Work Analysis with Resource-Aware Session Types

Ankush Das
Carnegie Mellon University

Jan Hoffmann
Carnegie Mellon University

Frank Pfenning
Carnegie Mellon University

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.

CCS Concepts • **Theory of computation → Linear logic**; **Program analysis**; *Operational semantics*;

Keywords Session types, Resource analysis, Type systems

ACM Reference Format:
Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS '18: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209146

1 Introduction

In the past years, there has been increasing interest in supporting developers to statically reason about the resource usage of their code. This research has numerous applications such as prevention of side channels leaking secret information [4, 25, 28], identification of complexity bugs [29], support of scheduling decisions [1], and help in profiling [18]. 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 [2, 3, 15]. The lack of analysis tools is in sharp contrast to the need for programming language support in this area: concurrent and distributed programs are 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.

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 the structure, protocols and resource bounds for inter-process communication and to perform a compositional and precise amortized analysis. *Session types* [24] 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. Recently, message-passing concurrency has been put onto a firm logical foundation by exhibiting a Curry-Howard isomorphism between intuitionistic linear logic and session-typed communication [8]. We use such protocols as the basis for 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 classical amortized analysis [33]) may be spent by 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 performed by a system, counting the number of messages that are exchanged. While this alone does not yet account for the concurrent nature of message-passing programs, it constitutes a 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 derive 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 also serve as input to a Brent-style theorem [7] that relates the complexity of the execution of a program on a k-processor machine to the program's work (focus of this paper) and span (resource usage with an unlimited number of processors).

Our analysis is based on a linear type system that combines standard binary session types as available in the SILL language [30, 35], and type-based amortized resource analysis [19, 21]. 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 interacts with, this design

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209146

provides a compositional resource specification. For closed programs (which evolve into a closed network of interacting processes) the bound becomes a single constant. In addition to the natural compositionality of type systems we also preserve the necessary support for deriving resource annotations via LP solving, a key feature of type-based amortized analysis. 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 [30, 35] that combines them monadically.

A conceptual challenge 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 as functions of interactions (messages sent) on 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 non-local fashion. Our solution is that messages and processes carry potential to 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 the type system with respect to an operational cost semantics that tracks the total number of messages exchanged in a network of communicating processes. We demonstrate the effectiveness of the technique by deriving tight bounds for standard examples of amortized analysis from 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.

2 Overview

We motivate and informally introduce 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.

Session Types. Session types were introduced by Honda [24] to describe the structure of communication just like standard data types describe the structure of data. We follow the approach and syntax of SILL [30, 35] 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 zero and one, eventually terminated by \$. Because the provider chooses which messages to send, we call this an *internal choice*, 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, labeled 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 zero, one, one, \$, *end*. A client of a channel c : bits has to branch on whether it receives zero, 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 : bits then c must afterwards have type $\mathbf{1}$. The next message along c must be *end* and its client has 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 inc messages to a counter, until we want to read its value and send val. At that point the counter will send a stream of bits representing its value as prescribed by the type bits. From the provider's point of view, a counter presents an *external choice*, since the client chooses between inc or 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 labeled 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.

Modeling a binary counter. We describe an implementation of a binary 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 exchanged along all channels in the system.

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 observe this by introducing a potential of 1 for every bit that is one and using this potential to *pay* for the expensive case in which an increment triggers multiple 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 the remaining potential of 2 is passed along 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 (indicated by • between $b0$ and s_3) and uses channel s_2. In our notation, we would write this as

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

We see that, logically, parallel composition with a private shared channel corresponds to an application of the cut rule. The *definitions* of e, $b0$ and $b1$ can be found in Figure 2. 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

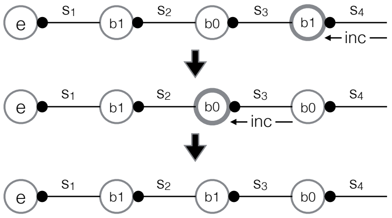

Figure 1. Binary counter system representing $5 = (101)_2$ with messages triggered when inc message is received on s_4.

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 inc. We do this using a superscript indicating the required potential with the label, postponing the discussion of 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.

$$t : \mathsf{ctr} \overset{0}{\vdash} b0 :: (s : \mathsf{ctr}) \quad (1)$$

$$t : \mathsf{ctr} \overset{1}{\vdash} b1 :: (s : \mathsf{ctr}) \quad (2)$$

$$\cdot \overset{0}{\vdash} e :: (s : \mathsf{ctr}) \quad (3)$$

With our formal typing rules (Section 5) we can verify these typing constraints, using the definitions of $b0$, $b1$ and e from Figure 2. Informally, we can reason as follows:

$b0$: When $b0$ receives 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 inc it receives 1 unit of potential which, when combined with the stored one, makes 2 units. It needs to send an inc message 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 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 val? Recall that $\mathsf{bits} = \oplus\{\mathsf{zero} : \mathsf{bits}, \mathsf{one} : \mathsf{bits}, \$: \mathbf{1}\}$. In our implementation, upon receiving a val message, a $b0$ or $b1$ process will first respond with zero or one respectively. It then sends 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 inc and val messages by sending 2 more units with inc and 2 units with val. That is, the following type is correct:

$$\mathsf{bits} = \oplus\{\mathsf{zero}^0 : \mathsf{bits}, \mathsf{one}^0 : \mathsf{bits}, \$^0 : \mathbf{1}^0\}$$

$$\mathsf{ctr} = \&\{\mathsf{inc}^3 : \mathsf{ctr}, \mathsf{val}^2 : \mathsf{bits}\}$$

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 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. This 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 [17]) 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:

$$\mathsf{bits} = \oplus\{\mathsf{zero}^0 : \mathsf{bits}, \mathsf{one}^0 : \mathsf{bits}, \$^0 : \mathbf{1}^0\}$$

$$\mathsf{ctr}[n] = \&\{\mathsf{inc}^1 : \mathsf{ctr}[n+1], \mathsf{val}^{2\lceil\log(n+1)\rceil+2} : \mathsf{bits}\}$$

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

$$t : \mathsf{ctr}[n] \overset{0}{\vdash} b0 :: (s : \mathsf{ctr}[2n])$$

$$t : \mathsf{ctr}[n] \overset{1}{\vdash} b1 :: (s : \mathsf{ctr}[2n+1])$$

$$\cdot \overset{0}{\vdash} e :: (s : \mathsf{ctr}[0])$$

Our type system verifies these types against the implementation of $b0$, $b1$, and e (see Figure 2). 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 precise 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 definitions. 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 typed as

$$\cdot \overset{p}{\vdash} Q :: (c : \mathbf{1}^0)$$

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

3 Resource-Aware SILL

We briefly introduce the linear, process-only fragment of SILL [30, 35], which integrates functional and concurrent computation. A program in SILL is a collection of processes exchanging messages

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

through channels. A new process is *spawned* by invoking a process definition, which also creates a fresh channel. The newly spawned process acts as a *provider* of the fresh channel while the parent process acts as its *client*. 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 along with a brief description of their communication protocol. They follow the type grammar below.

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

V denotes a type variable. Types may be defined mutually recursively in a global signature, where type definitions are constrained to be *contractive* [14]. This allows us to treat them equi-recursively, meaning we can silently replace a type variable by its definition for type-checking purposes.

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 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 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. Dually, a provider of $S \multimap T$ receives a channel of type S from its client. The types of the provider and client change 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 SILL.

$$\begin{aligned} P \quad ::= \quad & x \leftarrow X \leftarrow \overline{y} \ ; \ P \mid x \leftarrow y \mid \mathsf{close}\ x \mid \mathsf{wait}\ x \ ; \ P \\ & \mid x.l_k \ ; \ P \mid \mathsf{case}\ x\ (l_i \Rightarrow P_i)_{i \in I} \\ & \mid \mathsf{send}\ x\ w \ ; \ P \mid y \leftarrow \mathsf{recv}\ x \ ; \ P \end{aligned}$$

The term $x \leftarrow X \leftarrow \overline{y} \ ; \ P$ invokes a process definition X to spawn a new process Q, which uses the channels in $\overline{y}$ as a client and provides service along a fresh channel substituted for x in P. 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. 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 annotations in red). For each connective in Table 1, the first row presents the perspective of the provider, while the second presents 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. The last column provides the operational description.

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):

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

Figure 2 presents implementations of the $b0$, $b1$ and e processes respectively that were analyzed in Section 2. The comments (starting with %) show the channel types after the interaction on each line (again ignoring the annotations in red). Since the $b0$ process provides an external choice along s, $b0$ needs to branch based on the label received (line 4). If it receives the label inc, the type of the channel s updates to ctr, as indicated on the typing in the comment. At this point, we spawn the $b1$ process whose type (line 8) matches with the type on line 4. If instead $b0$ receives the val label along s, it continues at type bits. It sends zero (since the lowest bit is indeed zero) and 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 7) and $b0$ can terminate by forwarding further communication along t to s.

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

4 Cost Semantics

We present an operational cost semantics for Resource-Aware SILL tracking the work performed by the system. Our semantics is a substructural operational semantics [31] based on multiset rewriting [10] and asynchronous communication [30]. It can be seen as a combination of an asynchronous version of a recent synchronous session-type semantics [6] with the cost tracking semantics of Concurrent C0 [32]. The technical advantage of our semantics is that it avoids the complex operational artifacts of Silva et al. [32] such as message buffers: processes and messages can be typed with exactly the same typing rules, changing only the cost metric.

```
$(t : \mathsf{ctr}[n]) \overset{0}{\vdash} b0 :: (s : \mathsf{ctr}[2n])$
  $s \leftarrow b0 \leftarrow t =$
    case $s$
      (inc $\Rightarrow s \leftarrow b1 \leftarrow t$                 % $(t : \mathsf{ctr}[n]) \overset{1}{\vdash} s : \mathsf{ctr}[2n+1]$
      | val $\Rightarrow s.\mathsf{zero}$ ;                          % $(t : \mathsf{ctr}[n]) \overset{2\lceil\log(2n+1)\rceil+2-1}{\vdash} s : \mathsf{bits}$
              $t.\mathsf{val}$ ;               % $(t : \mathsf{bits}) \overset{2\lceil\log(2n+1)\rceil+1-2\lceil\log(n+1)\rceil-3}{\vdash} s : \mathsf{bits}$
              $s \leftarrow t$)                                   % $(t : \mathsf{bits}) \overset{0}{\vdash} s : \mathsf{bits}$
$(t : \mathsf{ctr}[n]) \overset{1}{\vdash} b1 :: (s : \mathsf{ctr}[2n+1])$
  $s \leftarrow b1 \leftarrow t =$
    case $s$
      (inc $\Rightarrow t.\mathsf{inc}$ ;                            % $(t : \mathsf{ctr}[n+1]) \overset{0}{\vdash} s : \mathsf{ctr}[2n+2]$
              $s \leftarrow b0 \leftarrow t$
      | val $\Rightarrow s.\mathsf{one}$ ;                           % $(t : \mathsf{ctr}[n]) \overset{2\lceil\log(2n+2)\rceil+2-1}{\vdash} s : \mathsf{bits}$
              $t.\mathsf{val}$ ;               % $(t : \mathsf{bits}) \overset{2\lceil\log(2n+2)\rceil+1-2\lceil\log(n+1)\rceil-3}{\vdash} s : \mathsf{bits}$
              $s \leftarrow t$)                                   % $(t : \mathsf{bits}) \overset{0}{\vdash} s : \mathsf{bits}$
$\cdot \overset{0}{\vdash} e :: (s : \mathsf{ctr}[0])$
  $s \leftarrow e =$
    case $s$
      (inc $\Rightarrow t \leftarrow e$ ;                            % $(t : \mathsf{ctr}[0]) \overset{1}{\vdash} (s : \mathsf{ctr}[1])$
              $s \leftarrow b1 \leftarrow t$
      | val $\Rightarrow s.e$ ;                                      % $\cdot \overset{2\lceil\log(0+1)\rceil+2-1}{\vdash} s : \mathbf{1}^0$
              close $s$)
```

Figure 2. Implementations for the $b0$, $b1$ and e processes with their type derivations demonstrating the binary counter.

Our cost semantics is asynchronous, that is, processes can continue their evaluation without waiting 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 [30]).

A collection of communicating processes 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 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 the 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 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.

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

The work is tracked by the local counter w in $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$ propositions. For process P, w maintains the total work performed by P so far. 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 process terminates we remove the respective predicate from the configuration, but preserve its work done. A process can terminate either by sending a close message, or by forwarding. In either case, we conveniently preserve the process’ work in the msg predicate to pass it on to the client process.

We will only count communication costs, ignoring internal computation. To this end, we introduce 3 costs, M^{label}, M^{channel} and M^{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 each cost to 1 will lead to counting the total number of messages exchanged.

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 match a part of the configuration. Consider for instance the rule $\&C_s$ that describes a client that sends label l_k along channel c.

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

The above 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 continuation channel c' and replace the premise by $\mathsf{proc}(d, w + M^{\mathsf{label}}, [c'/c]P)$ and $\mathsf{msg}(c', 0, c.l_k \ ; \ c' \leftarrow c)$ propositions. 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 sender is incremented by M^{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.

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

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 tells us that the next message will arrive on channel c'. Any work w encoded in the message is transferred to the recipient process.

The rest of the rules of cost semantics are given in Figure 3. The rule spawn_c describes the creation of a new channel c along with spawning a new process X implemented by P_c. This implementation is looked up in a signature for the semantics Σ which 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 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 (fwd_r^+) or provider (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

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

$$\frac{\mathsf{proc}(c, w, c \leftarrow d) \quad \mathsf{msg}(e, w', M_c)}{\mathsf{msg}(e, w + w', [d/c]M_c)}\ \mathsf{fwd}_r^- \qquad \frac{\mathsf{proc}(c, w, \mathsf{close}\ c)}{\mathsf{msg}(c, w + M^{\mathsf{close}}, \mathsf{close}\ c)}\ \mathbf{1}C_s \qquad \frac{\mathsf{msg}(c, w, \mathsf{close}\ c) \quad \mathsf{proc}(d, w', \mathsf{wait}\ c\ ;\ Q)}{\mathsf{proc}(d, w + w', Q)}\ \mathbf{1}C_r$$

$$\frac{\mathsf{proc}(c, w, c.l_k\ ;\ P) \qquad (c' \text{ fresh})}{\mathsf{proc}(c', w + M^{\mathsf{label}}, [c'/c]P) \quad \mathsf{msg}(c, 0, c.l_k\ ;\ c \leftarrow c')}\ \oplus C_s \qquad \frac{\mathsf{msg}(c, w, c.l_k\ ;\ c \leftarrow c') \quad \mathsf{proc}(d, w', \mathsf{case}\ c\ (l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(d, w + w', [c'/c]Q_k)}\ \oplus C_r$$

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

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

Figure 3. Cost semantics tracking total work for programs in SILL

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 present the resource-aware type system of our language which extends the linear-only fragment of SILL [30, 35] with resource annotations. It is in turn based on intuitionistic linear logic [16] 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 the 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 here 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 need 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 obtained by annotating simple session types with potential, defined by the following grammar.

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

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

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

Intuitively, the judgment describes a process in state P using the context Ω and signature Σ 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 Ω. The resource annotation q is a natural number and defines the potential stored in the process P.

When reasoning about the work performed by a system, we reason parametrically in certain quantities, such as the value of a counter, the number of elements in a queue, the potential carried by a message, or even the type of the elements in a queue. In an implementation, we would have to make type families, index domains, constraint solving, etc. explicit, but fortunately we can avoid the notational overhead that this entails. This is because the types and rules are always *schematic* in their parameters and quantification over these parameters can remain entirely at the metalevel. We model this by allowing (conceptually) infinite signatures with all instances of parametric definitions. In this way, when we reason parametrically we can be assured that any instance of what we derive is indeed a valid judgment. This allows us to focus on the key conceptual and technical contributions of our approach.

Σ defines this signature containing type and process definitions. It is defined as a possibly infinite set of type definitions $V = S_V$ and process definitions $x : S \leftarrow \mathcal{X} \mathrel{@} q \leftarrow \overline{y : W} = P_{x,\overline{y}}$. 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. The process definition $x : S \leftarrow \mathcal{X} \mathrel{@} q \leftarrow \overline{y : W} = P_{x,\overline{y}}$ defines a (possibly recursive) process named $\mathcal{X}$ implemented by $P_{x,\overline{y}}$ providing along channel $x : S$ and using the channels $\overline{y : W}$ as a client. The process also stores a potential q, shown as $\mathcal{X} \mathrel{@} q$ in the definition. For instance, $s : \mathsf{ctr}[2n] \leftarrow b0 \mathrel{@} 0 \leftarrow t : \mathsf{ctr}[n] = P_{s,t}$ ($P_{s,t}$ 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, $\Sigma; \Omega \vdash^{q}_{\mathsf{cf}} P :: (x : S)$, which follows the same typing rules as Figure 4, 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 Σ is used in the derivation of the cost-free judgment.

The principle behind 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.

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

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

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

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

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

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

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

$$\frac{q \geq r + M^{\mathsf{close}}}{\Sigma \; ; \; \cdot \vdash^{q} \mathsf{close}\; x :: (x : \mathbf{1}^r)} \; \mathbf{1}R$$

$$\frac{q + r \geq p \quad \Sigma \; ; \; \Omega \vdash^{p} Q :: (z : U)}{\Sigma \; ; \; \Omega \; (x : \mathbf{1}^r) \vdash^{q} \mathsf{wait}\; x \; ; \; Q :: (z : U)} \; \mathbf{1}L$$

Figure 4. Typing rules for session-typed programs

Since the continuation Q is typed with potential p, and the potential sent with the label is r_k, the total original potential need be at least $p + r_k + M^{\mathsf{label}}$. Thus, 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 prescribes that the branch P_i is typed with potential q_i. Moreover, 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 Ω into $\Omega_1 \; \Omega_2$, and we use Ω_1 to type the newly spawned process and Ω_2 for the continuation Q_x.

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

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 4. They are similar to the discussed rules and we omit their explanation.

As an illustration, the implementation and resource-aware type derivation (marked in red) of the binary counter is presented in Figure 2. The derivation provides a proof certificate that increment has an amortized cost of 1, while reading a value costs $2 \lceil \log(n + 1) \rceil + 2$.

$$\frac{}{\Sigma \; ; \; (\cdot) \overset{0}{\vDash} (\cdot) :: (\cdot)} \; \mathsf{emp}$$

$$\frac{\Sigma \; ; \; \Omega \overset{E}{\vDash} C :: \Omega' \qquad \Sigma \; ; \; \Omega' \overset{E'}{\vDash} C' :: \Omega''}{\Sigma \; ; \; \Omega \overset{E+E'}{\vDash} (C \; C') :: \Omega''} \; \mathsf{compose}$$

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

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

Figure 5. Typing rules for a configuration

6 Soundness

This section concludes the discussion of Resource-Aware SILL by demonstrating 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 interacting with other processes. Thus, we need to extend the typing rules to arbitrary configurations.

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

A signature Σ is *well formed* if (a) every type definition $V = S_V$ in Σ is contractive, and (b) every process definition $x : S \leftarrow \mathcal{X} \; @ \; p \leftarrow \overline{y : W} = P_{x, \overline{y}}$ in Σ is well typed according to the process typing judgment, that is, $\Sigma \; ; \; \overline{y : W} \vdash^{p} P_{x, \overline{y}} :: (x : S)$. Note that the same process name $\mathcal{X}$ can have different resource-aware types in the signature Σ. We pick the appropriate type while applying the spawn rule.

We use the following judgment to type a configuration.

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

It states that Σ is well-formed and that the configuration C uses the channels in the context Ω_1 and provides the channels in the context Ω_2. The natural number E denotes the sum of the total potential and work done by the system. We call E the energy of the configuration. The configuration typing judgment is defined using the rules presented in Figure 5. The rule emp defines that an empty configuration is well-typed with energy 0. The compose rule composes two configurations C and C'; C provides service on the channels in Ω' while C' uses the channels in Ω'. The energy of the composed configuration $C \; C'$ is obtained by summing up their individual energies. The rule C_{proc} creates a configuration out of a single process. The energy of this singleton configuration is obtained by adding the potential of the process and the work performed by it. Similarly, the rule C_{msg} creates a configuration out of a single message. Messages are typed in a cost-free judgment where $M^{\mathsf{label}} = M^{\mathsf{channel}} = M^{\mathsf{close}} = 0$, introduced in Section 5.

Soundness Theorem 6.1 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 energy of a configuration never increases during an evaluation step, i.e., the energy remains conserved with possibly some loss (due to throwing away potential, or friction).

Theorem 6.1 (Soundness). *Consider a well-typed configuration C w.r.t. a well-formed signature Σ such that $\Sigma; \Omega_1 \overset{E}{\vDash} C :: \Omega_2$. If $C \mapsto C'$, then there exist Ω_1', Ω_2' and E' such that $\Sigma; \Omega_1' \overset{E'}{\vDash} C' :: \Omega_2'$ and $E' \leq E$.*

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 (refer our technical report [12] for the complete proof). The preservation theorem is a corollary since soundness implies that the configuration C' is well-typed.

The soundness implies that the energy of an initial configuration is an upper bound on the energy of any configuration it will ever step to. In particular, if a configuration starts with 0 work, the initial potential (equal to initial energy) is an upper bound on the total work performed by an evaluation starting in that configuration.

Corollary 6.2 (Upper Bound). *If $\Sigma; \Omega_1 \overset{E}{\vDash} C :: \Omega_2$, and $C \mapsto^* C'$, then $E \geq W'$, where W' is the total work performed by the configuration C', i.e., the sum of the work performed by each process and message in C'. In particular, if the work done by the initial configuration C is 0, then the total potential P of the initial configuration satisfies $P \geq W'$.*

Proof. Applying the Soundness theorem successively, we get that if $C \mapsto^* C'$ and $\Sigma; \Omega_1 \overset{E}{\vDash} C :: \Omega_2$ and $\Sigma; \Omega_1' \overset{E'}{\vDash} C' :: \Omega_2'$, then $E' \leq E$. Since $W' \leq E'$ and $P = E$ (no initial work), we combine the inequalities to obtain $P \geq W'$. □

The progress theorem of Resource-Aware SILL is a direct consequence of progress in SILL [35]. 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 analyze the total work performed by a concurrent stack or queue implementation. Both these data structures 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 store_A (parameterized by type variable 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 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 an element stored in the queue (some), in which case it deletes this element, sends it to the client, and recurses.

To account for the resource cost, we 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 again set $M^{\mathsf{label}} = M^{\mathsf{channel}} = M^{\mathsf{close}} = 1$ to obtain a concrete bound.

Stacks The type for stacks is defined as follows.

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

A stack is implemented using a sequence of *elem* processes terminated by an *empty* process. Each *elem* process stores an element of the stack, while *empty* denotes the end of stack.

Inserting an element simply spawns a new *elem* process (which has no cost in our semantics), thus having no resource cost. Deleting an element terminates the *elem* process at the head. Before termination, it sends two messages back to the client, either none followed by close or some followed by element. Thus, deletion has a resource cost of 2. This is reflected in the stack_A type, where ins and del are annotated with 0 and 2 units of potential respectively.

Queues A queue is implemented by a sequence of *elem* processes (each storing one element) terminated by the *empty* process.

The queue interface is achieved by using the same store_A type 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 (for $n > 0$).

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

The $\dot{-}$ operator denotes the monus operator defined as $a \dot{-} b = \max(0, a - b)$. This prevents the type $\mathsf{queue}_A[0]$ from referring the undefined type $\mathsf{queue}_A[-1]$ in the del label. Resource-aware session types also allow us to provide a more precise type for queue_A, i.e., different types for $\mathsf{queue}_A[0]$ and $\mathsf{queue}_A[n]$ for $n > 0$.

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

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{\ &\mathsf{ins}^{2n} : A \overset{0}{\multimap} \mathsf{queue}_A[n+1],\\ &\mathsf{del}^2 : \oplus\{\mathsf{some}^0 : A \overset{0}{\otimes} \mathsf{queue}_A[n-1]\}\}\end{aligned}$$

For each insertion, the ins label along with the element travels to the end of the queue. There, it spawns a new *elem* process that holds the inserted element. Hence, the resource cost of each insertion is $2n$ where n is the size of the queue. On the other hand, deletion is similar to that of stack and has a resource cost of 2. Again, this is reflected in the queue_A type, where ins and del are annotated with $2n$ and 2 units of potential respectively.

The resource-aware types show that stacks are more efficient than queues. In particular, the label ins is annotated by 0 for stack_A and with $2n$ for queue_A. Hence, an efficiency comparison can be performed by simply observing the resource-aware session types without needing access to the implementation. The implementation and resource-aware type derivation for *elem* and *empty* can be found in a companion technical report [12].

Queues as two stacks In a functional language, a queue is often implemented with two stacks. The idea is to enqueue into the first stack and to dequeue from the second stack. If the second stack is empty then we copy the first stack over, thereby reversing its order. Since the cost of the dequeue operation can vary drastically between successive dequeues, 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. The type for such a queue implemented as two stacks 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 stack. While deletion, if the second stack is empty, then this stored potential in the first stack is used to pay for copying the elements over to the second stack. 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 store_A. Our type system uses 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 at termination.

We provide a general mechanism for implementing clients for a generic store (see [12] for more details). We define a generic store_A type where the potential annotations are arbitrary natural numbers (or functions of internal measure n).

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

A client of the store_A interface is defined by a list ℓ of ins and del messages that it sends to the store. We index the client $C_{\ell,n}$ by ℓ and the internal measure n of $\mathsf{store}_A[n]$. 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. 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 within the store. Hence, $C_{[],n}$ needs 0 potential. For the potential function, this means $\phi([], n) = 0$. Next, consider the client when the head of the list ℓ is ins. The client sends an ins label followed by an element x of type A. 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, consider the client when the head of the list ℓ is del. The client sends the del label and then case analyzes on the label it receives. If it receives the some label, it receives the element and then continues with $C_{\ell,n-1}$, else it receives the 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 ℓ and chaining the potential equations together, $\phi(\ell, n)$ achieves a resource bound on the client $C_{\ell,n}$.

The stack_A and queue_A interface types are specific instantiations of the store_A type. The derivation for a client of the stack_A interface defines the following potential equations.

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

Similarly, considering the queue type as another instantiation defines the following potential equations (again $\phi([], n) = 0$)

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

This allows us to compare arbitrary clients of two interfaces and compare their resource cost. The resource-aware types are expressive enough to obtain these resource bounds without referencing 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.

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. Both clients send the same number of insertions and deletions. However, their resource cost are completely different. Solving the above system of equations, we obtain $\phi(\ell_1, n) = 2mn + m(m-1) + 2m$, 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.

8 Related Work

Session types were introduced by Honda [24]. The technical development in this work is based on previous work in [30, 35]. 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 we use are inspired by [17]. In contrast to our work, the aforementioned articles do not discuss resource analysis.

In the context of process calculi, capabilities [34] and static analyses [26] have been used to statically restrict communication for controlling buffer sizes in languages without session types. For session-typed communication, upper bounding the size of message queues is simpler and studied in the compiler for Concurrent C0 [36]. In contrast to capabilities, our potential annotations do not control buffer sizes but provide a symbolic description of the number of messages exchanged at runtime. It is not clear how capabilities could be used to perform such an analysis.

Type systems for static resource bound analysis for sequential programs have been extensively studied (e.g., [11, 27]). 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 [21] and polynomial bounds [19] for sequential functional programs. It can also be integrated with program logics to derive bounds for imperative programs [5, 9]. Moreover, it has been used to derive bounds for term-rewrite systems [23] and object-oriented programs [22]. A recent work also considers bounds on the parallel evaluation cost (also called *span*) of functional programs [20]. 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 exchanged along channels. As a result, the formulation and proof of the soundness theorem is quite different from the soundness of sequential AARA systems.

We are only aware of a few other works that study resource bounds for concurrent programs. Gimenez et al. [15] 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. [2, 3] 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. In a companion paper [13], we complement the present system by analyzing the *parallel* complexity of session-typed programs by extending the basic session types with modalities inspired from linear-time temporal logic.

9 Conclusion and Future Work

We have introduced resource-aware session types, a linear type system that combines session types [24, 30] and type-based amortized resource analysis [19, 21] 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.

An important next step is designing an efficient implementation of Resource-Aware SILL with support for automatic inference of work bounds. 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 [19, 21]. To this end, we are working on an algorithmic version of the declarative type system presented here.

Inferring work bounds has several applications. One direction we plan to explore is the use of resource bounds in process scheduling. For instance, oracle schedulers [1] can use a priori knowledge of the runtime of each parallel thread to calculate thread creation overheads and enhance efficiency.

Acknowledgements This article is based on research that has been supported, in part, by AFRL under DARPA STAC award FA8750-15-C-0082 and in part by the NSF Foundation under Grant No. 1718267. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] Umut A Acar, Arthur Charguéraud, and Mike Rainey. 2016. Oracle-Guided Scheduling for Controlling Granularity in Implicitly Parallel Languages. *J. Funct. Programming* (2016).

[2] Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. 2015. Resource Analysis: From Sequential to Concurrent and Distributed Programs. In *FM'15*.

[3] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. 2016. May-Happen-in-Parallel Analysis for Actor-Based Concurrency. *ACM Trans. Comput. Log.* (2016).

[4] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition Instead of Self-composition for Proving the Absence of Timing Channels. In *PLDI'17*.

[5] Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *ESOP'10*.

[6] Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. In *ICFP'17*.

[7] Richard P Brent. 2013. *Algorithms for minimization without derivatives.* Courier Corporation.

[8] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR'10*.

[9] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *CAV'17*.

[10] Iliano Cervesato and Andre Scedrov. 2009. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information and Computation* (2009).

[11] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *ESOP'15*.

[12] A. Das, J. Hoffmann, and F. Pfenning. 2017. Work Analysis with Resource-Aware Session Types. *ArXiv e-prints* (Dec. 2017). arXiv:cs.PL/1712.08310

[13] A. Das, J. Hoffmann, and F. Pfenning. 2018. Parallel Complexity Analysis with Temporal Session Types. *ArXiv e-prints* (April 2018). arXiv:cs.PL/1804.06013

[14] Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the π-Calculus. *Acta Informatica* (2005).

[15] Stéphane Gimenez and Georg Moser. 2016. The Complexity of Interaction. In *POPL'16*.

[16] Jean-Yves Girard. 1987. Linear logic. *Theor. Comp. Sc.* (1987).

[17] Dennis Griffith and Elsa L. Gunter. 2013. Liquid Pi: Inferrable Dependent Session Types. In *NASA Formal Methods Symp.'13*.

[18] Rémy Haemmerlé, Pedro López-García, Umer Liqat, Maximiliano Klemen, John P. Gallagher, and Manuel V. Hermenegildo. 2016. A Transformational Approach to Parametric Accumulated-Cost Static Profiling. In *FLOPS'16*.

[19] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

[20] Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *ESOP'15*.

[21] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03*.

[22] Martin Hofmann and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *ESOP'06*.

[23] Martin Hofmann and Georg Moser. 2015. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *TLCA'15*.

[24] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*.

[25] Yu Feng Jia Chen and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *CCS'17*.

[26] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. 1995. Static analysis of communication for asynchronous concurrent programming languages. In *SAS'95*.

[27] Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *POPL'13*.

[28] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *S&P'17*.

[29] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *PLDI'15*.

[30] Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *FOSSACS'15*.

[31] Frank Pfenning and Robert J. Simmons. 2009. Substructural Operational Semantics As Ordered Logic Programming. In *LICS '09*.

[32] Miguel Silva, Mário Florido, and Frank Pfenning. 2016. Non-Blocking Concurrent Imperative Programming with Session Types. In *LINEARITY'16*.

[33] Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Alg. Disc. Methods* (1985).

[34] Tachio Terauchi and Adam Megacz. 2008. Inferring Channel Buffer Bounds Via Linear Programming. In *ESOP'08*.

[35] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP'13*.

[36] Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2016. Design and Implementation of Concurrent C0. In *LINEARITY'16*.