{0}{\vdash} b0 :: (s : \mathsf{ctr}) \tag{1}$$
$$t : \mathsf{ctr} \stackrel{1}{\vdash} b1 :: (s : \mathsf{ctr}) \tag{2}$$
$$\cdot \stackrel{0}{\vdash} e :: (s : \mathsf{ctr}) \tag{3}$$

With our formal typing rules (see Section 5) we can verify these typing constraints, using the definitions of $b0$, $b1$, and $e$. Informally, we can reason as follows:

**$b0$:** When $b0$ receives $\mathsf{inc}$ it receives 1 unit of potential. It continues as $b1$ (which requires no communication) which stores this 1 unit (as prescribed from the type of $b1$ in Equation 2).

**$b1$:** When $b1$ receives $\mathsf{inc}$ it receives 1 unit of potential which, when combined with the stored one, makes 2 units. It needs to send an $\mathsf{inc}$ messages which consumes these 2 units (1 to send the message, and 1 to send along a potential of 1 as prescribed in the type). It has no remaining potential, which is sufficient because it transitions to $b0$ which stores no potential (inferred from the type of $b0$ in Equation 1).

**$e$:** When $e$ receives $\mathsf{inc}$ it receives 1 unit of potential. It spawns a new process $e$ and continues as $b1$. Spawning a process is free, and $e$ requires no potential, so it can store the potential it received with $b1$ as required.

How do we handle the type annotation $\mathsf{val}^? : \mathsf{bits}$ of the label $\mathsf{val}$? Recall that $\mathsf{bits} = \oplus\{\mathsf{zero} : \mathsf{bits}, \mathsf{one} : \mathsf{bits}, \$ : \mathbf{1}\}$. In our implementation, upon receiving a $\mathsf{val}$ message, a $b0$ or $b1$ process will first respond with $\mathsf{zero}$ or $\mathsf{one}$. It then sends $\mathsf{val}$ along the channel it uses (representing the higher bits of the number) and terminates by *forwarding* further communication to the higher bits in the chain. The $e$ process will just send \$ and *end*, indicating the empty stream of bits.

We know we will have enough potential to carry out the required send operations if each process ($b0$, $b1$, and $e$) carries an additional 2 units of potential. We could impart these with the $\mathsf{inc}$ and $\mathsf{val}$ messages by sending 2 more units with $\mathsf{inc}$ and 2 units with $\mathsf{val}$. That is, the following type is correct:

$$\mathsf{bits} = \oplus\{\mathsf{zero}^0 : \mathsf{bits}, \mathsf{one}^0 : \mathsf{bits}, \$^0 : \mathbf{1}^0\}$$
$$\mathsf{ctr} = \&\{\mathsf{inc}^3 : \mathsf{ctr}, \mathsf{val}^2 : \mathsf{bits}\}$$

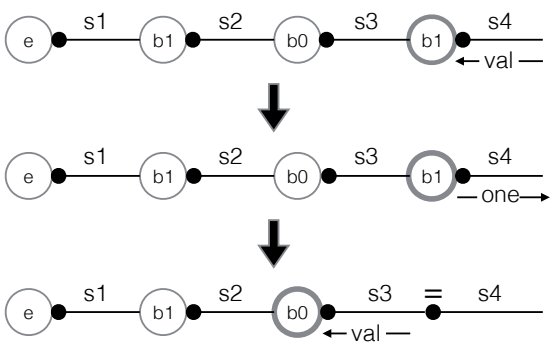


**Fig. 2.** A binary counter system representing $5 = (101)_2$ with the messages triggered when a val message is received along $s_4$.

Here, the superscript 0 in the type of bits indicates that the corresponding messages carry no potential.

However, this type is a gross over-approximation. The processes of a counter of value $n$, would carry $2n$ additional units of potential while only $2\lceil\log(n+1)\rceil+2$ are needed. To obtain this more precise bound, we need *families of session types.*

*A more precise analysis.* A more precise session type for this example requires that *in the type* we can refer either to the number of bits in the representation of a number or its value. This form of internal measure is needed only for type-checking purposes, not at runtime. It is also not intrinsically tied to a property of a representation, the way the length of a list in a functional language is tied to its memory requirements. We indicate these measures using square brackets, so that $\mathsf{ctr}[n]$ is a family of types, and $\mathsf{ctr}[0]$, for example, is a counter with value 0. Such type refinements have been considered in the literature on session types (see, for example, [24]) with respect to type-checking and inference. Here, we treat it as a meta-level notation to denote families of types. Following the reasoning above, we obtain the following type:

$$\begin{aligned}\mathsf{bits} &= \oplus\{\mathsf{zero}^0 : \mathsf{bits}, \mathsf{one}^0 : \mathsf{bits}, \$^0 : \mathbf{1}^0\}\\ \mathsf{ctr}[n] &= \&\{\mathsf{inc}^1 : \mathsf{ctr}[n+1], \mathsf{val}^{2\lceil\log(n+1)\rceil+2} : \mathsf{bits}\}\end{aligned}$$

To check the types of our implementation, we need to revisit and refine the typing of the $b0$, $b1$ and $e$ processes.

$$\begin{aligned}t : \mathsf{ctr}[n] &\stackrel{0}{\vdash} b0 :: (s : \mathsf{ctr}[2n])\\ t : \mathsf{ctr}[n] &\stackrel{1}{\vdash} b1 :: (s : \mathsf{ctr}[2n+1])\\ &\stackrel{0}{\vdash} e :: (s : \mathsf{ctr}[0])\end{aligned}$$

Our type system verifies these types against the implementation of $b0$, $b1$, and $e$ (see Section 3). The typing rules reduce the well-typedness of these processes to arithmetic inequalities which we can solve by hand, for example, using that $\log(2n) = \log(n) + 1$. The intrinsic measure $n$ and the particular potential

annotations are not automatically derived but come from our insight about the nature of the algorithms and programs.

Before introducing the formalism in which the programs are expressed, together with the typing rules that let us perform rigorous amortized analysis of the code (as expressed in the soundness theorem in Section 6), we again emphasize the *compositional nature* of the way resource bounds are expressed in the types themselves and in the typing judgments for process expressions. Of course, they reveal some intensional property of the implementations, namely a bound on its cost, so different implementations of the same plain session type may have different resource annotations.

The typing derivation provides a proof certificate on the resource bound for a process. For closed processes, usually typed as

$$\cdot \vdash^{p} Q :: (c : \mathbf{1})$$

the number $p$ provides a worst case bound for the number of messages sent during the computation of $Q$, which always ends with the process sending *end* along $c$.

## 3 Resource-Aware SILL

We briefly introduce the linear, process-only fragment of SILL [44,40], which integrates functional and concurrent computation. A program in SILL is a collection of processes exchanging messages through channels. A new process is *spawned* by invoking a process definition, which also creates a fresh channel that is *provided* by the new process. The process that invokes a process definition becomes the *client* of the new process, communicating with it according to the session types of the channel. The exacting nature of linear typing provides strong guarantees, including session fidelity (a form of preservation) and absence of deadlocks (a form of progress).

We present an overview of the session types in SILL with a brief description of their communication protocol. They follow the type grammar below.

$$S, T ::= V \mid \oplus\{l_i : S\} \mid \&\{l_i : S\} \mid S \multimap T \mid S \otimes T \mid \mathbf{1}$$

$V$ denotes a type variable here. Types may be defined mutually recursively in a global signature, where type definitions are constrained to be *contractive* [21]. This allows us to treat them equi-recursively, which means we can silently replace a type variable by its definition for the purpose of type-checking.

Internal choice $S \oplus T$ and external choice $S \& T$ have been generalized to n-ary labeled sums $\oplus\{l_i : S_i\}_{i \in I}$ and $\&\{l_i : S_i\}_{i \in I}$ (for some index set $I$) respectively. As a provider of internal choice $\oplus\{l_i : S_i\}_{i \in I}$, a process can send one of the labels $l_i$ to its client. As a dual, a provider of external choice $\&\{l_i : S_i\}_{i \in I}$ receives one of the labels $l_i$ which is sent by its client. We require external and internal choice to comprise at least one label, otherwise there would exist a linear channel without observable communication along it, which is computationally uninteresting and would complicate our proofs. The connectives $\otimes$ and $\multimap$ are used to send channels

| Type (current) | Continuation | Process Term (current) | Continuation | Description |
|---|---|---|---|---|
| $c : \oplus\{l_i^{q_i} : S_i\}$ | $c : S_k$ | $c.l_k\ ;\ P$ | $P$ | provider sends label $l_k$ along $c$ with potential $q_k$ |
| | | $\mathsf{case}\ c\ (l_i \Rightarrow Q_i)_{i \in I}$ | $Q_k$ | client receives label $l_k$ along $c$ with potential $q_k$ |
| $c : \&\{l_i^{q_i} : S_i\}$ | $c : S_k$ | $\mathsf{case}\ c\ (l_i \Rightarrow P_i)_{i \in I}$ | $P_k$ | provider receives label $l_k$ along $c$ with potential $q_k$ |
| | | $c.l_k\ ;\ Q$ | $Q$ | client sends label $l_k$ along $c$ with potential $q_k$ |
| $c : S \overset{q}{\otimes} T$ | $c : T$ | $\mathsf{send}\ c\ w\ ;\ P$ | $P$ | provider sends channel $w : S$ along $c$ with potential $q$ |
| | | $y \leftarrow \mathsf{recv}\ c\ ;\ Q_y$ | $[w/y]Q_y$ | client receives channel $w : S$ along $c$ with potential $q$ |
| $c : S \overset{q}{\multimap} T$ | $c : T$ | $y \leftarrow \mathsf{recv}\ c\ ;\ P_y$ | $[w/y]P_y$ | provider receives channel $w : S$ along $c$ with potential $q$ |
| | | $\mathsf{send}\ c\ w\ ;\ Q$ | $Q$ | client sends channel $w : S$ along $c$ with potential $q$ |
| $c : \mathbf{1}^q$ | – | $\mathsf{close}\ c$ | – | provider sends *end* along $c$ with potential $q$ |
| | | $\mathsf{wait}\ c\ ;\ Q$ | $Q$ | client receives *end* along $c$ with potential $q$ |

**Table 1.** Linear resource-aware session types

via other channels. A provider of $S \otimes T$ sends a channel of type $S$ to its client and then behaves as a provider of $T$. A provider of $S \multimap T$ receives a channel of type $S$ from its client. The type of the provider and client changes consistently, and the process terms of a provider and client come in matching pairs.

Formally, the syntax of process expressions of Resource-Aware SILL is same as in SILL.

$$\begin{aligned} P, Q ::= {} & x \leftarrow \mathcal{X} \leftarrow \overline{y}\ ;\ Q \mid x \leftarrow y \mid x.l_k\ ;\ P \mid \mathsf{case}\ x\ (l_i \Rightarrow P) \\ & \mid \mathsf{send}\ x\ w \mid y \leftarrow \mathsf{recv}\ x\ ;\ P \mid \mathsf{close}\ x \mid \mathsf{wait}\ x\ ;\ P \end{aligned}$$

The first term $x \leftarrow \mathcal{X} \leftarrow \overline{y}\ ;\ Q$ invokes a process definition $\mathcal{X}$ to spawn a new process $P$, which uses the channels in $\overline{y}$ as a client and provides service along a fresh channel substituted for $x$ in $Q$. A forwarding process $x \leftarrow y$ (which provides channel $x$) identifies channels $x$ and $y$ and terminates. The effect is that clients of $x$ will afterwards communicate along $y$. We saw an example of its use in Figure 2. The rest of the program constructs concern communication between two processes and are guided by their corresponding session type. Table 1 provides an overview of session types, associated process terms, and their operational description (ignore the portions in red). For each connective in Table 1, the first line provides the perspective of the provider, while the second line provides that of the client. The first two columns present the type of the channel before (**current**) and after (**continuation**) the interaction. Similarly, the next two columns present the process terms before and after the interaction. Finally, the last column presents the operational description.

```
1:  (t : ctr[n]) ⊢[0] b0 :: (s : ctr[2n])
2:    s ← b0 ← t =
3:          case s (inc ⇒ s ← b1 ← t        %  (t : ctr[n]) ⊢[1] s : ctr[2n + 1]
4:                  | val ⇒ s.zero ;        %  (t : ctr[n]) ⊢[2⌈log(2n+1)⌉+2−1] s : bits
5:                          t.val ;         %  (t : bits) ⊢[2⌈log(2n+1)⌉+1−2⌈log(n+1)⌉−3] s : bits
6:                          s ← t)          %  (t : bits) ⊢[0] s : bits

7:  (t : ctr[n]) ⊢[1] b1 :: (s : ctr[2n + 1])
8:    s ← b1 ← t =
9:          case s (inc ⇒ t.inc ;           %  (t : ctr[n + 1]) ⊢[0] s : ctr[2n + 2]
10:                         s ← b0 ← t      %
11:                 | val ⇒ s.one ;         %  (t : ctr[n]) ⊢[2⌈log(2n+2)⌉+2−1] s : bits
12:                         t.val ;         %  (t : bits) ⊢[2⌈log(2n+2)⌉+1−2⌈log(n+1)⌉−3] s : bits
13:                         s ← t)          %  (t : bits) ⊢[0] s : bits

14: · ⊢[0] e :: (s : ctr[0])
15:   s ← e =
16:         case s (inc ⇒ t ← e ;                %  (t : ctr[0]) ⊢[1] (s : ctr[1])
17:                     s ← b1 ← t
18:                 | val ⇒ s.e ;                %  · ⊢[2⌈log(0+1)⌉+2−1] s : 1^0
19:                     close s)
```

**Fig. 3.** Implementations for the $b0$, $b1$ and $e$ processes with their type derivations demonstrating the binary counter.

We conclude by illustrating the syntax, types and semantics of SILL using a simple example. Recall the counter protocol (ignoring the resource annotations in red):

$$\begin{aligned}\mathsf{bits} &= \oplus\{\mathsf{zero}^0 : \mathsf{bits}, \mathsf{one}^0 : \mathsf{bits}, \$^0 : \mathbf{1}^0\}\\ \mathsf{ctr}[n] &= \&\{\mathsf{inc}^1 : \mathsf{ctr}[n+1], \mathsf{val}^{2\lceil\log(n+1)\rceil+2} : \mathsf{bits}\}\end{aligned}$$

The type prescribes that a process providing service along a channel of type ctr will either receive an inc or a val label. If it receives an inc label, the channel will recurse back to the ctr type. If it receives a val label, it will continue by providing bits, sending a sequence of labels zero and one closed out with \$ and *end*.

We present implementations of the $b0$, $b1$ and $e$ processes respectively that were analyzed in Section 2 in Figure 3. In the comments we show the types of the channels after the interaction on each line (again ignoring the annotations in red). Since the $b0$ process provides and external choice along $s$, $b0$ needs to branch based on the label received (line 3). If it receives the label inc, the type of the channel updates to ctr, as indicated on the typing in the comment. At this point, we can spawn the $b1$ process since the type on line 3 matches with the type of the $b1$ process (line 7). If instead $b0$ receives the val label along $s$, it continues at type bits. It sends zero (since the lowest bit is indeed zero). It then requests the value of the higher bits by sending val along channel $t$. Now both $s$ and $t$ have type bits (indicated in the typing on line 4) and $b0$ can terminate by forwarding further communication along $s$ to $t$.

The $b1$ process operates similarly, taking care to handle the carry upon increment by sending an $\mathsf{inc}$ label along $t$. The $e$ process spawns a new $e$ process and continues as $b1$ upon receiving the label $\mathsf{inc}$ and closes the channel after sending \$ when receiving $\mathsf{val}$.

## 4 Cost Semantics

We present an operational cost semantics for Resource-Aware SILL that tracks the total work performed by a system. Like previous work, our semantics is a substructural operational semantics [41] based on multiset rewriting [17] and asynchronous communication [40]. It can be seen as a combination of an asynchronous version of a recently introduced synchronous session-type semantics [9] with the cost tracking semantics of Concurrent C0 [42]. The technical advantage of our semantics is that it avoids the complex operational artifacts of Silva et al. [42] such as message buffers: processes and messages can be typed with exactly the same typing rules, changing only the cost metric.

We will only count communication costs, ignoring internal. To this end, we introduce 3 costs, $M^{\mathsf{label}}$, $M^{\mathsf{channel}}$ and $M^{\mathsf{close}}$, for labels, channels, and close messages, respectively. A concrete semantics can be obtained by setting appropriate values for each of those metrics. For instance, setting $M^{\mathsf{label}} = M^{\mathsf{channel}} = M^{\mathsf{close}} = 1$ will lead to counting the total number of messages exchanged.

Our cost semantics is asynchronous, that is, processes can continue their evaluation without wait after sending a message. In order to guarantee session fidelity the semantics must ensure that messages are received in the order they are sent. Intuitively, we can think of channels as FIFO message buffers, although we will formally define them differently. Synchronous communication can be implemented in our language in a type-safe, logically motivated manner exploiting adjoint shift operators (see [40]).

A collection of communicating process is called a *configuration*. A configuration is formally modelled as a multiset of propositions $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$. The predicate $\mathsf{proc}(c, w, P)$ describes a process executing process expression $P$ and providing channel $c$. The predicate $\mathsf{msg}(c, w, M)$ describes the message $M$ on channel $c$. In order to guarantee that messages are received in they order they are sent, only *a single message* can be on a given channel $c$. In order for computation to remain truly asynchronous, every send operation (except for $\mathsf{close}$) on a channel $c$ creates not only a fresh message, but also a fresh continuation channel $c'$ for the next message. This continuation channel is encoded within the message via a forwarding operation. Remarkably, this simple device allows us to assign session types to messages just as if they were processes! Since $M$ need only encode a message, it has a restricted grammar.

$$\begin{aligned} M ::= {} & c \leftarrow c' \mid c.l_k \ ; \ c \leftarrow c' \mid c.l_k \ ; \ c' \leftarrow c \\ & \mid \mathsf{send}\ c\ e \ ; \ c \leftarrow c' \mid \mathsf{send}\ c\ e \ ; \ c' \leftarrow c \mid \mathsf{close}\ c \end{aligned}$$

The work is tracked by the local counter $w$ in the $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$ propositions. For a process $P$, $w$ maintains the total work performed by $P$ so far.

$$\frac{\Sigma(\mathcal{X}) = x \leftarrow \mathcal{X} \leftarrow \overline{y} = P_{x,\overline{y}} \qquad \mathsf{proc}(d, w, x \leftarrow \mathcal{X} \leftarrow \overline{e}\ ;\ Q_x) \qquad (c \text{ fresh})}{\mathsf{proc}(c, 0, [c/x, \overline{e}/y]P_{x,\overline{y}}) \qquad \mathsf{proc}(d, w, [c/x]Q_x)}\ \mathsf{spawn}_c$$

$$\frac{\mathsf{proc}(c, w, c \leftarrow d)}{\mathsf{msg}(c, w, c \leftarrow d)}\ \mathsf{fwd}_s$$

$$\frac{\mathsf{proc}(d, w, P) \qquad \mathsf{msg}(c, w', c \leftarrow d)}{\mathsf{proc}(c, w + w', [c/d]P)}\ \mathsf{fwd}_r^+$$

$$\frac{\mathsf{proc}(e, w, P_c) \qquad \mathsf{msg}(c, w', c \leftarrow d)}{\mathsf{proc}(e, w + w', [d/c]P_c)}\ \mathsf{fwd}_r^-$$

$$\frac{\mathsf{proc}(c, w, c.l_k\ ;\ P) \qquad (c' \text{ fresh})}{\mathsf{proc}(c', w + M^{\mathsf{label}}, [c'/c]P) \qquad \mathsf{msg}(c, 0, c.l_k\ ;\ c \leftarrow c')}\ \oplus C_s$$

$$\frac{\mathsf{msg}(c, w, c.l_k\ ;\ c \leftarrow c') \qquad \mathsf{proc}(d, w', \mathsf{case}\ c\ (l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(d, w + w', [c'/c]Q_k)}\ \oplus C_r$$

$$\frac{\mathsf{proc}(c, w, \mathsf{send}\ c\ e\ ;\ P) \qquad (c' \text{ fresh})}{\mathsf{proc}(c', w + M^{\mathsf{channel}}, [c'/c]P) \qquad \mathsf{msg}(c, 0, \mathsf{send}\ c\ e\ ;\ c \leftarrow c')}\ \otimes C_s$$

$$\frac{\mathsf{msg}(c, w, \mathsf{send}\ c\ e\ ;\ c \leftarrow c') \qquad \mathsf{proc}(d, w', x \leftarrow \mathsf{recv}\ c\ ;\ Q_x)}{\mathsf{proc}(d, w + w', [c'/c]Q_e)}\ \otimes C_r$$

$$\frac{\mathsf{proc}(d, w, \mathsf{send}\ c\ e\ ;\ P) \qquad (c' \text{ fresh})}{\mathsf{proc}(d, w + M^{\mathsf{channel}}, [c'/c]P) \qquad \mathsf{msg}(c', 0, \mathsf{send}\ c\ e\ ;\ c' \leftarrow c)}\ \multimap C_s$$

$$\frac{\mathsf{msg}(c', w, \mathsf{send}\ c\ e\ ;\ c' \leftarrow c) \qquad \mathsf{proc}(c, w', x \leftarrow \mathsf{recv}\ c\ ;\ Q_x)}{\mathsf{proc}(c, w + w', [c'/c]Q_e)}\ \multimap C_r$$

$$\frac{\mathsf{proc}(c, w, \mathsf{close}\ c)}{\mathsf{msg}(c, w + M^{\mathsf{close}}, \mathsf{close}\ c)}\ \mathbf{1}C_s$$

$$\frac{\mathsf{msg}(c, w, \mathsf{close}\ c) \qquad \mathsf{proc}(d, w', \mathsf{wait}\ c\ ;\ Q)}{\mathsf{proc}(d, w + w', Q)}\ \mathbf{1}C_r$$

**Fig. 4.** Cost semantics tracking total work for programs in SILL

When a process sends a message (i.e. creates a new msg predicate), we increment its counter $w$ by the cost for sending. When a processes terminates we remove the respective predicate from the configuration but need to preserve the work done by the process. A process can terminate either by sending a close message, or by forwarding. In either case, we can conveniently preserve the process' work in the msg predicate to pass it on to the client process.

The semantics is defined by a set of rules rewriting the configuration that *consume* the proposition in the premise of the rule and *produce* the propositions in the conclusion (rules should be read top-down!). A step consists of non-deterministic application of a rule whose premises matches a part of the configuration. Consider for instance the rule $C_s$ that describes a client that sends

label $l_k$ along channel $c$.

$$\frac{\mathsf{proc}(d, w, c.l_k\ ;\ P) \qquad (c' \text{ fresh})}{\mathsf{proc}(d, w + M^{\mathsf{label}}, [c'/c]P) \qquad \mathsf{msg}(c', 0, c.l_k\ ;\ c' \leftarrow c)}\ \&C_s$$

The rule can be applied to every proposition of the form $\mathsf{proc}(d, w, c.l_k\ ;\ P)$. When applying the rule, we generate a fresh channel continuation channel $c'$ and replace the premise by propositions $\mathsf{proc}(d, w+M^{\mathsf{label}}, [c'/c]P)$ and $\mathsf{msg}(c', 0, c.l_k\ ;\ c' \leftarrow c)$. The message predicate contains the process $c.l_k\ ;\ c' \leftarrow c$ which will eventually deliver the message to the provider along $c$ and will continue communication along $c'$ (which is achieved by $c' \leftarrow c$). The work of the process is incremented by $M^{\mathsf{label}}$ to account for the sent message, while the work of the message is 0.

Conversely, the rule $\&C_r$ defines how a provider receives a label $l_k$ along $c$.

$$\frac{\mathsf{msg}(c', w, c.l_k\ ;\ c' \leftarrow c) \qquad \mathsf{proc}(c, w', \mathsf{case}\ c\ (l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(c, w + w', [c'/c]Q_k)}\ \&C_r$$

The rule replaces the msg and proc propositions in the configuration that match the premises, with the single proc proposition in the conclusion. Since the provider receives the label $l_k$, it continues as $Q_k$. However, we replace $c$ with $c'$ in $Q_k$ since the forwarding $c' \leftarrow c$ in the message process tells us that the next message will arrive on channel $c'$. If there is any work $w$ encoded in the message, it is transferred to the recipient. This is somewhat more general than necessary for this particular rule, since in the current system the work $w$ in a label-sending message $c.l_k$ will always be 0.

The rest of the rules of cost semantics are given in Figure 4. The rule $\mathsf{spawn}_c$ describes the creation of a new channel $c$ along with a spawning new process $\mathcal{X}$ implemented by $P_c$. This implementation is looked up in a signature for the semantics $\Sigma$ (maps process names to the implementation code). The new process is spawned with 0 work (as it has not sent any messages so far), while $Q_c$ continues with the same amount of work. In the rule $\mathsf{fwd}_s$ a forwarding process creates a *forwarding message* and terminates. The work carried by this special message is the same as the work done by the process, now defunct. A forwarding message form does not carry any real information (except for the work $w$!); it just serves to identify the two channels $c$ and $d$. In an implementation this could be as simple as concatenating two message buffers. We therefore do not count forwarding messages when computing the work. Another reason forward messages are special is that unlike all other forms of messages, they are neither prescribed by nor manifest in a channel's type. In our formal rules, the forwarding message can be absorbed either into the client ($\mathsf{fwd}_r^+$) or provider ($\mathsf{fwd}_r^-$), in both cases preserving the total amount of work.

The rules of the cost semantics are successively applied to a configuration until the configuration becomes empty or the configuration is stuck and none of the rules can be applied. At any point in this local stepping, the total work performed by the system can be obtained by summing the local counters $w$ for each predicate in the configuration. We will prove in Section 6 that this total work can be upper bounded by the initial potential of the configuration that is typed in our resource-aware type system.

## 5 Type System

We now present the resource-aware type system of our language which extends the linear-only fragment of SILL [44,40] with resource annotations. It is in turn based on intuitionistic linear logic [23] with sequents of the form

$$A_1, A_2, \ldots, A_n \vdash A$$

where $A_1, \ldots A_n$ are the linear antecedents and $A$ is the succedent. Under the Curry-Howard isomorphism for intuitionistic linear logic, propositions are related to session types, proofs to processes, and cut reduction in proofs to communication. Appealing to this correspondence, we assign a process term $P$ to the above judgment and label each hypothesis as well as the conclusion with a channel.

$$(x_1 : A_1), (x_2 : A_2), \ldots, (x_n : A_n) \vdash P :: (x : A)$$

The resulting judgment states that process $P$ provides a service of session type $A$ along channel $x$, using the services of session types $A_1, \ldots, A_n$ provided along channels $x_1, \ldots, x_n$ respectively. The assignment of a channel to the conclusion is convenient because, unlike functions, processes do not evaluate to a value but continue to communicate along their providing channel once they have been created. For the judgment to be well-formed, all the channel names have to be distinct. Whether a session type is used or provided is determined by its positioning to the left or right, respectively, of the turnstile.

Resource-aware session types are given by the following grammar.

$$S, T ::= V \mid \oplus\{l_i^{q_i} : S\} \mid \&\{l_i^{q_i} : S\} \mid S \overset{q}{\multimap} T \mid S \overset{q}{\otimes} T \mid \mathbf{1}^q$$

Here, $V$ is a type variable. The meaning of the types and the process terms associated with it are defined in Table 1 (annotations and descriptions pertaining to potentials are marked in red).

The typing judgment of Resource-Aware SILL has the form

$$\Sigma; \Omega \overset{q}{\vdash} P :: (x : S)\ .$$

Intuitively, the judgment describes a process in state $P$ using the context $\Omega$ and signature $\Sigma$ and providing service along channel $x$ of type $S$. In other words, $P$ is the provider for channel $x : S$, and a client for all the channels in $\Omega$. The resource annotation $q$ is a natural number and defines the potential stored in the process $P$. $\Sigma$ defines the signature containing type and process definitions. These definitions are needed to typecheck processes which refer to a type definition or spawn a new process.

The signature $\Sigma$ is defined as a possibly infinite set of type definitions $V = S_V$ and process definitions $x : S \leftarrow \mathcal{X} \mathbin{@} q \leftarrow \overline{y : W} = P_{x,\overline{y}}$. The equation $V = S_V$ is used to define the type variable $V$ as $S_V$. We treat such definitions *equirecursively*. For instance, $\mathsf{ctr}[n] = \&\{\mathsf{inc}^1 : \mathsf{ctr}[n+1], \mathsf{val}^{2\lceil \log(n+1) \rceil + 2} : \mathsf{bits}\}$ exists in the signature for all $n \in \mathbb{N}$ for the binary counter system. Type families

$$\frac{q \geq p + r_k + M^{\mathsf{label}} \quad \Sigma\ ;\ \Omega \Vdash^{p} P :: (x : S_k) \quad (k \in I)}{\Sigma\ ;\ \Omega \Vdash^{q} (x.l_k\ ;\ P) :: (x : \oplus\{l_i^{r_i} : S_i\}_{i \in I})}\ \oplus R_k$$

$$\frac{q + r_i \geq q_i \quad \Sigma\ ;\ \Omega\ (x : S_i) \Vdash^{q_i} Q_i :: (z : U) \quad (\forall i \in I)}{\Sigma\ ;\ \Omega\ (x : \oplus\{l_i^{r_i} : S_i\}_{i \in I}) \Vdash^{q} \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i \in I} :: (z : U)}\ \oplus L$$

$$\frac{q + r \geq p \quad \Sigma\ ;\ \Omega\ (y : S) \Vdash^{p} P_y :: (x : T)}{\Sigma\ ;\ \Omega \Vdash^{q} (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x : S \overset{r}{\multimap} T)}\ \multimap R$$

$$\frac{q \geq p + r + M^{\mathsf{channel}} \quad \Sigma\ ;\ \Omega\ (x : T) \Vdash^{p} Q :: (z : U)}{\Sigma\ ;\ \Omega\ (w : S)\ (x : S \overset{r}{\multimap} T) \Vdash^{q} (\mathsf{send}\ x\ w\ ;\ Q) :: (z : U)}\ \multimap L$$

$$\frac{q \geq p + r + M^{\mathsf{channel}} \quad \Sigma\ ;\ \Omega \Vdash^{p} P :: (x : T)}{\Sigma\ ;\ (w : S)\ \Omega \Vdash^{q} \mathsf{send}\ x\ w\ ;\ P :: (x : S \overset{r}{\otimes} T)}\ \otimes R$$

$$\frac{q + r \geq p \quad \Sigma\ ;\ \Omega\ (y : S)\ (x : T) \Vdash^{p} Q_y :: (z : U)}{\Sigma\ ;\ \Omega\ (x : S \overset{r}{\otimes} T) \Vdash^{q} y \leftarrow \mathsf{recv}\ x\ ;\ Q_y :: (z : U)}\ \otimes L$$

$$\frac{q \geq r + M^{\mathsf{close}}}{\Sigma\ ;\ \cdot \Vdash^{q} \mathsf{close}\ x :: (x : \mathbf{1}^{r})}\ \mathbf{1}R \qquad \frac{q + r \geq p \quad \Sigma\ ;\ \Omega \Vdash^{p} Q :: (z : U)}{\Sigma\ ;\ \Omega\ (x : \mathbf{1}^{r}) \Vdash^{q} \mathsf{wait}\ x\ ;\ Q :: (z : U)}\ \mathbf{1}L$$

$$\frac{(V = S_V) \in \Sigma \quad \Sigma\ ;\ \Omega \Vdash^{q} P :: (x : S_V)}{\Sigma\ ;\ \Omega \Vdash^{q} P :: (x : V)}\ \mu R$$

$$\frac{(V = S_V) \in \Sigma \quad \Sigma\ ;\ \Omega\ (x : S_V) \Vdash^{q} P :: (z : U)}{\Sigma\ ;\ \Omega\ (x : V) \Vdash^{q} P :: (z : U)}\ \mu L$$

**Fig. 5.** Typing rules for session-typed programs (remaining rules are given in the text)

exist only at the meta-level and $\mathsf{ctr}[n]$ is treated as a regular type variable. The process definition $x : S \leftarrow \mathcal{X} \mathbin{@} q \leftarrow \overline{y : W} = P_{x,\overline{y}}$ defines a (possibly recursive) process named $\mathcal{X}$ that is implemented by $P_{x,\overline{y}}$ provides along channel $x : S$, and uses the channels $\overline{y : W}$ as a client. The process also stores a potential $q$, shown as $\mathcal{X} \mathbin{@} q$ in the signature. For instance, for the binary counter system, $s : \mathsf{ctr}[2n] \leftarrow b0 \mathbin{@} 0 \leftarrow t : \mathsf{ctr}[n] = P_{s,t}$ ($P_{s,t}$ defines the implementation of $b0$) exists in the signature for all $n \in \mathbb{N}$.

Messages are typed differently from processes as their work counters $w$ (introduced in the predicate $\mathsf{msg}(c, w, M)$) are not incremented when they actually deliver the message to the receiver. Hence, to type the messages, we define an auxiliary cost-free typing judgment, $\Sigma; \Omega \overset{q}{\underset{\mathsf{cf}}{\Vdash}} P :: (x : S)$, which follows the same typing rules as Figure 5, but with $M^{\mathsf{label}} = M^{\mathsf{channel}} = M^{\mathsf{close}} = 0$. This avoids paying the cost for sending a message twice. A fresh signature $\Sigma$ is used in the derivation of the cost-free judgment.

The idea of the type system is that each message carries potential and the sending process pays the potential along with the cost of sending a message from its local potential. The receiving process receives the potential when it receives the message and adds it to its local potential. For example, consider the rule $\&L_k$ for a client sending a label $l_k$ along channel $x$.

$$\frac{q \geq p + r_k + M^{\mathsf{label}} \quad \Sigma\ ;\ \Omega\ (x : S_k) \vdash^{p} Q :: (z : U)}{\Sigma\ ;\ \Omega\ (x : \&\{l_i^{r_i} : S_i\}) \vdash^{q} x.l_k\ ;\ Q :: (z : U)}\ \&L_k$$

Since the continuation $Q$ needs potential $p$ to typecheck, and the potential to be sent with the label is $r_k$, we need a total potential of at least $p+r_k+M^{\mathsf{label}}$, where $M^{\mathsf{label}}$ is the cost of sending a label. Hence, we get the constraint $q \geq p+r_k+M^{\mathsf{label}}$.

The rule $\&R$ describes a provider that is awaiting a message on channel $x$ and has local potential $q$ available.

$$\frac{q + r_i \geq q_i \quad \Sigma\ ;\ \Omega \vdash^{q_i} P_i :: (x : S_i) \quad (\forall i \in I)}{\Sigma\ ;\ \Omega \vdash^{q} \mathsf{case}\ x\ (l_i \Rightarrow P_i)_{i \in I} :: (x : \&\{l_i^{r_i} : S_i\})}\ \&R$$

The second premise tells us that the branch $P_i$ needs potential $q_i$ to typecheck. But the branch $P_i$ is reached after receiving the label $l_i$ with potential $r_i$. Hence, the initial potential $q$ must be able to cover the difference $q_i - r_i$. Since potential $q$ can typecheck all the branches, we get the constraint $q \geq q_i - r_i$ for all $i$.

To spawn a new process defined by $\mathcal{X}$, we split the context $\Omega$ into $\Omega_1$ $\Omega_2$, and we use $\Omega_1$ to type the newly spawned process and $\Omega_2$ for the continuation $Q_x$.

$$\frac{\begin{array}{c} r \geq p + q \quad x' : S \leftarrow \mathcal{X} \mathbin{@} p \leftarrow \overline{y' : W} = P_{x', \overline{y'}} \in \Sigma \\ \Omega_1 = \overline{y : W} \quad \Sigma\ ;\ \Omega_2\ (x : S) \vdash^{q} Q_x :: (z : U) \end{array}}{\Sigma\ ;\ \Omega_1\ \Omega_2 \vdash^{r} (x \leftarrow \mathcal{X} \leftarrow \overline{y}\ ;\ Q_x) :: (z : U)}\ \mathsf{spawn}$$

If the spawned process needs potential $p$ (indicated by the signature) and the continuation needs potential $q$ then the whole process needs potential $r \geq p + q$.

A forwarding process $x \leftarrow y$ terminates and its potential $q$ is lost. Since we do not count forwarding messages in our cost semantics, we don't need any potential to type the forward.

$$\frac{q \geq 0}{\Sigma\ ;\ y : S \vdash^{q} x \leftarrow y :: (x : S)}\ \mathsf{id}$$

The rest of the rules are given in Figure 5. They are similar to the discussed rules and we omit their explanation.

As an illustration, the resource-aware type for the binary counter was presented in Section 3 (marked in red). Also, Figure 3 provides the type derivation of the $b0$, $b1$ and $e$ processes (again marked in red). The annotations, along with the type derivation, prove that an increment has an amortized resource cost of 1 (potential annotation of inc in bits type) and reading a value has a resource cost of $2\lceil \log(n+1) \rceil + 2$.

$$\frac{}{\Sigma\ ;\ (\cdot) \overset{0}{\vDash} (\cdot) :: (\cdot)}\ \mathsf{emp} \qquad \frac{\Sigma\ ;\ \Omega \overset{S}{\vDash} \mathcal{C} :: \Omega' \qquad \Sigma\ ;\ \Omega' \overset{S'}{\vDash} \mathcal{C}' :: \Omega''}{\Sigma\ ;\ \Omega \overset{S+S'}{\vDash} (\mathcal{C}\ \mathcal{C}') :: \Omega''}\ \mathsf{compose}$$

$$\frac{\Sigma\ ;\ \Omega_1 \overset{p}{\Vdash} P :: (x : A)}{\Sigma\ ;\ \Omega\ \Omega_1 \overset{p+w}{\vDash} (\mathsf{proc}(x, w, P)) :: (\Omega\ (x : A))}\ C_{\mathsf{proc}}$$

$$\frac{\Sigma\ ;\ \Omega_1 \overset{p}{\underset{\mathsf{cf}}{\Vdash}} P :: (x : A)}{\Sigma\ ;\ \Omega\ \Omega_1 \overset{p+w}{\vDash} (\mathsf{msg}(x, w, P)) :: (\Omega\ (x : A))}\ C_{\mathsf{msg}}$$

**Fig. 6.** Typing rules for a configuration

## 6 Soundness

This section concludes the discussion of Resource-Aware SILL by proving the soundness of the resource-aware type system with respect to the cost semantics. So far, we have analyzed and type-checked processes in isolation. However, as our cost semantics indicates, processes always exist in a configuration, where they interact with other processes. Hence, we need to extend the typing rules to configurations.

*Configuration Typing* At runtime, a program state in Resource-Aware SILL is a set of processes interacting via messages. Such a set is represented as as a multi-set of proc and msg predicates as described in Section 4. To type the resulting configuration $\mathcal{C}$, we first need to define a well-formed signature.

A signature $\Sigma$ is said to be *well formed* if every process definition $x : S \leftarrow \mathcal{X}\ @\ p \leftarrow \overline{y : W} = P_{x,\overline{y}}$ in $\Sigma$ is well typed according to the process typing judgment, i.e. $\Sigma\ ;\ \Omega \overset{p}{\Vdash} P_{x,\overline{y}} :: (x : S)$.

We use the following judgment to type a configuration.

$$\Sigma; \Omega_1 \overset{S}{\vDash} \mathcal{C} :: \Omega_2$$

It states that $\Sigma$ is well-formed and that the configuration $\mathcal{C}$ uses the channels in the context $\Omega_1$ and provides the channels in the context $\Omega_2$. The natural number $S$ denotes the sum of the total potential and work done by the system. We call $S$ the weight of the configuration.

The configuration typing judgment is defined using the rules presented in Figure 6. The rule emp defines that an empty configuration is well-typed with weight 0. The rule compose composes two configurations $\mathcal{C}$ and $\mathcal{C}'$: $\mathcal{C}$ provides service on the channels in $\Omega'$ while $\mathcal{C}'$ uses the channels in $\Omega'$. The weight of the composed configuration $\mathcal{C}\ \mathcal{C}'$ is obtained by summing up their individual weights. The rule $C_{\mathsf{proc}}$ creates a configuration out of a single process. The weight of this singleton configuration is obtained by adding the potential of the process and the work performed by it. Similarly, the rule $C_{\mathsf{msg}}$ creates a configuration out of a single message. Since we account for the cost while typing the processes (see Figure 5), using the same judgment to type the processes would lead to

paying for the same message twice. Hence, the messages are typed in a cost-free judgment where $M^{\mathsf{label}} = M^{\mathsf{channel}} = M^{\mathsf{close}} = 0$.

*Soundness* Theorem 1 is the main theorem of the paper. It is a stronger version of a classical type preservation theorem and the usual type preservation is a direct consequence. Intuitively, it states that the weight of a configuration never increases during an evaluation step. This also implies that the weight of the initial configuration is an upper bound on the weight of any configuration it can ever step to. The soundness connects the potential with the work (i.e. the type system with the cost semantics).

**Theorem 1 (Soundness).** *Consider a well-typed configuration $\mathcal{C}$ w.r.t. a well-formed signature $\Sigma$ such that $\Sigma; \Omega_1 \overset{S}{\vDash} \mathcal{C} :: \Omega_2$. If $\mathcal{C} \mapsto \mathcal{C}'$, then there exist $\Omega_1'$, $\Omega_2'$ and $S'$ such that $\Sigma; \Omega_1' \overset{S'}{\vDash} \mathcal{C}' :: \Omega_2'$ and $S' \leq S$.*

The proof of the soundness theorem is achieved by a case analysis on the cost semantics, followed by an inversion on the typing of a configuration. The complete proof is presented in Appendix A. The preservation theorem is a corollary of the soundness, since we prove that the configuration $\mathcal{C}'$ is well-typed.

The soundness implies that the weight of a configuration is an upper bound on the total work performed by an evaluation starting in that configuration. We are particularly interested in the special case of a configuration that starts with 0 work. In this case, the weight corresponds to the initial potential of the system.

**Corollary 1 (Upper Bound).** *If $\Sigma; \Omega_1 \overset{S}{\vDash} \mathcal{C} :: \Omega_2$, and $\mathcal{C} \mapsto^* \mathcal{C}'$, then $S \geq W'$, where $W'$ is the total work performed by the configuration $\mathcal{C}'$, i.e. the sum of the work performed by each process and message in $\mathcal{C}'$. In particular, if the work done by the initial configuration $\mathcal{C}$ is 0, then the potential $P$ of the initial configuration satisfies $P \geq W'$.*

*Proof.* Applying the Soundness theorem successively, we get that if $\mathcal{C} \mapsto^* \mathcal{C}'$ and $\Sigma; \Omega_1 \overset{S}{\vDash} \mathcal{C} :: \Omega_2$ and $\Sigma; \Omega_1' \overset{S'}{\vDash} \mathcal{C}' :: \Omega_2'$, then $S' \leq S$. Also, $S' = P' + W'$, where $P'$ is the total potential of $\mathcal{C}'$, while $W'$ is the total work performed so far in $\mathcal{C}'$. Since $P' \geq 0$, we get that $W' \leq P' + W' = S' \leq S$. In particular, if $W = 0$, we get that $P = P + W = S \geq W'$, where $P$ and $W$ are the potential and work of the initial configuration respectively.

The progress theorem of Resource-Aware SILL is a direct consequence of progress in SILL [44]. Our cost semantics are a cost observing semantics, i.e. it is just annotated with counters observing the work. Hence, any runtime step that can be taken by a program in SILL can be taken in Resource-Aware SILL.

## 7 Case Study: Stacks and Queues

As an illustration of our type system, we present a case study on stacks and queues. Stacks and queues have the same interaction protocol: they store elements

of a variable type $A$ and support inserting and deleting elements. They only differ in their implementation and resource usage. We express their common interface type as the simple session type $\mathsf{store}_A$.

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

The session type dictates that a process providing a service of type $\mathsf{store}_A$, gives a client the choice to either insert (ins) or delete (del) an element of type A. Upon receipt of the label ins, the providing process expects to receive a channel of type $A$ to be enqueued and recurses. Upon receipt of the label del, the providing process either indicates that the queue is empty (none), in which case it terminates, or that there is a channel stored in the queue (some), in which case it deletes this channel, sends it to the client, and recurses.

To account for the resource cost, we need to add potential annotations leading to two different resource-aware types for stacks and queues. Since we are interested in counting the total number of messages exchanged, we set $M^{\mathsf{label}} = M^{\mathsf{channel}} = M^{\mathsf{close}} = 1$ in our type system to obtain a concrete bound.

*Stacks* The type for stacks is defined below.

$$\begin{aligned}\mathsf{stack}_A = \&\{\ &\mathsf{ins}^0 : A \overset{0}{\multimap} \mathsf{stack}_A,\\ &\mathsf{del}^2 : \oplus\{\mathsf{none}^0 : \mathbf{1}^0, \mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{stack}_A\}\}\end{aligned}$$

A stack is implemented as a sequence of *elem* processes terminated by an *empty* process. The implementation and type derivation of *elem* is presented below.

```
1: (x:A) (t:stack_A) |-^0 elem :: (s : stack_A)
2:   s <- elem <- x t =
3:     case s
4:       (ins => y <- recv s ;          %  (y:A) (x:A) (t:stack_A) |-^0 s : stack_A
5:               s' <- elem <- x t ;   %  (y:A) (s' : stack_A) |-^0 s : stack_A
6:               s <- elem <- y s'
7:       | del => s.some ;              %  (x:A) (t:stack_A) |-^1 s : A (x)^0 stack_A
8:               send s x ;             %  t:stack_A |-^0 s : stack_A
9:               s <- t)
```

The recursive *elem* process stores an element of the stack. It uses channel $x : A$ (element being stored) and channel $t : \mathsf{stack}_A$ (tail of the stack) and provides service along $s : \mathsf{stack}_A$. The implementation demonstrates that if the *elem* process receives an ins message along $s$, it receives the element $y$ (line 4), spawns a new *elem* process using its original element $x$ (line 5), and continues with another instance of the *elem* process with the received element $y$ (line 6). In this way, it adds the element $y$ to the head of the sequence. Otherwise, *elem* receives a del message along $s$ and responds with the some label (line 7), followed by the channel $x$ it stores (line 8). It then forwards all communication along $s$ to $t$.

Inserting an element has no resource cost, since no messages are sent by the *elem* process. Similarly, deleting an element has a cost of 2, which is used to send two messages: the $\mathsf{some}$ label and the element $x$. This is reflected by the type $\mathsf{stack}_A$, which needs 0 and 2 potential units for insertion and deletion, respectively, as indicated by the resource annotations.

We now implement and type the *empty* process.

```
10: · |-^0 empty :: (s : stack_A)
11:   s <- empty =
12:       case s (ins => y <- recv s ;          %  (y:A) |-^0 s : stack_A
13:                   e <- empty ;              %  (y:A) (e : stack_A) |-^0 s : stack_A
14:                   s <- elem <- y e
15:               | del => s.none ;             %  · |-^1 s : 1
16:                   close s)
```

The sequence of *elem* processes ends with an *empty* process, providing service along channel $s$ where it can receive the label $\mathsf{ins}$ or $\mathsf{del}$. If it receives the label $\mathsf{ins}$, it receives the element $y$ to be inserted (line 12), spawns a new *empty* process (line 13), and continues execution as an *elem* process with the received element (line 14). On receiving the label $\mathsf{del}$, it just sends the $\mathsf{none}$ label (line 15) followed by the $\mathsf{close}$ message (line 16), indicating that the stack is empty.

Inserting an element sends no messages and thus has cost 0. Deleting an element sends two messages and has cost 2, which is reflected in the resource annotations of the labels in the type $\mathsf{store}_A$. Note that deleting an element requires the system to send back two messages, either the $\mathsf{none}$ label followed by the close message, or the $\mathsf{some}$ label followed by the element. Therefore, an implementation of stacks will have a resource cost of at least 2 for deletion. This shows that the above implementation is the most efficient w.r.t. our cost semantics because insertion has no resource cost, and deletion has the least possible cost.

*Queues* Next, we consider the queue interface which is achieved by using the same $\mathsf{store}_A$ interface and annotating it with a different potential. The tight potential bound depends on the number of elements stored in the queue. Hence, a precise resource-aware type needs access to this internal measure in the type. A type $\mathsf{queue}_A[n]$ intuitively defines a queue of size $n$, i.e. a process offering along a channel of type $\mathsf{queue}_A[n]$ connects a sequence of $n$ elements.

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

Similar to a stack, a queue is also implemented by a sequence of *elem* processes, connected via channels, and terminated by the *empty* process. We show the implementation of *elem* below.

```
1:  (x:A) (t:queue_A[n - 1]) |-^0 elem :: (s : queue_A[n])
2:   s <- elem <- x t =
```

```
3:    case s (
4:     ins ⇒  y ← recv s ;   % (y:A) (x:A) (t:queue_A[n − 1]) |=^{2n} s:queue_A[n + 1]
5:            t.ins ;        % (y:A)(x:A)(t:A −^0∘ queue_A[n]) |-^1 s:queue_A[n + 1]
6:            send t y ;     %  (x : A) (t : queue_A[n]) |-^0 s : queue_A[n + 1]
7:            s ← elem ← x t
8:     | del ⇒ s.some ;       %  (x:A)(t:queue_A[n − 1]) |-^1 s:A ⊗^0 queue_A[n − 1]
9:            send s x ;     %  t:queue_A[n − 1] |-^0 s : queue_A[n − 1]
10:           s ← t)
```

Similar to the implementation of a stack, the *elem* process provides along $s : \mathsf{queue}_A$, stores the element $x : A$, and uses the tail of the queue $t : \mathsf{queue}_A$. When the *elem* process receives the ins message along $s$, it receives the element $y$ (line 4), and passes the ins message (line 5) along with $y$ (line 5) to $t$. Since the process at the other end of $t$ is also implemented using *elem*, it passes along the element to its tail too. Thus the element travels to the end of the queue where it is finally inserted. The deletion is similar to that for stack.

For each insertion, the ins label along with the element travels to the end of the queue. Hence, the resource cost of each insertion is $2n$ where $n$ is the size of the queue and this is reflected in the type $\mathsf{queue}_A$ in the potential annotation of ins as $2n$. Similar to the stack, deletion has a resource cost of 2 to get back the some label and the element. We now consider the *empty* process.

```
· |-^0 empty :: (s : queue_A[0])
  s ← empty =
    case s (
    ins ⇒  y ← recv s ;          %  (y:A)  |-^0 s : queue_A[1]
           e ← empty ;           %  (y:A) (e : queue_A[0]) |-^0 s : queue_A[1]
           s ← elem ← y e
    | del ⇒ s.none ;             %  · |-^1 s : 1
           close s)
```

The implementation of *empty* process is identical to that of stacks. Since insertion does not cause the process to send any messages, its resource cost is 0. On the other hand, deletion costs 2 units because the process sends back the none label followed by the close message. This is correctly reflected in the queue type. Since $s : \mathsf{queue}_A[0]$, the annotation for ins is $2n = 2 \cdot 0 = 0$. Similarly, del is annotated with a potential of 2.

The resource-aware types show that the implementation for stacks is more efficient than that of queues. This follows from the potential annotation. The label ins is annotated by $2n$ for $\mathsf{stack}_A$ and with 0 for $\mathsf{queue}_A$. The label del has the same annotation in both types.

*Functional queues* In a functional language, a queue is often implemented with two lists. The idea is to enqueue into the first list and to dequeue from the second list. If the second list is empty then we copy the first list over, thereby reversing

its order. Since the cost of the dequeue operation varies drastically between the dequeue operations, amortized analysis is again instrumental in the analysis of the worst-case behavior and shows that the worst-case amortized cost for deletion is actually a constant.

Appendix B.3 contains an implementation of a functional queue in Resource-Aware SILL. The type of the queue is

$$\begin{aligned}\mathsf{queue}'_A = \&\{\ \mathsf{ins}^6 : A \overset{0}{\multimap} \mathsf{queue}'_A,\\ \mathsf{del}^2 : \oplus\{\ \mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{queue}'_A, \mathsf{none}^0 : \mathbf{1}^0\}\}\end{aligned}$$

Resource-aware session types enable us to translate the amortized analysis to the distributed setting. The type prescribes that an insertion has an amortized cost of 6 while the deletion has an amortized cost of 2. The main idea here is that the elements are inserted with a constant potential in the first list. While deleting, if the second list is empty, then this stored potential in the first list is used to pay for copying the elements over to the second list. The exact potential annotations for the two lists can be found in Appendix B.3. As demonstrated from the resource-aware type, this implementation is more efficient than the previous queue implementation, which has a linear resource cost for insertion.

*Generic clients* The notion of efficiency of a store can be generalized and quantified by considering clients for the stack and queue interface. A client interacts with a generic store via a sequence of insertions and deletions. A provider can then implement the store as a stack, queue, priority queue, etc. (same interface) and just expose the resource-aware type for $\mathsf{store}_A$. Our type system can then use just the interface type and the generic client implementation to derive resource bounds on the client. For simplicity, the clients are typed in an affine type system which allows us to throw away dummy channels (see below).

We provide a general mechanism for implementing clients for a generic store. We define a generic $\mathsf{store}_A$ type at which the potential annotations are arbitrary natural numbers.

$$\begin{aligned}\mathsf{store}_A[n] = \&\{\ \mathsf{ins}^i : A \overset{a}{\multimap} \mathsf{store}_A[n+1],\\ \mathsf{del}^d : \oplus\{\mathsf{none}^p : \mathbf{1}^e, \mathsf{some}^s : A \overset{t}{\otimes} \mathsf{store}_A[n-1]\}\}\end{aligned}$$

A client is defined by a list $\ell$ of $\mathsf{ins}$ and $\mathsf{del}$ messages that it sends to the store. We index the client $C_{\ell,n}$ using $\ell$, and the internal measure $n$ of the $\mathsf{store}_A$ type. The channel along which the client provides is irrelevant for our analysis and is represented using a dummy channel $d : D$. For ease of notation, we define the potential needed for a client $C_{\ell,n}$ as a function $\phi(\ell, n)$.

We implement the client $C_{\ell,n}$ as follows. First, consider the case when $\ell = []$, i.e. an empty list.

$$\begin{aligned}\cdot \overset{0}{\vdash} C_{[],n} :: (d : D)\\ d \leftarrow C_{[],n} = \mathsf{close}\ d\end{aligned}$$

The client for an empty list just closes the channel $d$. We assume that all clients are typed with the cost-free metric to only count for the messages sent inside the stores. So $C_{[],0}$ needs 0 potential. For the potential function, this means $\phi([], n) = 0$.

Next, we implement the client when the head of the list $\ell$ is $\mathsf{ins}$.

$$
\begin{array}{l}
\Omega\ (x : A)\ (s : \mathsf{store}_A[n]) \vDash^{q} C_{\mathsf{ins}::\ell,n} :: (d : D) \\
\quad d \leftarrow C_{\mathsf{ins}::\ell,n} \leftarrow \Omega\ x\ s = \\
\qquad s.\mathsf{ins}\ ; \qquad\qquad \% \quad \Omega\ (x : A)\ (s : A \overset{a}{\multimap} \mathsf{store}_A[n+1]) \vDash^{q-i} d : D \\
\qquad \mathsf{send}\ s\ x\ ; \qquad \% \quad \Omega\ (s : \mathsf{store}_A[n+1]) \vDash^{q-i-a} d : D \\
\qquad d \leftarrow C_{\ell,n+1} \leftarrow \Omega\ s
\end{array}
$$

The client sends an $\mathsf{ins}$ label followed by the element $x$. If $C_{\mathsf{ins}::\ell,n}$ needs a potential $q$, then the type derivation informs us that $C_{\ell,n+1}$ needs a potential $q - i - a$. Thus, $\phi(\mathsf{ins} :: \ell, n) = \phi(\ell, n+1)+i+a$. Finally, we show the client implementation if the head of the list $\ell$ is $\mathsf{del}$.

$$
\begin{array}{l}
\Omega\ (s : \mathsf{store}_A[n]) \vDash^{q} C_{\mathsf{del}::\ell} :: (d : D) \\
\quad d \leftarrow C_{\mathsf{del}::\ell,n} \leftarrow \Omega\ s = \\
\qquad s.\mathsf{del}\ ; \qquad \% \quad \Omega\ (s : \oplus\{\mathsf{none}^n : \mathbf{1}^e, \mathsf{some}^s : A \overset{t}{\otimes} \mathsf{store}_A[n-1]\}) \vDash^{q-d} d : D \\
\qquad \mathsf{case}\ s\ ( \\
\qquad \mathsf{some} \Rightarrow \quad x \leftarrow \mathsf{recv}\ s\ ; \qquad \% \quad \Omega\ (x : A)\ (s : \mathsf{store}_A[n-1]) \vDash^{q-d+s+t} d : D \\
\qquad\qquad\quad\ d \leftarrow C_{\ell,n-1} \leftarrow \Omega\ x\ s \\
\qquad \mid \mathsf{none} \Rightarrow \ \mathsf{wait}\ s) \qquad\qquad \% \quad \Omega \vDash^{q-d+p+e} d : D
\end{array}
$$

The client sends the $\mathsf{del}$ label and then case analyzes on the label it receives. If it receives the $\mathsf{some}$ label, it receives the element and then continues with $C_{\ell,n-1}$, else it receives the $\mathsf{none}$ label and waits for the channel $s$ to close. In terms of the potential function, this means

$$
\phi(\mathsf{del} :: \ell, n) = \begin{cases} \phi(\ell, n-1) + d - s - t & \text{if} \quad n > 0 \\ \max(0, d - p - e) & \text{otherwise} \end{cases}
$$

Walking through the list $\ell$ and chaining the potential equations together, we can achieve a resource bound on the client $C_{\ell,n}$ by computing $\phi(\ell, n)$.

The $\mathsf{stack}_A$ and $\mathsf{queue}_A$ interface types are specific instantiations of the $\mathsf{store}_A$ type. For the stack interface, plugging in appropriate potential annotations $i = a = p = e = s = t = 0$, and $d = 2$, we get (ignoring the case where the stack becomes empty)

$$
\phi([], n) = 0 \qquad \phi(\mathsf{ins} :: \ell, n) = \phi(\ell, n+1) \qquad \phi(\mathsf{del} :: \ell, n) = \phi(\ell, n-1) + 2
$$

Similarly, considering the queue type as another instantiation of the $\mathsf{store}_A$ type, and plugging $a = p = e = s = t = 0$, $i = 2n$ and $d = 2$, we get

$$
\phi([], n) = 0 \qquad \phi(\mathsf{ins} :: \ell, n) = \phi(\ell, n+1) + 2n \qquad \phi(\mathsf{del} :: \ell, n) = \phi(\ell, n-1) + 2
$$

Finally, looking at $\mathsf{queue}'_A$ as another instantiation of the $\mathsf{store}_A$ type and plugging $a = p = e = s = t = 0$, $i = 6$ and $d = 2$, we get

$$\phi([], n) = 0 \qquad \phi(\mathsf{ins} :: \ell, n) = \phi(\ell, n+1) + 6 \qquad \phi(\mathsf{del} :: \ell, n) = \phi(\ell, n-1) + 2$$

This allows us to compare arbitrary clients of two (same or different) interfaces and compare their resource cost. The resource-aware types are expressive enough to obtain these resource bounds without referring the implementation of the store interface. For instance, an important property of queues is that every insertion is more costly than the previous one. The cost of insertion depends on the size of the queue, which, in turn, increases with every insertion. Hence, the complexity of the queue system depends on the sequence in which inserts and deletes are performed. In particular, we can consider the efficiency of two different clients for the queue system, by solving the above system of equations.

For instance, consider two clients $Q_{\ell_1,n}$ and $Q_{\ell_2,n}$, with two different message lists $\ell_1 = [\mathsf{ins}, \ldots, \mathsf{ins}, \mathsf{del}, \ldots, \mathsf{del}]$, i.e. $m$ insertions followed by $m$ deletions, and $\ell_2 = [\mathsf{ins}, \mathsf{del}, \mathsf{ins}, \mathsf{del}, \ldots, \mathsf{ins}, \mathsf{del}]$, i.e. $m$ instances of alternate insertions and deletions. In both cases, we have the same number of insertions and deletions. However, the resource cost of the two systems are completely different. Solving the system of equations, we get that $\phi(\ell_1, n) = 2mn + m(m-1) + 2m$, while $\phi(\ell_2, n) = 2m(n+1)$, which shows that the second client is an order of magnitude more efficient than the first one. More examples are presented in Appendix B.

## 8 Related Work

Session types have been introduced by Honda [32,33]. The technical development in this article is based on previous work on [44,40]. By removing the potential annotation from the type rules in Section 5 we arrive at the type system of loc. cit. The internal measures and type families that we use are inspired by [24]. Other recent innovations in session types include sharing of resources [9] and dynamic monitors [34]. In contrast to our work, all the aforementioned articles do not discuss static resource analysis.

Static resource bound analysis for sequential programs has been extensively studied. Successful approaches are based on refinement types [18,15], linear dependent types [37], abstract interpretation [46,25,5,16], deriving and solving recurrence relations [19,20,4,36], term rewriting [11,8]. These works do not consider message-passing programs nor concurrent or parallel evaluation.

Our work is based on type-based amortized resource analysis. Automatic amortized resource analysis (AARA) has been introduced as a type system to automatically derive linear [30] and polynomial bounds [27] for sequential functional programs. It can also be integrated with program logics to derive bounds for imperative programs [7,14]. Moreover, it has been used to derive bounds for term-rewrite systems [31] and object-oriented programs [29]. A recent work also considers bounds on the parallel evaluation cost (also called *span*) of functional programs [28]. The innovation of our work is the integration of AARA and session types and the analysis of message-passing programs that communicate

with the outside world. Instead of function arguments, our bounds depend on the messages that are send along channels. As a result, the formulation and proof of the soundness theorem is quite different from the soundness of sequential AARA.

We are only aware of a couple of other works that study resource bounds for concurrent programs. Gimenez et al. [22] introduced a technique for analyzing the parallel and sequential space and time cost of evaluating interaction nets. While it also based on linear logic and potential annotations, the flavor of the analysis is quite different. Interaction nets are mainly used to model parallel evaluation while session types focus on the interaction of processes. A main innovation of our work is that processes can exchange potential via messages. It is not clear how we can represent the examples we consider in this article as interaction nets. Albert et al. [3,2] have studied techniques for deriving bounds on the cost of concurrent programs that are based on the actor model. While the goals of the work are similar to ours, the used technique and considered examples are dissimilar. A major difference is that our method is type-based and compositional. A unique feature of our work is that types describe bounds as functions of the messages that are sent along a channel.

## 9 Conclusion

We have introduced resource-aware session types, a linear type system that combines session types [32,40] and type-based amortized resource analysis [30,27] to reason about the resource usage of message-passing processes. The soundness of the type system has been proved for a core session-typed language with respect to a cost semantics that tracks the total communication cost in a system of processes. We have demonstrated that our technique can be used to prove tight resource bounds and supports amortized reasoning by analyzing standard session-type data structures such as distributed binary counters, stacks, and queues.

Our approach addresses some of the main challenges of analyzing message-passing programs such as compositionality and description of symbolic bounds. However, there are several open problems that we plan to tackle as part of future work. The technique we have developed in this paper does not yet account for the concurrent execution cost of processes, or the *span*. We are working on a companion paper that describes a type-based analysis to derive bounds on the span; the earliest time a concurrent computation terminates assuming an infinite number of processors. Due to data dependencies in a concurrent program, a process needs to wait for messages from other processes, and computing these waiting times statically makes span analysis challenging.

Similarly, we have focused on the foundations and meta theory of resource-aware session types in this paper. The next step is to implement our analysis. An advantage of our method is that it is based upon type-based amortized resource analysis for sequential programs. We will integrate the type system with SILL for functional programs [27]. We designed the type system with automation in mind and we are confident that we can support automatic type inference using templates and LP solving similar to AARA [30,27]. To this end, we are working on an algorithmic version of the declarative type system presented here.

## References


1. Acar, U.A., Charguéraud, A., Rainey, M.: Oracle-guided scheduling for controlling granularity in implicitly parallel languages. J. Funct. Program. 26 (2016)
2. Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: Resource analysis: From sequential to concurrent and distributed programs. In: Formal Methods - 20th International Symposium (FM'15) (2015)
3. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: May-Happen-in-Parallel Analysis for Actor-Based Concurrency. ACM Trans. Comput. Log. 17(2) (2016)
4. Albert, E., Genaim, S., Masud, A.N.: On the Inference of Resource Usage Upper and Lower Bounds. ACM Transactions on Computational Logic 14(3) (2013)
5. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In: (SAS'10) (2010)
6. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition Instead of Self-composition for Proving the Absence of Timing Channels. In: (PLDI'17)
7. Atkey, R.: Amortised Resource Analysis with Separation Logic. In: (ESOP'10)
8. Avanzini, M., Lago, U.D., Moser, G.: Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In: 29th Int. Conf. on Func. Prog. (ICFP'15)
9. Balzer, S., Pfenning, F.: Manifest sharing with session types (ICFP) (Aug 2017)
10. Brent, R.P.: Algorithms for minimization without derivatives. Courier Corporation (2013)
11. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating Runtime and Size Complexity Analysis of Integer Programs. In: (TACAS'14) (2014)
12. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Proc. of the 21st Int. Conf. on Concurrency Theory (CONCUR 2010). pp. 222–236 (Aug 2010)
13. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science 26(3), 367–423 (2016)
14. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated Resource Analysis with Coq Proof Objects. In: 29th Int. Conf. on Computer-Aided Verification (CAV'17) (2017)
15. Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational Cost Analysis. In: 44th Symposium on Principles of Programming Languages (POPL'17) (2017)
16. Cerný, P., Henzinger, T.A., Kovács, L., Radhakrishna, A., Zwirchmayr, J.: Segment Abstraction for Worst-Case Execution Time Analysis. In: (ESOP'15) (2015)
17. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Electron. Notes Theor. Comput. Sci. 165, 145–176 (Nov 2006)
18. Çiçek, E., Garg, D., Acar, U.A.: Refinement Types for Incremental Computational Complexity. In: 24th European Symposium on Programming (ESOP'15) (2015)
19. Danner, N., Licata, D.R., Ramyaa, R.: Denotational Cost Semantics for Functional Languages with Inductive Types. In: 29th Int. Conf. on Functional Programming (ICFP'15)
20. Flores-Montoya, A., Hähnle, R.: Resource Analysis of Complex Programs with Cost Equations. In: Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)
21. Gay, S.J., Hole, M.: Subtyping for session types in the $\pi$-calculus. Acta Informatica (2005)
22. Gimenez, S., Moser, G.: The complexity of interaction. In: (POPL'16) (2016)
23. Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1 – 101 (1987)
24. Griffith, D., Gunter, E.L.: Liquid pi: Inferrable dependent session types. In: Proceedings of the NASA Formal Methods Symposium. pp. 186–197. Springer LNCS 7871 (2013)
25. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th Principles of Prog. Langs. (POPL'09)

26. Haemmerlé, R., López-García, P., Liqat, U., Klemen, M., Gallagher, J.P., Hermenegildo, M.V.: A Transformational Approach to Parametric Accumulated-Cost Static Profiling. In: Functional and Logic Programming - 13th International Symposium (FLOPS'16) (2016)
27. Hoffmann, J., Das, A., Weng, S.C.: Towards Automatic Resource Bound Analysis for OCaml. In: 44th Symposium on Principles of Programming Languages (POPL'17) (2017)
28. Hoffmann, J., Shao, Z.: Automatic Static Cost Analysis for Parallel Programs. In: 24th European Symposium on Programming (ESOP'15) (2015)
29. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: (ESOP'06)
30. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03) (2003)
31. Hofmann, M., Moser, G.: Multivariate Amortised Resource Analysis for Term Rewrite Systems. In: 13th Int. Conf. on Typed Lambda Calculi and Appl. (TLCA'15) (2015)
32. Honda, K.: Types for dyadic interaction, pp. 509–523. Springer Berlin Heidelberg (1993)
33. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. ESOP'98, Springer LNCS 1381
34. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and blame assignment for higher-order session types. In: Proc. of the 43rd ACM Symp. on Prin. of Prog. Lang. POPL '16
35. Jia Chen, Y.F., Dillig, I.: Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In: Conf. on Comp. and Comm. Security (CCS'17)
36. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.: Compositional recurrence analysis revisited. In: Conference on Programming Language Design and Implementation (PLDI'17) (2017)
37. Lago, U.D., Petit, B.: The Geometry of Types. In: (POPL'13) (2013)
38. Ngo, V.C., Dehesa-Azuara, M., Fredrikson, M., Hoffmann, J.: Verifying and Synthesizing Constant-Resource Implementations with Types. In: 38th IEEE Symp. on S&P (2017)
39. Olivo, O., Dillig, I., Lin, C.: Static Detection of Asymptotic Performance Bugs in Collection Traversals. In: Conf. on Prog. Lang. Design and Impl. (PLDI'15) (2015)
40. Pfenning, F., Griffith, D.: Polarized Substructural Session Types. Springer Berlin (2015)
41. Pfenning, F., Simmons, R.J.: Substructural operational semantics as ordered logic programming. In: Proc. of 24th IEEE Symp. on Logic In Comp. Sc. LICS '09 (2009)
42. Silva, M., Florido, M., Pfenning, F.: Non-blocking concurrent imperative programming with session types. In: Proc. 4th Int. Workshop on Linearity, LINEARITY 2016. (2016)
43. Tarjan, R.E.: Amortized Computational Complexity. SIAM J. Alg. Disc. Methods (1985)
44. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: (ESOP'13). Springer LNCS 7792, Rome, Italy (Mar 2013)
45. Wadler, P.: Propositions as sessions. In: (ICFP'12). ACM Press, Copenhagen, Denmark
46. Zuleger, F., Sinn, M., Gulwani, S., Veith, H.: Bound Analysis of Imperative Programs with the Size-change Abstraction. In: 18th Int. Static Analysis Symp. (SAS'11) (2011)

## A Proof of Soundness Theorem

We present the complete proof of the soundness theorem. We reiterate the rules of cost semantics and typing. Figure 4 presents the cost semantics for our session-typed language, while Figure 5 presents the full set of typing rules. Finally, Theorem 1 defines the soundness theorem establishing that the typing rules for a configuration presented in Figure 6 are sound w.r.t. the rules for cost semantics presented in Figure 4. We follow the complete proof of the soundness theorem.

*Proof.* The proof proceeds by case analysis on the cost semantics of our language, i.e. on the judgment $\mathcal{C} \mapsto \mathcal{C}'$. By the compose rule, we can split the configuration such that $\mathcal{C} = (\mathcal{C}_M\ \mathcal{D})$ and $\mathcal{C}' = (\mathcal{C}'_M\ \mathcal{D})$ and $\mathcal{C}_M \mapsto \mathcal{C}'_M$. Using the compose rule,

$$\frac{\Sigma\ ;\ \Omega \overset{S_M}{\vDash} \mathcal{C}_M :: \Omega' \quad \Sigma\ ;\ \Omega' \overset{S_D}{\vDash} \mathcal{D} :: \Omega''}{\Sigma\ ;\ \Omega \overset{S_M+S_D}{\vDash} (\mathcal{C}_M\ \mathcal{D}) :: \Omega''}\ \mathsf{compose}$$

$$\frac{\Sigma\ ;\ \Omega \overset{S_M}{\vDash} \mathcal{C}'_M :: \Omega' \qquad \Sigma\ ;\ \Omega' \overset{S_D}{\vDash} \mathcal{D} :: \Omega''}{\Sigma\ ;\ \Omega \overset{S'_M+S_D}{\vDash} (\mathcal{C}'_M\ \mathcal{D}) :: \Omega''}\ \mathsf{compose}'$$

Hence, to show that $S' \leq S$, it suffices to show that $S'_M \leq S_M$. We proceed by case analysis on the $\mathcal{C}_M \mapsto \mathcal{C}'_M$ judgment.

- Case $(\mathsf{spawn}_c) : \mathcal{C}_M = \mathsf{proc}(d, w, x \leftarrow P_x \leftarrow \overline{y}\ ;\ Q_x)$. Inverting the typing rule spawn on configuration $\mathcal{C}_M$, we get

$$r \geq p + q \qquad \Omega_1\ \Omega_2 \overset{r}{\vdash} x \leftarrow P_x \leftarrow \overline{y}\ ;\ Q_x :: (d : U) \tag{4}$$

  and in $\mathcal{C}'_M = \mathsf{proc}(c, 0, P_c)\ \mathsf{proc}(d, w, Q_c)$, we get (the premise due to inversion)

$$\Omega_1 \overset{p}{\vdash} P_c :: (c : S) \qquad \Omega_2\ (c : S) \overset{q}{\vdash} Q_c :: (d : U)$$

  From the cost semantics rule $\mathsf{spawn}_c$,

$$\frac{\mathsf{proc}(d, w, x \leftarrow P_x \leftarrow \overline{y}\ ;\ Q_x)}{\mathsf{proc}(c, 0, P_c) \qquad \mathsf{proc}(d, w, Q_c)}\ \mathsf{spawn}_c$$

  Since $S$ is the sum of work and potential of each process in the configuration, we get $S_M = r + w$, while $S'_M = (p_P + w_P) + (p_Q + w_Q) = (p + 0) + (q + w) = p + q + w \leq r + w = S_M$ since $p + q \leq r$ (by Equation 4).
- Case $(\mathsf{fwd}_s) : \mathcal{C}_M = \mathsf{proc}(c, w, c \leftarrow d)$. Inverting the typing rule fwd on $\mathcal{C}_M$,

$$q \geq 0 \qquad d : S \overset{q}{\vdash} c \leftarrow d :: (c : S)$$

  Inverting the same rule in $\mathcal{C}'_M$,

$$q' \geq 0 \qquad d : S \overset{q'}{\underset{\mathsf{cf}}{\vdash}} c \leftarrow d :: (c : S)$$

  Using the semantics rule $\mathsf{fwd}_s$,

$$\frac{\mathsf{proc}(c, w, c \leftarrow d)}{\mathsf{msg}(c, w, c \leftarrow d)}\ \mathsf{fwd}_s$$

  Since we are free to choose $q'$, we set $q' \leq q$. Computing $S_M$ and $S'_M$, we get

$$\begin{aligned} S'_M &= q' + w \\ &\leq q + w \\ &= S_M \end{aligned}$$

– Case $(\mathsf{fwd}_r^+) : \mathcal{C}_M = \mathsf{proc}(d, w, P)\ \mathsf{msg}(c, w', c \leftarrow d)$. Inverting the $\mathsf{fwd}$ rule for $\mathcal{C}_M$,

$$\begin{aligned} q_1 \geq 0 \qquad \Omega \Vdash^{q_1} P :: (d : S) \\ q_2 \geq 0 \qquad d : S \Vdash^{q_2} c \leftarrow d :: (c : S) \end{aligned} \tag{5}$$

Using Equation 5 and noting that $c$ and $d$ have the same type $S$, we get for $\mathcal{C}'_M$,

$$q'_1 \geq 0 \qquad \Omega \Vdash^{q'_1} [c/d]P :: (c : S)$$

From the cost semantics rule $\mathsf{fwd}_r^+$, we get

$$\frac{\mathsf{proc}(d, w, P) \qquad \mathsf{msg}(c, w', c \leftarrow d)}{\mathsf{proc}(c, w + w', [c/d]P)}\ \mathsf{fwd}_r^+$$

Since we are free to choose $q'_1$, we set $q'_1 \leq q_1$. Computing $S_M$ and $S'_M$, we get

$$\begin{aligned} S'_M &= q'_1 + w + w' \\ &\leq q_1 + q_2 + w + w' \\ &= (q_1 + w) + (q_2 + w') \\ &= S_M \end{aligned}$$

– Case $(\mathsf{fwd}_r^-) : \mathcal{C}_M = \mathsf{proc}(e, w, P)\ \mathsf{msg}(c, w', c \leftarrow d)$. Inverting the $\mathsf{fwd}$ rule for $\mathcal{C}_M$,

$$\begin{aligned} q_1 \geq 0 \qquad \Omega\ (c : S) \Vdash^{q_1} P :: (e : U) \\ q_2 \geq 0 \qquad d : S \Vdash^{q_2} c \leftarrow d :: (c : S) \end{aligned} \tag{6}$$

Using Equation 6 and noting that $c$ and $d$ have the same type $S$, we get for $\mathcal{C}'_M$,

$$q'_1 \geq 0 \qquad \Omega\ (d : S) \Vdash^{q'_1} [d/c]P :: (e : U)$$

From the cost semantics rule $\mathsf{fwd}_r^-$, we get

$$\frac{\mathsf{proc}(e, w, P) \qquad \mathsf{msg}(c, w', c \leftarrow d)}{\mathsf{proc}(e, w + w', [d/c]P)}\ \mathsf{fwd}_r^-$$

Since we are free to choose $q'_1$, we set $q'_1 \leq q_1$.

$$\begin{aligned} S'_M &= q'_1 + w + w' \\ &\leq q_1 + q_2 + w + w' \\ &= (q_1 + w) + (q_2 + w') \\ &= S_M \end{aligned}$$

– Case $(\oplus C_s) : \mathcal{C}_M = \mathsf{proc}(c, w, c.l_k\ ;\ P)$. Inverting the typing rule $\oplus R_k$ on $\mathcal{C}_M$, we get

$$q_1 \geq p + r_k + M^{\mathsf{label}} \qquad \Omega \Vdash^{q_1} (c.l_k\ ;\ P) :: (c : \oplus\{l_i^{r_i} : S_i\}_{i \in I}) \tag{7}$$

and in $\mathcal{C}'_M$, we get (the premise due to inversion)

$$\Omega \Vdash^{p} [c'/c]P :: (c' : S_k)$$

$$q'_1 \geq q_2 + r_k \qquad c' : S_k \Vdash^{q'_1} (c.l_k\ ;\ c \leftarrow c') :: (c : \oplus\{l_i^{r_i} : S_i\}_{i \in I}) \tag{8}$$

$$q_2 \geq 0 \qquad c' : S_k \Vdash^{q_2} c \leftarrow c' :: (c : S_k) \tag{9}$$

Using the cost semantics rule $\oplus C_s$, we get

$$\frac{\mathsf{proc}(c, w, c.l_k\ ;\ P)}{\mathsf{proc}(c', w + M^{\mathsf{label}}, [c'/c]P) \qquad \mathsf{msg}(c, 0, c.l_k\ ;\ c \leftarrow c')}\ \oplus C_s$$

Again, $S'_M = (p + w + M^{\mathsf{label}}) + (q'_1 + 0)$. Since, we need to prove that there exists such an $S'_M$, we can choose $q'_1$ and $q_2$ arbitrarily such that they satisfy Equations 8 and 9. We set $q_2 = 0$ and $q'_1 = r_k$. Hence, $S'_M = (p + w + M^{\mathsf{label}}) + r_k \leq q_1 + w \leq S_M$ (by Equation 7)

– Case $(\oplus C_r) : \mathcal{C}_M = \mathsf{msg}(c, w, c.l_k\ ;\ c \leftarrow c')\ \mathsf{proc}(d, w', \mathsf{case}\ c\ (l_i \Rightarrow Q_i)_{i \in I})$. Inverting the typing rule $\oplus L$ on $\mathcal{C}_M$, we get

$$q_1 \geq q_2 + r_k \qquad c' : S_k \Vdash^{q_1} (c.l_k\ ;\ c \leftarrow c') :: (c : \oplus\{l_i^{r_i} : S_i\}_{i \in I}) \tag{10}$$

$$q_2 \geq 0 \qquad c' : S_k \Vdash^{q_2} c \leftarrow c' :: (c : S_k) \tag{11}$$

$$q_3 + r_k \geq q_k \qquad \Omega\ (c : \oplus\{l_i^{r_i} : S_i\}_{i \in I}) \Vdash^{q_3} \mathsf{case}\ c\ (l_i \Rightarrow Q_i)_{i \in I} :: (d : U) \tag{12}$$

and in $\mathcal{C}'_M$ (premise of the typing rule due to inversion), we get

$$\Omega\ (c' : S_k) \Vdash^{q_k} [c'/c]Q_k :: (d : U)$$

Using the cost semantics rule $\oplus C_r$, we get

$$\frac{\mathsf{msg}(c, w, c.l_k\ ;\ c \leftarrow c') \qquad \mathsf{proc}(d, w', \mathsf{case}\ c\ (l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(d, w + w', [c'/c]Q_k)}\ \oplus C_r$$

Again, $S'_M = q_k + w + w' \leq q_3 + r_k + w + w' \leq q_3 + q_2 + r_k + w + w' \leq q_3 + q_1 + w + w' \leq (q_1 + w) + (q_3 + w') = S_M$ (by Equations 10, 11 and 12).

– Case $(\& C_s)$ : Analogous to $\multimap C_s$.
– Case $(\& C_r)$ : Analogous to $\multimap C_r$.
– Case $(\otimes C_s)$ : Analogous to $\oplus C_s$.
– Case $(\otimes C_r)$ : Analogous to $\oplus C_r$.
– Case $(\multimap C_s) : \mathcal{C}_M = \mathsf{proc}(d, w, \mathsf{send}\ c\ e\ ;\ P)$. Applying the rule $\multimap L$ on $\mathcal{C}_M$, we get

$$q_1 \geq p + r + M^{\mathsf{channel}} \qquad \Omega\ (e : S)\ (c : S \overset{r}{\multimap} T) \Vdash^{q_1} (\mathsf{send}\ c\ e\ ;\ P) :: (d : U)$$

Inverting the same rule on $\mathcal{C}'_M$, we get

$$\Omega\ (c' : T) \Vdash^{p} [c'/c]P :: (d : U)$$

$$q_1' \geq q_2 + r \qquad (e : S)\ (c : S \overset{r}{\multimap} T) \mid^{q_1'} \mathsf{send}\ c\ e\ ;\ c' \leftarrow c :: (c' : T)$$

$$q_2 \geq 0 \qquad c : T \mid^{q_2} c' \leftarrow c :: (c' : T)$$

From the cost semantics rule $\multimap C_s$, we get

$$\frac{\mathsf{proc}(d, w, \mathsf{send}\ c\ e\ ;\ P)}{\mathsf{proc}(d, w + M^{\mathsf{channel}}, [c'/c]P) \qquad \mathsf{msg}(c', 0, \mathsf{send}\ c\ e\ ;\ c' \leftarrow c)}\ \multimap C_s$$

Since we can choose arbitrary values for $q_1'$ and $q_2$ satisfying the above inequalities, we set $q_2 = 0$ and $q_1' = r$. Computing $S_M$ and $S_M'$, we get

$$\begin{aligned} S_M' &= (p + w + M^{\mathsf{channel}}) + (q_1' + w') \\ &= (p + r + M^{\mathsf{channel}} + w) + (q_1' - r + w') \\ &\leq q_1 + w + w') \\ &= S_M \end{aligned}$$

– Case $(\multimap C_r) : \mathcal{C}_M = \mathsf{msg}(c', w, \mathsf{send}\ c\ e\ ;\ c' \leftarrow c)\ \mathsf{proc}(c, w', x \leftarrow \mathsf{recv}\ c\ ;\ Q_x)$. Applying the rule $\multimap L$ on the message in $\mathcal{C}_M$, we get

$$q_1 \geq q_2 + r \qquad (e : S)\ (c : S \overset{r}{\multimap} T) \mid^{q_1} \mathsf{send}\ c\ e\ ;\ c' \leftarrow c :: (c' : T)$$

$$q_2 \geq 0 \qquad c : T \mid^{q_2} c' \leftarrow c :: (c' : T)$$

Applying the rule $\multimap R$ on the process in $\mathcal{C}_M$, we get

$$q_3 + r \geq p \qquad \Omega \mid^{q_3} (x \leftarrow \mathsf{recv}\ c\ ;\ Q_x) :: (x : S \overset{r}{\multimap} T)$$

Inverting the $\multimap R$ rule, we get for $\mathcal{C}_M'$,

$$\Omega\ (e : S) \mid^{p} [c'/c]Q_e :: (c' : T)$$

From the cost semantics rule $\multimap C_r$, we get

$$\frac{\mathsf{msg}(c', w, \mathsf{send}\ c\ e\ ;\ c' \leftarrow c) \qquad \mathsf{proc}(c, w', x \leftarrow \mathsf{recv}\ c\ ;\ Q_x)}{\mathsf{proc}(c, w + w', [c'/c]Q_e)}\ \multimap C_r$$

Computing $S_M$ and $S_M'$, we get

$$\begin{aligned} S_M' &= p + w + w' \\ &\leq q_3 + r + w + w' \\ &\leq q_3 + q_2 + r + w + w' \\ &\leq q_3 + q_1 + w + w' \\ &= (q_1 + w) + (q_3 + w') \\ &= S_M \end{aligned}$$

– Case $(\mathbf{1}C_s) : \mathcal{C}_M = \mathsf{proc}(c, w, \mathsf{close}\ c)$. Applying the $\mathbf{1}R$ rule on $\mathcal{C}_M$, we get

$$q \geq r + M^{\mathsf{close}} \qquad \cdot \Vdash^{q} \mathsf{close}\ c :: (c : \mathbf{1}^r)$$

Inverting the same rule for $\mathcal{C}'_M$, we get

$$q' \geq r \qquad \cdot \Vdash^{q'} \mathsf{close}\ c :: (c : \mathbf{1}^r)$$

From the cost semantics rule $\mathbf{1}C_s$, we get

$$\frac{\mathsf{proc}(c, w, \mathsf{close}\ c)}{\mathsf{msg}(c, w + M^{\mathsf{close}}, \mathsf{close}\ c)}\ \mathbf{1}C_s$$

Setting $q' = r$ and computing $S_M$ and $S'_M$, we get

$$\begin{aligned} S'_M &= q' + w + M^{\mathsf{close}} \\ &= r + w + M^{\mathsf{close}} \\ &\leq q + w \\ &= S_M \end{aligned}$$

– Case $(\mathbf{1}C_r) : \mathcal{C}_M = \mathsf{msg}(c, w, \mathsf{close}\ c)\ \mathsf{proc}(d, w', \mathsf{wait}\ c\ ;\ Q)$. Applying the rule $\mathbf{1}R$ on the message in $\mathcal{C}_M$, we get

$$q_1 \geq r \qquad \cdot \Vdash^{q_1} \mathsf{close}\ c :: (c : \mathbf{1}^r)$$

Applying the $\mathbf{1}L$ rule on the process in $\mathcal{C}_M$, we get

$$q_2 + r \geq p \qquad \Omega\ (c : \mathbf{1}^r) \Vdash^{q_2} \mathsf{wait}\ c\ ;\ Q :: (d : U)$$

Inverting the $\mathbf{1}L$ rule for $\mathcal{C}'_M$, we get

$$\Omega \Vdash^{p} Q :: (d : U)$$

From the cost semantics rule $\mathbf{1}C_r$, we get

$$\frac{\mathsf{msg}(c, w, \mathsf{close}\ c) \qquad \mathsf{proc}(d, w', \mathsf{wait}\ c\ ;\ Q)}{\mathsf{proc}(d, w + w', Q)}\ \mathbf{1}C_r$$

Computing $S_M$ and $S'_M$, we get

$$\begin{aligned} S'_M &= p + w + w' + M^{\mathsf{close}} \\ &\leq q_2 + r + w + w' \\ &\leq q_2 + q_1 + w + w' \\ &\leq (q_1 + w) + (q_2 + w') \\ &= S_M \end{aligned}$$

Hence, in all of the above cases, $S'_M \leq S_M$ establishing that $S' \leq S$, thus showing that the potential type system is sound w.r.t. the cost semantics.

## B More Examples

Our type system is quite expressive and can be used to derive bounds on many more examples. In this section, we will derive bounds on several list processes. We will start with simple examples, such as the *nil*, *cons* and *append* processes. We will then derive bounds on stacks and queues being implemented using lists. Finally, we will conclude with some higher order functions such as *map* and $fold$. For each of the following examples, we assume the standard cost metric, where we count the number of messages exchanged, i.e. $M^{\mathsf{label}} = M^{\mathsf{channel}} = M^{\mathsf{close}} = 1$. First, we consider the list protocol as a simple session type.

$$\begin{aligned}\mathsf{list}_A = \oplus\{\ & \mathsf{cons} : A \otimes \mathsf{list}_A,\\ & \mathsf{nil} : \mathbf{1}\}\end{aligned}$$

The type prescribes that a process providing service of type $\mathsf{list}_A$ will either send a label $\mathsf{cons}$ followed by an element of type $A$ and recurse, or will send a $\mathsf{nil}$ label followed by a close message and then terminate. On the client side (i.e. a process that uses a channel of type $\mathsf{list}_A$ in its context), the opposite behavior is observed, i.e. a client receives the messages that the provider sends (sequence of $\mathsf{cons}$ labels and elements terminated by a $\mathsf{nil}$ label and the close messsage). The *append* process is a SILL implementation of the standard append function which appends two lists.

### B.1 Basic Processes

We present the implementations of *nil*, *cons* and *append* processes.

#### *nil*

The *nil* process is used to create an empty list. Formally, a *nil* process uses an empty context, and provides an empty list along a channel $l : \mathsf{list}_A$. Concretely, this means it sends a $\mathsf{nil}$ label followed by a $\mathsf{close}$ message along $l$. First, we introduce the resource-aware session type for $\mathsf{list}_A$.

$$\begin{aligned}\mathsf{list}_A = \oplus\{\ & \mathsf{nil}^0 : \mathbf{1}^0,\\ & \mathsf{cons}^0 : A \overset{0}{\otimes} \mathsf{list}_A\}\end{aligned}$$

This resource-aware type decorates each label and type operator with 0 potential, implying that none of the messages carry any potential, and the process potential needs to pay only for the cost of sending the messages. We present the implementation followed by the type derivation for the *nil* process.

$$\begin{array}{l}\cdot \overset{2}{\vdash} nil :: (l : \mathsf{list}_A)\\ \quad l \leftarrow nil =\\ \qquad l.\mathsf{nil}\ ; \qquad \% \quad \cdot \overset{1}{\vdash} l : \mathbf{1}^0\\ \qquad \mathsf{close}\ l \qquad \% \quad \cdot \overset{0}{\vdash} \cdot\end{array}$$

The type of *nil* process shows that the process potential needed is 2, which intuitively agrees with our cost model. The process sends two messages, each of them costing unit potential. We explain the type derivation briefly. The initial type of the process is $\cdot \stackrel{2}{\vdash} l : \mathsf{list}_A$. Now, for $l$ to behave as an empty list, the *nil* process needs to send the $\mathsf{nil}$ label first. As the $\mathsf{list}_A$ type prescribes, the $\mathsf{nil}$ label carries no potential, this send only costs 1. Updating the type of $l$ and the process potential, we get $\cdot \stackrel{1}{\vdash} l : \mathbf{1}^0$. Finally, the *nil* process needs to send the $\mathsf{close}$ message, which again costs 1 as the type $\mathbf{1}$ carries no potential in the type definition of $\mathsf{list}_A$. Thus, our type system successully verifies that the *nil* process needs a potential of 2, hence its resource usage is 2.

***cons***

Now, let's look at the *cons* process.

$(x : A)\ (t : \mathsf{list}^0_A) \stackrel{2}{\vdash} \mathit{cons} :: (l : \mathsf{list}^0_A)$
$l \leftarrow \mathit{cons} \leftarrow x\ t =$

| | | |
|---|---|---|
| $l.\mathsf{cons}$ ; | % | $(x : A)\ (t : \mathsf{list}^0_A) \stackrel{1}{\vdash} l : A \overset{0}{\otimes} \mathsf{list}^0_A$ |
| $\mathsf{send}\ l\ x$ ; | % | $(t : \mathsf{list}^0_A) \stackrel{0}{\vdash} l : \mathsf{list}^0_A$ |
| $l \leftarrow t$ | | |

We can do another annotated type for *cons*.

$\mathsf{list}^1_A = \oplus\{\ \mathsf{nil}^0 : \mathbf{1},$
$\qquad\qquad \mathsf{cons}^1 : A \overset{0}{\otimes} \mathsf{list}^1_A\}$

$(x : A)\ (t : \mathsf{list}^1_A) \stackrel{3}{\vdash} \mathit{cons} :: (l : \mathsf{list}^1_A)$
$l \leftarrow \mathit{cons} \leftarrow x\ t =$

| | | |
|---|---|---|
| $l.\mathsf{cons}$ ; | % | $(x : A)\ (t : \mathsf{list}^1_A) \stackrel{1}{\vdash} l : A \overset{0}{\otimes} \mathsf{list}^1_A$ |
| $\mathsf{send}\ l\ x$ ; | % | $(t : \mathsf{list}^1_A) \stackrel{0}{\vdash} l : \mathsf{list}^1_A$ |
| $l \leftarrow t$ | | |

***append***

Finally, let's try the *append* process.

$\mathsf{list}^2_A = \oplus\{\ \mathsf{nil}^0 : \mathbf{1},$
$\qquad\qquad \mathsf{cons}^2 : A \overset{0}{\otimes} \mathsf{list}^2_A\}$

$(l_1 : \mathsf{list}^2_A)\ (l_2 : \mathsf{list}^0_A) \stackrel{0}{\vdash} \mathit{append} :: (l : \mathsf{list}^0_A)$
$l \leftarrow \mathit{append} \leftarrow l_1\ l_2 =$

| | | | | |
|---|---|---|---|---|
| $\mathsf{case}\ l_1$ | $(\mathsf{cons} \Rightarrow$ | $x \leftarrow \mathsf{recv}\ l_1$ ; | % | $(x : A)\ (l_1 : \mathsf{list}^2_A)\ (l_2 : \mathsf{list}^0_A) \stackrel{2}{\vdash} l : \mathsf{list}^0_A$ |
| | | $l.\mathsf{cons}$ ; | % | $(x : A)\ (l_1 : \mathsf{list}^2_A)\ (l_2 : \mathsf{list}^0_A) \stackrel{1}{\vdash} l : \mathsf{list}^0_A$ |
| | | $\mathsf{send}\ l\ x$ ; | % | $(l_1 : \mathsf{list}^2_A)\ (l_2 : \mathsf{list}^0_A) \stackrel{0}{\vdash} l : \mathsf{list}^0_A$ |
| | | $l \leftarrow \mathit{append} \leftarrow l_1\ l_2$ | | |
| | $\mid \mathsf{nil} \Rightarrow$ | $\mathsf{wait}\ l_1$ ; | % | $(l_1 : \mathbf{1})\ (l_2 : \mathsf{list}^0_A) \stackrel{0}{\vdash} l : \mathsf{list}^0_A$ |
| | | $l \leftarrow l_2)$ | % | $(l_2 : \mathsf{list}^0_A) \stackrel{0}{\vdash} l : \mathsf{list}^0_A$ |

### B.2 Stacks as Lists

The stack interface introduced in Section 7 can be implemented using lists. First, we define the resource-aware types.

$$\begin{array}{l}\mathsf{list}^2_A = \oplus\{\ \mathsf{nil}^2 : \mathbf{1}^0, \\ \qquad\qquad \mathsf{cons}^2 : A \overset{0}{\otimes} \mathsf{list}^2_A\}\end{array}$$

$$\begin{array}{l}\mathsf{stack}^4_A = \&\{\ \mathsf{ins}^4 : A \overset{0}{\multimap} \mathsf{stack}^4_A, \\ \qquad\qquad \mathsf{del}^0 : \oplus\{\ \mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{stack}^4_A, \\ \qquad\qquad\qquad\qquad \mathsf{none}^0 : \mathbf{1}^0\}\}\end{array}$$

We need several sub-processes for the implementation of the stack using a list. We will implement and type them first.

$$\begin{array}{l}\cdot \ \vdash^4 \mathit{nil} :: (l : \mathsf{list}^2_A) \\ \quad l \leftarrow \mathit{nil} = \\ \qquad l.\mathsf{nil}\ ; \qquad\qquad \% \quad \cdot \vdash^1 l : \mathbf{1}^0 \\ \qquad \mathsf{close}\ l \qquad\qquad \% \quad \cdot \vdash^0 \cdot\end{array}$$

$$\begin{array}{l}(x : A)\ (t : \mathsf{list}^2_A)\ \vdash^4 \mathit{cons} :: (l : \mathsf{list}^2_A) \\ \quad l \leftarrow \mathit{cons} \leftarrow x\ t = \\ \qquad l.\mathsf{cons}\ ; \qquad\qquad \% \quad (x : A)\ (t : \mathsf{list}^2_A) \vdash^1 l : A \overset{0}{\otimes} \mathsf{list}^2_A \\ \qquad \mathsf{send}\ l\ x\ ; \qquad\qquad \% \quad (t : \mathsf{list}^2_A) \vdash^0 l : \mathsf{list}^2_A \\ \qquad l \leftarrow t\end{array}$$

Finally, we can implement the stack interface using two processes, the first is *stack_new*, which creates an empty list and uses it as an empty stack.

$$\begin{array}{l}\cdot \vdash^4 \mathit{stack\_new} :: s : (\mathsf{stack}^4_A) \\ \quad s \leftarrow \mathit{stack\_new} = \\ \qquad e \leftarrow \mathit{nil}\ ; \qquad\qquad (e : \mathsf{list}^2_A) \vdash^0 (s : \mathsf{stack}^4_A) \\ \qquad s \leftarrow \mathit{stack} \leftarrow e\end{array}$$

The main process is called *stack*. It uses a list in its context and provides service along $s$ which behaves as a stack.

$$\begin{array}{l}l : \mathsf{list}^2_A \vdash^0 \mathit{stack} :: s : (\mathsf{stack}^4_A) \\ \quad s \leftarrow \mathit{stack} \leftarrow l \\ \qquad \mathsf{case}\ s \\ \qquad\quad (\mathsf{ins} \Rightarrow x \leftarrow \mathsf{recv}\ s\ ; \qquad\qquad \% \quad (x : A)(l : \mathsf{list}^2_A) \vdash^4 s : \mathsf{stack}^2_A \\ \qquad\qquad\quad\ \ l' \leftarrow \mathit{cons} \leftarrow x\ t \qquad \% \quad l' : \mathsf{list}^2_A \vdash^0 s : \mathsf{stack}^4_A \\ \qquad\qquad\quad\ \ s \leftarrow \mathit{stack} \leftarrow l \\ \qquad\qquad \mathsf{del} \Rightarrow\ \mathsf{case}\ l \\ \qquad\qquad\qquad (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l\ ; \qquad\qquad \% \quad (x : A)(l : \mathsf{list}^2_A) \vdash^2 s : \oplus\{\mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{stack}^4_A, \mathsf{none}^0 : \mathbf{1}^0\} \\ \qquad\qquad\qquad\qquad\ \ s.\mathsf{some}\ ; \qquad\qquad \% \quad (x : A)(l : \mathsf{list}^2_A) \vdash^1 s : A \overset{0}{\otimes} \mathsf{stack}^4_A\end{array}$$

$$
\begin{array}{lll}
 & \mathsf{send}\ s\ x\ ; & \%\quad (l : \mathsf{list}^2_A) \stackrel{0}{\vdash} s : \mathsf{stack}^4_A \\
 & s \leftarrow \mathit{stack} \leftarrow l & \\
\mathsf{nil} \Rightarrow & s.\mathsf{none} & \%\quad (l : \mathbf{1}^0) \stackrel{1}{\vdash} s : \mathbf{1}^0 \\
 & \mathsf{wait}\ l\ ; & \%\quad \cdot \stackrel{1}{\vdash} s : \mathbf{1}^0 \\
 & \mathsf{close}\ s) &
\end{array}
$$

### B.3 Queues as 2 Lists

A queue can be implemented using 2 lists. Insertion in such a queue has a constant amortized cost. Since our type system supports amortized analysis, we can derive a constant resource bound for such an implementation. The resource-aware types we will be using are as follows.

$$
\begin{array}{l}
\mathsf{list}^{(2,2)}_A = \oplus\{\ \mathsf{nil}^2 : \mathbf{1}^0, \\
\qquad\qquad \mathsf{cons}^2 : A \overset{0}{\otimes} \mathsf{list}^{(2,2)}_A\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{list}^4_A = \oplus\{\ \mathsf{nil}^0 : \mathbf{1}^0, \\
\qquad\qquad \mathsf{cons}^4 : A \overset{0}{\otimes} \mathsf{list}^4_A\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{queue}^{(6,2)}_A = \&\{\ \mathsf{enq}^6 : A \overset{0}{\multimap} \mathsf{queue}^{(6,2)}_A, \\
\qquad\qquad \mathsf{deq}^2 : \oplus\{\ \mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{queue}^{(6,2)}_A, \\
\qquad\qquad\qquad\qquad \mathsf{none}^0 : \mathbf{1}^0\}\}
\end{array}
$$

Again, we use the *nil* and *cons* sub-processes with a different type.

$$
\begin{array}{lll}
\cdot \stackrel{2}{\vdash} \mathit{nil} :: (l : \mathsf{list}^4_A) & & \\
\quad l \leftarrow \mathit{nil} = & & \\
\qquad l.\mathsf{nil}\ ; & \%\quad \cdot \stackrel{1}{\vdash} l : \mathbf{1}^0 & \\
\qquad \mathsf{close}\ l & \%\quad \cdot \stackrel{0}{\vdash} \cdot &
\end{array}
$$

$$
\begin{array}{lll}
(x : A)\ (t : \mathsf{list}^4_A) \stackrel{6}{\vdash} \mathit{cons} :: (l : \mathsf{list}^4_A) & & \\
\quad l \leftarrow \mathit{cons} \leftarrow x\ t = & & \\
\qquad l.\mathsf{cons}\ ; & \%\quad (x : A)\ (t : \mathsf{list}^4_A) \stackrel{1}{\vdash} l : A \overset{0}{\otimes} \mathsf{list}^4_A & \\
\qquad \mathsf{send}\ l\ x\ ; & \%\quad (t : \mathsf{list}^4_A) \stackrel{0}{\vdash} l : \mathsf{list}^4_A & \\
\qquad l \leftarrow t & &
\end{array}
$$

The main process *queue*2 acts as a client for 2 lists, and provides service along a queue interface. We provide the implementation and its type derivation below.

$$
\begin{array}{lll}
(\mathit{in} : \mathsf{list}^4_A)\ (\mathit{out} : \mathsf{list}^{(2,2)}_A) \stackrel{0}{\vdash} \mathit{queue}2 :: s : (\mathsf{queue}^{(6,2)}_A) & & \\
\quad s \leftarrow \mathit{queue}2 \leftarrow l & & \\
\qquad \mathsf{case}\ s & & \\
\qquad\quad (\mathsf{enq} \Rightarrow x \leftarrow \mathsf{recv}\ s\ ; & \%\quad (x : A)(\mathit{in} : \mathsf{list}^4_A)\ (\mathit{out} : \mathsf{list}^{(2,2)}_A) \stackrel{6}{\vdash} s : \mathsf{queue}^{(6,2)}_A & \\
\qquad\qquad \mathit{in}' \leftarrow \mathit{cons} \leftarrow x\ \mathit{in} & \%\quad (\mathit{in}' : \mathsf{list}^4_A)\ (\mathit{out} : \mathsf{list}^{(2,2)}_A) \stackrel{0}{\vdash} s : \mathsf{queue}^{(6,2)}_A & \\
\qquad\qquad s \leftarrow \mathit{queue}2 \leftarrow \mathit{in}\ \mathit{out} & & \\
\qquad\quad \mathsf{deq} \Rightarrow\ \mathsf{case}\ \mathit{out} & &
\end{array}
$$

$$
\begin{array}{lllll}
(\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ out\ ; & \% & (x : A)\ (in : \mathsf{list}^4_A)\ (out : \mathsf{list}^{(2,2)}_A) \stackrel{2}{\vdash} & & \\
 & \% & \quad s : \oplus\{\mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{queue}^{(6,2)}_A, \mathsf{none}^0 : \mathbf{1}^0\} & & \\
\quad s.\mathsf{some}\ ; & \% & (x : A)\ (in : \mathsf{list}^2_A)\ (out : \mathsf{list}^0_A) \stackrel{1}{\vdash} s : A \overset{0}{\otimes} \mathsf{queue}^4_A & & \\
\quad \mathsf{send}\ s\ x\ ; & \% & (in : \mathsf{list}^2_A)\ (out : \mathsf{list}^0_A) \stackrel{0}{\vdash} s : \mathsf{queue}^4_A & & \\
\quad s \leftarrow queue2 \leftarrow in\ out & & & & \\
\mathsf{nil} \Rightarrow \quad \mathsf{wait}\ out\ ; & \% & (in : \mathsf{list}^2_A) \stackrel{4}{\vdash} s : \mathsf{queue}^4_A & & \\
\quad out' \leftarrow rev \leftarrow in & \% & (out' : \mathsf{list}^0_A) \stackrel{2}{\vdash} s : \mathsf{queue}^4_A & & \\
\quad \mathsf{case}\ out' & & & & \\
\quad\quad (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ out'\ ; & \% & (x : A)\ (out' : \mathsf{list}^0_A) \stackrel{2}{\vdash} & & \\
 & \% & \quad s : \oplus\{\mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{queue}^4_A, \mathsf{none}^0 : \mathbf{1}^0\} & & \\
\quad\quad\quad s.\mathsf{some}\ ; & \% & (x : A)\ (out' : \mathsf{list}^0_A) \stackrel{1}{\vdash} s : A \overset{0}{\otimes} \mathsf{queue}^4_A & & \\
\quad\quad\quad \mathsf{send}\ s\ x\ ; & \% & (out' : \mathsf{list}^0_A) \stackrel{0}{\vdash} s : \mathsf{queue}^4_A & & \\
\quad\quad\quad in' \leftarrow nil\ ; & \% & (in' : \mathsf{list}^2_A)\ (out' : \mathsf{list}^0_A) \stackrel{0}{\vdash} s : \mathsf{queue}^4_A & & \\
\quad\quad\quad s \leftarrow queue2 \leftarrow in'\ out' & & & & \\
\quad\quad \mathsf{nil} \Rightarrow \quad \mathsf{wait}\ out'\ ; & \% & \cdot \stackrel{0}{\vdash} s : \oplus\{\mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{queue}^4_A, \mathsf{none}^0 : \mathbf{1}^0\} & & \\
\quad\quad\quad s.\mathsf{none}\ ; & \% & \cdot \stackrel{0}{\vdash} s : \mathbf{1}^0 & & \\
\quad\quad\quad \mathsf{close}\ s))) & & & &
\end{array}
$$

### B.4 Higher Order Functions

Let's consider some higher order functions. First, let's consider the *map* function.

$$
\begin{array}{l}
\mathsf{mapper}^2_{AB} = \&\{\ \mathsf{next}^0 : A \overset{0}{\multimap} B \overset{2}{\otimes} \mathsf{mapper}^2_{AB}, \\
\qquad\qquad \mathsf{done}^0 : \mathbf{1}^2\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{list}^2_A = \oplus\{\ \mathsf{nil}^1 : \mathbf{1}^0, \\
\qquad\qquad \mathsf{cons}^2 : A \overset{0}{\otimes} \mathsf{list}^2_A\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{list}^0_A = \oplus\{\ \mathsf{nil}^0 : \mathbf{1}^0, \\
\qquad\qquad \mathsf{cons}^0 : A \overset{0}{\otimes} \mathsf{list}^0_A\}
\end{array}
$$

Now, let's consider the implementation of the *map* function.

$$
\begin{array}{lllll}
(l : \mathsf{list}^2_A)\ (m : \mathsf{mapper}^2_{AB}) \stackrel{0}{\vdash} map :: (k : \mathsf{list}^0_A) & & & & \\
\quad k \leftarrow map \leftarrow l\ m = & & & & \\
\quad\quad \mathsf{case}\ l & & & & \\
\quad\quad\quad (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l\ ; & \% & (x : A)\ (l : \mathsf{list}^2_A)\ (m : \mathsf{mapper}^2_{AB}) \stackrel{2}{\vdash} (k : \mathsf{list}^0_A) & & \\
\quad\quad\quad\quad m.\mathsf{next}\ ; & \% & (x : A)\ (l : \mathsf{list}^2_A)\ (m : A \overset{0}{\multimap} B \overset{2}{\otimes} \mathsf{mapper}^2_{AB}) \stackrel{1}{\vdash} (k : \mathsf{list}^0_A) & & \\
\quad\quad\quad\quad \mathsf{send}\ m\ x\ ; & \% & (l : \mathsf{list}^2_A)\ (m : B \overset{2}{\otimes} \mathsf{mapper}^2_{AB}) \stackrel{0}{\vdash} (k : \mathsf{list}^0_A) & & \\
\quad\quad\quad\quad y \leftarrow \mathsf{recv}\ m\ ; & \% & (l : \mathsf{list}^2_A)\ (y : B)\ (m : \mathsf{mapper}^2_{AB}) \stackrel{2}{\vdash} (k : \mathsf{list}^0_A) & & \\
\quad\quad\quad\quad k.\mathsf{cons}\ ; & \% & (l : \mathsf{list}^2_A)\ (y : B)\ (m : \mathsf{mapper}^2_{AB}) \stackrel{1}{\vdash} (k : A \overset{0}{\otimes} \mathsf{list}^0_A) & & \\
\quad\quad\quad\quad \mathsf{send}\ k\ y\ ; & \% & (l : \mathsf{list}^2_A)\ (m : \mathsf{mapper}^2_{AB}) \stackrel{0}{\vdash} (k : \mathsf{list}^0_A) & & \\
\quad\quad\quad\quad k \leftarrow map \leftarrow l\ m & & & & \\
\quad\quad\quad \mathsf{nil} \Rightarrow \quad \mathsf{wait}\ l\ ; & \% & (m : \mathsf{mapper}^2_{AB}) \stackrel{1}{\vdash} (k : \mathsf{list}^0_A) & &
\end{array}
$$

$m$.done ; % $(m : \mathbf{1}^2) \stackrel{|0}{\vdash} (k : \mathsf{list}^0_A)$
wait $m$ ; % $\cdot \stackrel{|2}{\vdash} (k : \mathsf{list}^0_A)$
$k$.nil ; % $\cdot \stackrel{|1}{\vdash} (k : \mathbf{1}^0)$
close $k$)

Now, let's consider the $fold$ function.

$$\mathsf{folder}^0_{AB} = \&\{\ \mathsf{next}^0 : A \stackrel{0}{\multimap} B \stackrel{0}{\multimap} B \stackrel{0}{\otimes} \mathsf{folder}^0_{AB},\ \mathsf{done}^0 : \mathbf{1}^0\}$$

$$\mathsf{list}^3_A = \oplus\{\ \mathsf{nil}^1 : \mathbf{1}^0,\ \mathsf{cons}^3 : A \stackrel{0}{\otimes} \mathsf{list}^2_A\}$$

Now, let's consider the implementation of the $fold$ function.

$(l : \mathsf{list}^3_A)\ (m : \mathsf{folder}^0_{AB})\ (b : B) \stackrel{|0}{\vdash} fold :: (r : B)$
$r \leftarrow fold \leftarrow l\ m\ b =$
case $l$
(cons $\Rightarrow$ $x \leftarrow$ recv $l$ ; % $(x : A)\ (l : \mathsf{list}^2_A)\ (m : \mathsf{folder}^0_{AB})\ (b : B) \stackrel{|3}{\vdash} (r : B)$
$m$.next ; % $(x : A)\ (l : \mathsf{list}^3_A)\ (m : A \stackrel{0}{\multimap} B \stackrel{0}{\multimap} B \stackrel{2}{\otimes} \mathsf{folder}^0_{AB})\ (b : B) \stackrel{|2}{\vdash} (r : B)$
send $m\ x$ ; % $(l : \mathsf{list}^3_A)\ (m : B \stackrel{0}{\multimap} B \stackrel{2}{\otimes} \mathsf{folder}^0_{AB})\ (b : B) \stackrel{|1}{\vdash} (r : B)$
send $m\ b$ ; % $(l : \mathsf{list}^3_A)\ (m : B \stackrel{2}{\otimes} \mathsf{folder}^0_{AB}) \stackrel{|0}{\vdash} (r : B)$
$y \leftarrow$ recv $m$ ; % $(l : \mathsf{list}^2_A)\ (y : B)\ (m : \mathsf{folder}^0_{AB}) \stackrel{|2}{\vdash} (r : B)$
$k \leftarrow map \leftarrow l\ m$
nil $\Rightarrow$ wait $l$ ; % $(m : \mathsf{folder}^0_{AB})\ (b : B) \stackrel{|1}{\vdash} (r : B)$
$m$.done ; % $(m : \mathbf{1}^0)\ (b : B) \stackrel{|0}{\vdash} (r : B)$
wait $m$ ; % $(b : B) \stackrel{|0}{\vdash} (r : B)$
$r \leftarrow b$)