CARNEGIE MELLON UNIVERSITY

Resource-Aware Session Types for Digital Contracts

Thesis Proposal

by

Ankush Das

Thesis proposal submitted in partial fulfillment
for the degree of Doctor of Philosophy

Thesis committee:
Jan Hoffmann (chair)
Frank Pfenning
Bryan Parno
Andrew Miller (UIUC)
Shaz Qadeer (Facebook)

January 2019

Abstract

Digital contracts are protocols that describe and enforce execution of a contract, often among mutually distrusting parties. Programming digital contracts comes with its unique challenges, which include describing and enforcing protocols of interaction, analyzing resource usage and tracking linear assets and accounting for fault tolerance. This thesis designs programming languages and techniques for implementation and analysis of digital contracts. The core of the language is based on resource-aware session types. The thesis first introduces resource-aware session types after providing a brief background on session types and resource analysis. It also formalizes sequential and parallel complexity analysis of concurrent programs using resource-aware session types. Second, the thesis describes how these techniques can be applied to the design of Nomos, a domain-specific language for implementation of digital contracts. Nomos uses shared session types to ensure clients interact with contracts in mutual exclusion, tracks assets using a linear type system and derives bounds on resource usage. As proposed work, I plan to design an efficient implementation of Nomos to evaluate its usability. Furthermore, I would like to add support for fault tolerance such that Nomos contracts can interact with clients implemented in a different language, thus providing safety guarantees in the presence of potentially malicious clients.

Contents

Chapter 1

Introduction

Digital contracts are computer protocols that describe and enforce the execution of a contract. With the rise of blockchains and cryptocurrencies such as Bitcoin [81], Ethereum [106], and Tezos [50], digital contracts have become popular in the form of *smart contracts*, which provide potentially distrusting parties with programmable money and an enforcement mechanism that does not rely on third parties. Smart contracts have been used to implement auctions [1], investment instruments [79], insurance agreements [67], supply chain management [75], and mortgage loans [78]. In general, digital contracts hold the promise to reduce friction, lower cost, and broaden access to financial infrastructure.

Smart contracts are implemented using a high-level programming language such as Solidity [36], Rholang [5] and Liquidity [3]. It is then compiled down to bytecode and executed using a runtime environment (e.g. Ethereum Virtual Machine for the Ethereum blockchain). However, these languages have significant shortcomings as they do not accommodate the domain-specific requirements of digital contracts.

- Instead of centering contracts on their interactions with users, the high-level protocol of the intended interactions with a contract is buried in the implementation code, hampering understanding, formal reasoning, and trust.
- Resource (or *gas*) usage of digital contracts is of particular importance for transparency and consensus. However, obliviousness of resource usage in existing contract languages makes it hard to predict the cost of executing a contract and prevent denial-of-service vulnerabilities.
- Existing languages fail to enforce linearity of assets, endangering the validity of a contract when assets are duplicated or deleted, accidentally or maliciously [77].

Such limitations of the contract languages can lead to vulnerabilities in the implemented contracts which can be exploited by malicious users having direct financial consequences. A well-known example is the attack on The DAO [79], resulting in a multi-million dollar theft by exploiting a contract vulnerability. Maybe even more important is the potential erosion of trust as a result of such failures.

Resource-Aware Session Types Data types remain one of the most effective ideas in programming languages. Types act as a fundamental unit of composition, and play a pivotal role in many aspects of software. However, even today, programming systems model communications using protocols which are not directly supported by type systems. Session types are a type discipline for communication-centric programming, that is based on message-passing programs that interact via *channels*. The channels are typed providing a protocol for communication taking place on the channel. Type checking guarantees that the processes obey the protocol. For example, a channel of type int list will either send a cons message, followed by an integer and recurse, or will send a nil message, and terminate. The type system also guarantees deadlock freedom (also called *progress*) and session fidelity (also called *preservation*). Since there is no shared memory, such programs cannot suffer from race conditions either. Thus, session types provide a powerful language for safe concurrent programming.

An important aspect of session types is that they can be enhanced to account for resource usage. For session-typed programs, and concurrent programs in general, two main resources of interest are *work* and *span*. Work, or sequential complexity refers to the total number of steps taken by each thread/process in the program, while span, or parallel complexity refers to the earliest time in which the computations can finish, when provided with an unlimited number of processors. In prior work, I have analyzed both work [38] and span [39] of session-typed programs.

Work analysis is based on a linear type system that combines standard session types with type-based amortized analysis. Each session type constructor is decorated with a natural number declaring the *potential* that must be transferred along with the corresponding message. This potential (in the sense of classical amortized analysis [97]) may either be spent by sending other messages in the process network, or stored in a process for future interactions. Since the process interface is characterized entirely by the resource-aware session type of the channels it interacts with, this design provides a compositional resource specification. 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 on a channel. As a result, the type system derives parametric bounds on the resource usage of message-passing processes. Finally, a type safety theorem proves that the derived bounds are sound with respect to an operational cost semantics that tracks the total number of messages exchanged in a network of communicating processes.

In addition to work, the timing of messages is of central interest for analyzing parallel cost. We developed a type system that captures the parallel complexity of session-typed programs by adding *temporal modalities next* ($\bigcirc A$), *always* ($\Box A$) and *eventually* ($\Diamond A$), interpreted over a linear model of time. When considered as types, the temporal modalities allow us to express properties of concurrent programs such as the *message rate* of a stream, the *latency* of a pipeline, the *response time* of a concurrent data structure, or the *span* of a fork-join parallel computation, all in the same uniform manner. The circle operator ($\bigcirc$) expresses precision of message timings, while the box ($\Box$) and diamond operators ($\Diamond$) provide flexibility, allowing us

to express the timing of a wide variety of standard session-typed programs. Finally, a type safety theorem establishes that the message timing expressed by the type system are realized by the timed operational semantics.

Programming Digital Contracts with Resource-Aware Session Types Recently, I have designed the type-theoretic foundations of Nomos [40], a programming language whose genetics match directly with the domain-specific requirements to provide strong static guarantees that facilitate the design of correct contracts. To express and enforce the protocols underlying a contract, we base Nomos on resource-aware session types. Type checking can be automated and guarantees that Nomos programs follow the communication protocol expressed by the type.

Resource-aware types also make transaction cost in Nomos predictable and transparent, and prevents bugs resulting from excessive resource usage. Since resource-aware session types are parametric in the cost model, they can be instantiated to derive the gas bounds for Nomos programs. The type soundness theorem for Nomos guarantees that these bounds are both sound and precise. Other advantages of this type-based resource analysis are natural compositionality and reduction of bound inference to off-the-shelf LP solving.

To eliminate a class of bugs in which the internal state of a contract loses track of its assets, Nomos integrates a linear type system into a functional language. Linear type systems use the ideas of Girard's linear logic [48] to represent certain resources ensuring they are never discarded or duplicated. Assets such as money and other commodities that can be exchanged between parties in a contract are typed using a linear channel. Type safety guarantees that processes maintain proper ownership of linear assets and do not terminate while holding access to a linear asset.

Since there exist multiple clients of a contract, we use a *shared* session type [19] to define the protocol of a contract. This ensures that clients interact with a contract in *mutual exclusion*. The type clearly demarcates the parts of the protocol that become a critical section using $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ modality marking its start and $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ modality marking its end. Programmatically, $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ translates into an acquire of the contract, while $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ into its release. In summary, Nomos goes beyond smart contracts and can be used to implement general digital contracts.

1.1 Proposed Work

The most important aspects of a programming language are safety, usability and efficiency. I have already proved a type soundness theorem, including a proof of progress and type preservation. However, for a usability evaluation, we need an efficient implementation for Nomos. I plan to first *design an implementation of resource-aware session types, and then extend it to Nomos*. There are interesting design decisions to consider in the implementation. First, resource bounds for several standard session-typed programs depend on type refinements. For

instance, work analysis for enqueue and dequeue operations in a queue may depend on its size. Similarly, timing analysis for appending one list to another depends on the size of the former list. However, refinement and general dependent types complicate the implementation. Defining subtyping and type equality, both declaratively and algorithmically, is a serious challenge. Second, manual analysis of resource usage of Nomos contracts can be tedious. To this end, we are considering adding support for automatic inference of resource usage for contracts. Third, Nomos currently has no notion of time preventing implementation of contracts depending on time. For instance, a standard auction or lottery contract might be required to be open for a specified period of time. Since temporal session types have already been explored, we are considering if we can augment Nomos with temporal session types without significantly increasing the programmer burden. Finally, to assist smart contract developers, I plan to develop a more intuitive surface syntax. This might include designing specific libraries for handling linear resources and using type safe dictionaries.

With an implementation, I will manually translate standard contract programs from existing smart contract languages such as Solidity to Nomos. This will help me evaluate Nomos' usability and compare it with other state-of-the-art languages. I would also like to compile contract programs down to bytecode and deploy it on the blockchain. This way, I will be able to confirm whether the gas bounds derived from Nomos' type system matches its actual gas cost at runtime. To guarantee type safety, Nomos also currently does not support direct contract-to-contract interaction. I would like to evaluate how severe this limitation is, and if this restriction can be alleviated without compromising type safety.

In summary, the main implementation challenges that I propose to address for resource-aware session types are

- Type dependencies that are necessary to express resource usage (Section 6.1)
- Type constraints that are needed to ensure validity of refinement types (Section 6.2)
- Type equality in the presence of type dependencies (Section 6.3)
- Work and time reconstruction (Section 6.4)

The crucial steps that I plan to explore with the implementation of Nomos are (Section 6.5)

- Integration with a resource-aware functional layer
- Adding support for shared channels
- Implementation of standard smart contracts in Nomos

Further Proposed Work The safety guarantees of Nomos only hold when all processes deployed on the blockchain are well-typed. However, this is a severe restriction considering the wide range of existing languages that can be used to implement and deploy contracts. This allows malicious users to interact with Nomos contracts through untyped or ill-typed processes. However, Nomos currently does not provide any error handling features. Existing smart contract languages such as Solidity use state-reverting exceptions to handle errors. Such an exception will undo all changes made to the state by the current call. Solidity also provides a require function to ensure valid conditions on the input arguments and the contract state, thus establishing pre-conditions. However, it provides no mechanism to handle exceptions, except for reverting the contract state. More importantly, reverting the state does not recover the linear resources that were exchanged during the partial transaction execution.

Error handling and exceptions is an active area of research with regard to session types [30, 44, 80]. However, none of the existing techniques have explored state reversion. Moreover, exception handling is especially challenging in the presence of linear assets as they cannot be discarded. I am considering techniques from runtime monitoring [68] to first detect violation of the protocol. Once a violation is detected, I am considering adding support for exception handling to Nomos without compromising type safety and linearity. The most important requirement here is that the contract must be reverted to a well-formed state when a partial transaction is aborted. This will allow the contract to interact with future clients in a well-formed manner. Thus, we will be able to protect Nomos contracts against malicious clients, and also prevent denial-of-service attacks. This work will follow my work on implementation.

Safety guarantees are critical for smart contract languages. Any source of vulnerability can potentially be exploited by malicious users leading to loss of individual funds. I believe resource-aware session types can provide many useful safety guarantees, without increasing programmer burden significantly. Beyond smart contracts, resource-aware session types provide resource usage information of programs statically, which can be used to design efficient scheduling techniques for distributed systems. Resource cost of concurrent programs is an important problem and resource-aware session types go a long way in addressing it.

1.2 Overview

Thesis Statement *Session types serve as a sound and practical type-theoretic foundation for digital contracts providing strong domain specific guarantees while simplifying programming.*

Chapter 2 provides necessary background for understanding this thesis. It first provides the fundamentals of vanilla session types, along with their formalization. Then, it explains the technique of type-based automatic amortized resource analysis in the context of a functional programming language.

Chapter 3 describes *work analysis* of a session-typed process. Work (or sequential complexity) of a process is defined as the total number of operations executed by it. The chapter introduces

a novel resource-aware session type system that augments simple session types with new type constructors that measure the work performed by a process.

Chapter 4 presents *time analysis* of session-typed processes. The type system introduces type constructors inspired from linear temporal logic to the simple session type system that measure the time of a process execution. This accounts for the parallel complexity of a process.

Chapter 5 designs a new language, named Nomos, based on shared and resource-aware session types that can be used to implement digital contracts. The language handles assets using a linear type system, eliminates the re-entrancy problem by design and provides multi-user support. Resource-aware types also enable the programmer to statically understand the resource usage of contracts. The language is proved to satisfy type preservation (session fidelity) and a limited form of progress (deadlock freedom).

Chapter 6 is the main topic of the proposal. It focuses on implementing resource-aware session types, and extending it to Nomos. Moreover, there are other directions that I would like to explore (time permitting) beyond the scope of the thesis. These have been summarized in Section 6.6.

Finally, Chapter 7 discusses further proposed work that I will address if time permits. This mainly includes improving the robustness of Nomos, in particular enhancing it with error handling features by integrating it with runtime monitors. These monitors observe the communication behavior and protect contracts against malicious ill-typed clients.

Chapter 2

Background

This chapter introduces two central concepts of this thesis, namely session types and resource analysis. The first half of the chapter focuses on session types, defining them formally with static and dynamic semantics. The second half introduces type-based amortized analysis for functional programming languages. Resource-aware session types formalize the combination of these two techniques.

2.1 Session Types

Session types are a type discipline for communication-centric programming based on message passing via channels. Session-typed channels describe and enforce the protocol of communication among processes. The base system of session types is derived from a Curry-Howard interpretation of intuitionistic linear logic [28]. This chapter focuses on the linear fragment of SILL [86] that internalizes session-based concurrency. Session types were introduced by Honda [64].

Linear logic [48] is a substructural logic that enjoys *exchange* as its only structural property, i.e., it does not exhibit *weakening* or *contraction*. As a result, purely linear propositions can be viewed as resources that must be used *exactly once* in a proof. Here, I adopt the intuitionistic version of linear logic, yielding the following sequent

$$A_1, \ldots, A_n \vdash C$$

where $A_1, \ldots, A_n$ are linear antecedents, while C is the linear 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, a process term P is assigned to the above judgment and each hypothesis

as well as the conclusion is labeled with a *channel*:

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

The resulting judgment states that process P *provides* a service of session type A along channel z, *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 until they terminate. For the judgment to be well-formed, all channel names have to be distinct. The antecedents are often abbreviated to Δ.

The balance between providing and using a session is established by the two fundamental rules of the sequent calculus that are independent of all logical connectives: *cut* and *identity*. Cut states that if P provides service A along channel x, then Q can use the service along the same channel at the same type. Identity states that a client of service A can directly provide A.

$$\frac{\Delta_1 \vdash P_x :: (x : A) \qquad \Delta_2, x : A \vdash Q_x :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P_x \ ; \ Q_x :: (z : C)}\ \mathsf{cut} \qquad \frac{}{y : A \vdash x \leftarrow y :: (x : A)}\ \mathsf{id}$$

Operationally, the process $x \leftarrow P_x \ ; \ Q_x$ creates a globally fresh channel c, spawns a new process $[c/x]P_x$ providing along c, and continues as $[c/x]Q_x$. Conversely, the process $c \leftarrow d$ *forwards* any message M that arrives along d to c and vice-versa. Because channels are used linearly, the forwarding process can then terminate, making sure to apply proper renaming.

The operational semantics are formalized as a system of *multiset rewriting rules* [34]. I introduce semantic objects $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$ describing process P (or message M) providing service along channel c. Remarkably, in this formulation, a message is just a particular form of process, thereby not requiring any special rules for typing; it can be typed just as processes. The semantics rules for cut and id are presented below.

$$\begin{array}{ll} (\mathsf{cut}C) & \mathsf{proc}(d, x \leftarrow P_x \ ; \ Q_x) \ \mapsto \ \mathsf{proc}(c, [c/x]P_x), \mathsf{proc}(d, [c/x]Q_x) \quad (c \text{ fresh}) \\ (\mathsf{id}^+C) & \mathsf{msg}(d, M), \mathsf{proc}(c, c \leftarrow d) \ \mapsto \ \mathsf{msg}(c, [c/d]M) \\ (\mathsf{id}^-C) & \mathsf{proc}(c, c \leftarrow d), \mathsf{msg}(e, M(c)) \ \mapsto \ \mathsf{msg}(e, [d/c]M(c)) \end{array}$$

Here, I adopt the convention to use x, y and z for channel *variables* and c, d and e for *channels*. Channels are created at runtime and substituted for channel variables in process terms. In the last rule, $M(c)$ indicates that c must occur in M, implying it is the sole client of c.

The Curry-Howard correspondence gives each linear logic connective an interpretation as a session type. This session type prescribes the kind of message that must be sent or received along a channel of this type. Table 2.1 summarizes the description of the type along with the provider action. I follow a detailed description of each session type operator.

Type	Provider Action	Session Continuation
$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send label $k \in L$	A_k
$\&\{\ell : A_\ell\}_{\ell \in L}$	receive and branch on label $k \in L$	A_k
$\mathbf{1}$	send token close	*none*
$A \otimes B$	send channel $c : A$	B
$A \multimap B$	receive channel $c : A$	B

TABLE 2.1: Basic Session Types. Every provider action has a matching client action.

Internal Choice

A type A is said to describe a *session*, which is a particular sequence of interactions. As a first type construct, consider *internal choice* $\oplus\{\ell : A_\ell\}_{\ell \in L}$, an n-ary labeled generalization of the linear logic connective $A \oplus B$. A process that provides $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ can send any label $k \in L$ along x and then continue by providing $x : A_k$. The corresponding process is written as $(x.k\ ;\ P)$, where P is the continuation that provides A_k. This typing is formalized by the *right rule* $\oplus R$ in linear sequent calculus. The corresponding client branches on the label received along x as specified by the *left rule* $\oplus L$.

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

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

Operationally, since communication is asynchronous, the process $(c.k\ ;\ P)$ sends a message k along c and continues as P without waiting for it to be received. As a technical device to ensure that consecutive messages on a channel arrive in order, the sender also creates a fresh continuation channel c' so that the message k is actually represented as $(c.k\ ;\ c \leftarrow c')$ (read: send k along c and continue as c'). The provider also substitutes c' for c enforcing that the next message is sent on c'.

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

When the message k is received along c, the client selects branch k and also substitutes the continuation channel c' for c, thereby ensuring that it receives the next message on c'. This implicit substitution of the continuation channel ensures the ordering of the messages.

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

External Choice

The dual of internal choice is *external choice* $\&\{\ell : A_\ell\}_{\ell \in L}$, which is the n-ary labeled generalization of the linear logic connective $A \,\&\, B$. This dual operator simply reverses the role of the provider and client. The provider process of $x : \&\{\ell : A_\ell\}_{\ell \in L}$ branches on receiving a label $k \in L$ (described in $\&R$), while the client sends this label (described in $\&L$).

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

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

The operational semantics rules are just the inverse of internal choice. The provider receives the branching label k sent by the provider. Both processes perform appropriate substitutions to ensure the order of messages sent and received is preserved.

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

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

Higher-Order Channels

Session types allow channels to be *higher-order*, i.e., channels can be exchanged over channels. The session type corresponding to the linear logic connective $A \otimes B$ allows its provider to send a channel of type A and then continue with providing B. The corresponding process term $(\mathsf{send}\ x\ w\ ;\ P)$ describes sending channel w over channel x and continuing with P. This typing is provided by the rule $\otimes R$. The client, on the other hand, receives this channel using the term $(y \leftarrow \mathsf{recv}\ x\ ;\ Q)$and binds it to a channel variable y, as described by $\otimes L$.

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

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

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

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

The lolli ($\multimap$) operator is dual to $\otimes$. The provider and client invert their roles, i.e., the provider of $x : A \multimap B$ receives a channel of type A sent by its client.

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

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

$$(\multimap S) \quad \mathsf{proc}(d, \mathsf{send}\ c\ e\ ;\ Q) \ \mapsto\ \mathsf{msg}(c', \mathsf{send}\ c\ e\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \qquad (c' \text{ fresh})$$
$$(\multimap C) \quad \mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c), \mathsf{msg}(c', \mathsf{send}\ c\ e\ ;\ c' \leftarrow c) \ \mapsto\ \mathsf{proc}(c', [c', d/c, x]P)$$

Termination

The type $\mathbf{1}$, the multiplicative unit of linear logic, represents termination of a process, which (due to linearity) is not allowed to use any channels.

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

Operationally, a client has to wait for the corresponding closing message, which has no continuation since the provider terminates.

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

Process Definitions

Process definitions have the form $\Delta \vdash f = P :: (x : A)$ where f is the name of the process and P its definition. All definitions are collected in a fixed global signature Σ. Also, since process definitions are mutually recursive, it is required that for every process in the signature is well-typed w.r.t. Σ, i.e. $\Sigma\ ;\ \Delta \vdash P :: (x : A)$. For readability of the examples, I break a definition into two declarations, one providing the type and the other the process definition binding the variables x and those in Ω (generally omitting their types):

$$\Delta \vdash f :: (x : A)$$
$$x \leftarrow f \leftarrow \Delta = P$$

A new instance of a defined process f can be spawned with the expression

$$x \leftarrow f \leftarrow \overline{y}\ ;\ Q$$

where $\overline{y}$ is a sequence of variables matching the antecedents Δ. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q. The operational semantics

reduces the spawn to a cut.

$$(\mathsf{def}C) \quad \mathsf{proc}(c, x \leftarrow f \leftarrow \overline{e}\ ;\ Q) \ \mapsto\ \mathsf{proc}(a, [a/x, \overline{e}/\Delta]P), \mathsf{proc}(c, [a/x]Q) \quad (a \text{ fresh})$$

where $\Delta \vdash f = P :: (x : A) \in \Sigma$. Here I write $\overline{e}/\Delta$ to denote substitution of the channels in $\overline{e}$ for the corresponding variables in Δ.

Sometimes a process invocation is a *tail call*, written without a continuation as $x \leftarrow f \leftarrow \overline{y}$. This is a short-hand for $x' \leftarrow f \leftarrow \overline{y}\ ;\ x \leftarrow x'$ for a fresh variable x', that is, a fresh channel is created and immediately identified with x (although it is generally implemented more efficiently).

Recursive Types

Session types can be naturally extended to include recursive types. For this purpose I allow (possibly mutually recursive) type definitions $X = A$ in the signature, where I require A to be *contractive* [45]. This means here that A should not itself be a type name. The type definitions are *equi-recursive* so X can be silently replaced by A during type checking, and no explicit rules for recursive types are needed.

2.1.1 Examples

As a first example, consider a stream of bits defined recursively as

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$$

When considering bits as representing natural numbers, the least significant bit is sent first. For example, a process *six* sending the number $6 = (110)_2$ would be

$$\cdot \vdash \mathit{six} :: (x : \mathsf{bits})$$
$$x \leftarrow \mathit{six} = x.\mathsf{b0}\ ;\ x.\mathsf{b1}\ ;\ x.\mathsf{b1}\ ;\ x.\$\ ;\ \mathsf{close}\ x$$

Executing $\mathsf{proc}(c_0, c_0 \leftarrow \mathit{six})$ yields (with some fresh channels $c_1, \ldots, c_4$)

$$\begin{aligned} \mathsf{proc}(c_0, c_0 \leftarrow \mathit{six}) \ \mapsto^*\ & \mathsf{msg}(c_4, \mathsf{close}\ c_4), \\ & \mathsf{msg}(c_3, c_3.\$\ ;\ c_3 \leftarrow c_4), \\ & \mathsf{msg}(c_2, c_2.\mathsf{b1}\ ;\ c_2 \leftarrow c_3), \\ & \mathsf{msg}(c_1, c_1.\mathsf{b1}\ ;\ c_1 \leftarrow c_2), \\ & \mathsf{msg}(c_0, c_0.\mathsf{b0}\ ;\ c_0 \leftarrow c_1), \end{aligned}$$

As a first example of a recursive process definition, consider one that just copies the incoming bits on to the outgoing bits.

$y : \mathsf{bits} \vdash \mathit{copy} :: (x : \mathsf{bits})$
$x \leftarrow \mathit{copy} \leftarrow y =$
  $\mathsf{case}\ y\ (\mathsf{b0} \Rightarrow x.\mathsf{b0}\ ;\ x \leftarrow \mathit{copy} \leftarrow y$ % received b0 on y, send b0 on x, recurse
    $\mid \mathsf{b1} \Rightarrow x.\mathsf{b1}\ ;\ x \leftarrow \mathit{copy} \leftarrow y$ % received b1 on y, send b1 on x, recurse
    $\mid \$ \Rightarrow x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{close}\ x)$ % received \$ on y, send \$ on x, wait on y, close x

Note the occurrence of a (recursive) *tail call* to *copy*.

A last example: to increment a bit stream turn b0 to b1 but then forward the remaining bits unchanged ($x \leftarrow y$), or turn b1 to b0 but then increment the remaining stream ($x \leftarrow \mathit{plus1} \leftarrow y$) to capture the effect of the carry bit.

$y : \mathsf{bits} \vdash \mathit{plus1} :: (x : \mathsf{bits})$
$x \leftarrow \mathit{plus1} \leftarrow y =$
  $\mathsf{case}\ y\ (\mathsf{b0} \Rightarrow x.\mathsf{b1}\ ;\ x \leftarrow y$
    $\mid \mathsf{b1} \Rightarrow x.\mathsf{b0}\ ;\ x \leftarrow \mathit{plus1} \leftarrow y$
    $\mid \$ \Rightarrow x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{close}\ x)$

2.1.2 Preservation and Progress

The main theorems that exhibit the deep connection between our type system and the operational semantics are the usual *type preservation* and *progress*, sometimes called *session fidelity* and *deadlock freedom*, respectively.

So far, I have only described individual processes. However, processes exist in a *configuration*. A process configuration is a multiset of semantic objects, $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$, where any two offered channels are distinct. A key question is how to type these configurations. Since they consist of both processes and messages, they both *use* and *provide* a collection of channels. And even though a configuration is treated as a multiset, typing imposes a partial order on the processes and messages where a provider of a channel appears to the left of its client.

A configuration is typed w.r.t. a signature providing the type declaration of each process. A signature Σ is *well formed* if (a) every type definition $V = A_V$ is *contractive*, and (b) every process definition $\Delta \vdash f = P :: (x : A)$ in Σ is well typed according to the process typing judgment, i.e. $\Sigma\ ;\ \Delta \vdash P :: (x : A)$.

I use the following judgment to type a configuration.

$$\Sigma\ ;\ \Delta_1 \vDash \mathcal{C} :: \Delta_2$$

It states that Σ is well-formed and that the configuration $\mathcal{C}$ uses the channels in the context Δ_1 and provides the channels in the context Δ_2. The configuration typing judgment is defined

$$\frac{}{\Sigma\ ;\ \Delta \vDash (\cdot) :: \Delta}\ \mathsf{empty} \qquad \frac{\Sigma\ ;\ \Delta_0 \vDash \mathcal{C}_1 :: \Delta_1 \qquad \Sigma\ ;\ \Delta_1 \vDash \mathcal{C}_2 :: \Delta_2}{\Sigma\ ;\ \Delta_0 \vDash (\mathcal{C}_1\ \mathcal{C}_2) :: \Delta_2}\ \mathsf{compose}$$

$$\frac{\Sigma\ ;\ \Delta_1 \vdash P :: (c : A)}{\Sigma\ ;\ \Delta, \Delta_1 \vDash \mathsf{proc}(c, P) :: (\Delta, (c : A))}\ \mathsf{proc} \qquad \frac{\Sigma\ ;\ \Delta_1 \vDash P :: (c : A)}{\Sigma\ ;\ \Delta, \Delta_1 \vDash \mathsf{msg}(c, P) :: (\Delta, (c : A))}\ \mathsf{msg}$$

FIGURE 2.1: Typing rules for a configuration

using the rules presented in Figure 2.1. The rule empty defines that an empty configuration is well-typed. The rule compose composes two configurations $\mathcal{C}_1$ and $\mathcal{C}_2$; $\mathcal{C}_1$ provides service on the channels in Δ_1 while $\mathcal{C}_2$ uses the channels in Δ_2. The proc rule creates a configuration out of a single process. Similarly, the msg rule creates a configuration out of a single message.

Theorem 2.1 (Type Preservation). *If* $\Sigma\ ;\ \Delta' \vDash \mathcal{C} :: \Delta$ *and* $\mathcal{C} \mapsto \mathcal{D}$*, then* $\Sigma\ ;\ \Delta' \vDash \mathcal{D} :: \Delta$.

Proof. By case analysis on the transition rule, applying inversion to the given typing derivation, and then assembling a new derivation of $\mathcal{D}$. □

A process or message is said to be *poised* if it is trying to communicate along the channel that it provides. A poised process is comparable to a value in a sequential language. A configuration is poised if every process or message in the configuration is poised. Conceptually, this implies that the configuration is trying to communicate externally, i.e. along one of the channel it provides. The progress theorem then shows that either a configuration can take a step or it is poised. To prove this I show first that the typing derivation can be rearranged to go strictly from right to left and then proceed by induction over this particular derivation.

Theorem 2.2 (Global Progress). *If* $\cdot \vDash \mathcal{C} :: \Delta$ *then either*

(i) $\mathcal{C} \mapsto \mathcal{D}$ *for some* $\mathcal{D}$*, or*

(ii) $\mathcal{C}$ *is poised.*

Proof. By induction on the right-to-left typing of $\mathcal{C}$ so that either $\mathcal{C}$ is empty (and therefore poised) or $\mathcal{C} = (\mathcal{D}\ \mathsf{proc}(c, P))$ or $\mathcal{C} = (\mathcal{D}\ \mathsf{msg}(c, M))$. By induction hypothesis, $\mathcal{D}$ can either take a step (and then so can $\mathcal{C}$), or $\mathcal{D}$ is poised. In the latter case, I analyze the cases for P and M, applying multiple steps of inversion to show that in each case either $\mathcal{C}$ can take a step or is poised. □

2.2 Resource Analysis

The quality of software crucially depends on the amount of resources – time, memory and energy – that are required for its execution. Statically understanding and controlling resource usage continues to be a central issue in software development. Recent years have

seen fast progress in developing tools and frameworks for statically reasoning about resource usage. The obtained *size change* information forms the basis for the computation of actual bounds on loop iterations and recursion depths; using counter instrumentation [53], ranking functions [9, 15, 26, 96], recurrence relations [10, 11] and abstract interpretation [33, 107]. Automatic resource analysis for functional programs are based on sized types [101], term-rewriting [18] and amortized resource analysis [58, 61, 69, 95].

Automatic amortized resource (AARA) was introduced for a strict first-order functional language with built-in data types [61]. Since then, AARA techniques have been applied to univariate polynomial bounds [56], multivariate bounds [58], higher-order functional programs [69] and user-defined data types [60]. In this section, I will mainly focus on linear bounds for first-order programs as it outlines the main ideas of AARA without complicating the type system.

2.2.1 Manual Amortized Analysis

Often the cost of an operation on a data structure depends on its state. Thus, it is natural to account for the total cost of a sequence of operations on such a data structure. To analyze such a sequence of operations, Sleator and Tarjan [97] proposed amortized analysis with the *potential method*.

The concept of potential is inspired by the notion of potential energy in physics. The idea is to define a potential function $\Phi : \mathcal{D} \to \mathbb{R}^{\geq 0}$ that maps data structure $D \in \mathcal{D}$ to a non-negative number. Operations on the data structure can then *increase* or *decrease* the potential. The amortized cost of an operation $\mathsf{op}(D)$ is then defined as the sum of its actual cost K and the difference of the potential caused by op, i.e., $K + \Phi(\mathsf{op}(D)) - \Phi(D)$. The sum of the amortized costs over a sequence of operations and the initial potential of D then furnishes an upper bound on the actual cost of the sequence.

A standard example that demonstrates the benefits of amortization is the analysis of a functional queue, represented as two lists L_{in} and L_{out}. Enqueuing an element simply adds it to the head of L_{in}, while dequeuing removes the element from the head of L_{out}. If L_{out} is empty, the elements from L_{in} are transferred to L_{out}, thereby reversing the order of the elements.

The cost of a dequeue operation for this queue depends on the state of L_{out}, whether its empty or not. In the worst case, when L_{out} is empty, the cost of dequeue is linear. However, we can introduce a potential $\Phi(L_{in}, L_{out}) = 2\,|L_{in}|$. Then, the amortized cost of enqueue is 3 – one to pay for consing to L_{in}, and two for the increase in potential. More importantly, the amortized cost of dequeue is 1. If L_{out} is not empty, the cost of detach is 1, while there is no change to the potential. While if L_{out} is empty, the potential stored in L_{in} is used to pay for the cost of transfer from L_{in} to L_{out}. Formally, the cost of transfer is $2\,|L_{in}|$, equal to the change in the potential. Hence, the amortized cost of dequeue remains 1, used to pay for the detach from L_{out} after the transfer. Thus, amortized analysis proves that the worst-case cost of both enqueue and dequeue operations is constant.

2.2.2 Automatic Amortized Analysis

The potential method from amortized analysis can be applied to statically analyze functional programs. The key idea here is that the arguments of a function store potential, which is consumed during function evaluation. The initial potential of the arguments therefore equals the sum of the resource cost of the function, and the potential of the return value. Thus, it acts as an upper bound on the resource cost. Since the excess potential is stored in the result, this technique is completely compositional.

Automation is a key requirement here since requiring the programmer to provide the potential functions will significantly increase their burden. To make automation feasible, it is necessary to restrict the space of possible potential functions. There is a precision-scalability trade-off here since increasing the space of potential functions will improve the precision of resource bounds, but will make automation more challenging.

I will restrict the potential functions to be *linear* and the language to be strict, first-order and functional. I attach the potential of the data structure to its type. Then, a sound type checking algorithm statically verifies that the potential is sufficient to pay for all operations that are performed on this data structure during any possible evaluation of the program. Consider the *append* function that takes two lists and appends the second list to the first. The function is implemented as follows.

```
let rec append l1 l2 =
    match l1 with
    | [] -> l2
    | x::xs -> let ys = append xs l2 in x::ys
```

To understand the resource usage for *append*, we first need to fix the resource we are interested in counting. For this example, suppose we count the number of cons (::) operations. The above code suggests that the number of cons operations equals the length of l_1. To understand the type-based analysis, consider the following type for *append*.

$$\mathit{append} : L^1(A) \times L^0(A) \xrightarrow{0/0} L^0(A)$$

Intuitively, this describes that a unit potential is attached to every element in l_1, and no potential on l_2 and the result. In the nil branch of the match, the context is assigned the type $l_1 : L^1(A), l_2 : L^0(A)$. Since l_1 is nil, the total potential of the context is 0, and l_2 is directly returned. In the cons branch, the context is typed as $x : A, xs : L^1(A), l_2 : L^0(A)$. Remarkably, the recursive call to *append* utilizes the same type, since xs and l_2 have the same type as described in the signature. After the recursive call, the context becomes $x : A, ys : L^0(A), l_2 : L^0(A)$, and the unit potential stored in x is used to perform the cons operation.

The type inference algorithm assigns variable potential annotations to *append*.

$$\mathit{append} : L^{q_1}(A) \times L^{q_2}(A) \xrightarrow{r/s} L^{q}(A)$$

The inference algorithm then derives linear constraints on the annotations. For *append*, the constraints generated are $q_1 \geq q_2 + 1$, and $q_2 = q$, and $r \geq s$. These constraints are solved with an off-the-shelf linear-programming solver (LP solver) whose goal is to minimize the initial potential to derive the most precise bound. The LP solver recovers the annotation values described earlier, thus proving the exact bound $|l_1|$ for *append*.

Chapter 3

Work Analysis

This chapter studies the foundations of worst-case resource analysis for session-typed programs. The key idea here is to rely on *resource-aware session types* to describe the resource bounds for inter-process communication. We extend session types to not only exchange messages, but also potential along a channel. The potential (in the sense of classical amortized analysis) 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.

Here, I mainly 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 significant and necessary first step. The derived bounds 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 [25] that relates the complexity of the execution of a program on a k-processor machine to the program's work (this chapter) and span (next chapter).

The analysis is based on a linear type system that extends standard session types with two new type constructors, one to receive potential ($\triangleleft^r$) and one to send potential ($\triangleright^r$). The superscript r declares the amount of potential that must be transferred (conceptually!). Since the interface to a process is characterized entirely by the resource-aware session types of the channels it interacts with, this design provides a compositional resource specification. For closed programs (which evolve into a closed network of interacting processes), the bound becomes a single constant.

A conceptual challenge is to express symbolic bounds in a setting without static data structures and intrinsic sizes. The 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. My 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.

The main contributions are as follows. I present the first session type system for deriving parametric bounds on the resource usage of message-passing processes. I also 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. I also demonstrate the effectiveness of the technique by deriving tight bounds for some standard examples of amortized analysis from the literature on session types. I 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.

3.1 Overview

This section will motivate and informally introduce resource-aware session types and show how they can be used to analyze the resource usage of message-passing processes. I describe an implementation of a counter and use resource-aware session types to analyze its resource usage. Like in the rest of this chapter, the resource we are interested in is the total number of messages sent along all channels in the system.

As a first simple example, I 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 $\$$.

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

For instance, the number $6 = (110)_2$ would be represented by the sequence of messages zero, one, one, $\$$, *close*. A client of a channel $c : \mathsf{bits}$ has to branch on whether it receives zero, one, or $\$$. As a second example, I describe the interface to a counter. A client can repeatedly send inc messages to a counter, until they 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.

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

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. This is observed by introducing a potential of 1 for every bit that is 1 and using this potential to *pay* for the expensive case in which an increment triggers many flips. When the lowest bit is zero, it is flipped to one (costing 1) and a remaining potential

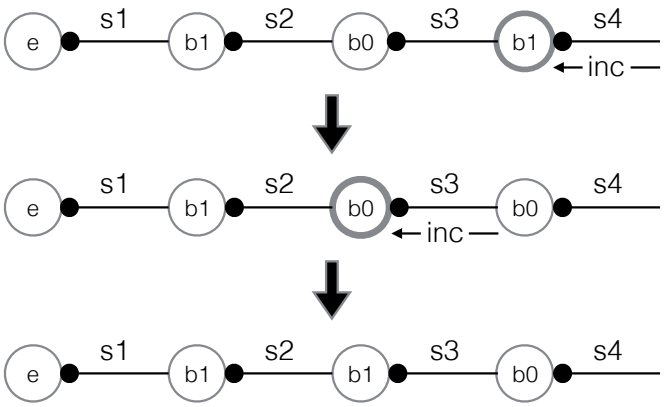

FIGURE 3.1: Binary counter system representing $5 = (101)_2$ with messages triggered when inc message is received on s_4.

of 1 is also stored with this bit. When the lowest bit is one, the stored potential is used 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.

A binary counter is modeled 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 3.1, the process $b0$ offers along channel s_3 (indicated by $\bullet$ between $b0$ and s_3) and uses channel s_2. This is formally written as

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

The *definitions* of e, $b0$, and $b1$ can be found in Figures 3.3 and 3.4. The only channel visible to an outside client (not shown) is s_4. Figure 3.1 shows the messages triggered if an increment message is received along s_4.

Expressing resource bounds. My 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. My 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 the 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 right after the label inc. We do this using the $\triangleleft$ operator indicating the required potential with the superscript, postponing the discussion of val.

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

When types are assigned to processes, we use the more expressive resource-aware session types. We indicate the potential stored in a particular process as a superscript on the turnstile.

$$t : \mathsf{ctr} \quad \vdash^{0} \quad b0 :: (s : \mathsf{ctr}) \tag{3.1}$$

$$t : \mathsf{ctr} \quad \vdash^{1} \quad b1 :: (s : \mathsf{ctr}) \tag{3.2}$$

$$\cdot \quad \vdash^{0} \quad e :: (s : \mathsf{ctr}) \tag{3.3}$$

These typing constraints can be verified using the typing rules of the system, using the definitions of $b0$, $b1$, and e. Informally, the reason is as follows:

$b0$**:** After $b0$ 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 13).

$b1$**:** After $b1$ receives inc it receives 1 unit of potential which, when combined with the stored one, makes 2 units. It sends an inc message which consumes 1 unit, followed by sending a unit potential, thereby consuming the 2 units. 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**:** After 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 $\mathsf{val} : \triangleleft^{?}\mathsf{bits}$ 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. Figure 3.2 demonstrates the messages triggered when val message is received along s_4. The e process will just send \$ and *close*, indicating the empty stream of bits.

There will be enough potential to carry out the required send operations if each process ($b0$, $b1$, and e) carries an additional 2 units of potential. These could be imparted with the inc and val messages by sending 2 more units with inc and 2 units with val. That is, the following type is valid:

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

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, I need to introduce *families of session types*.

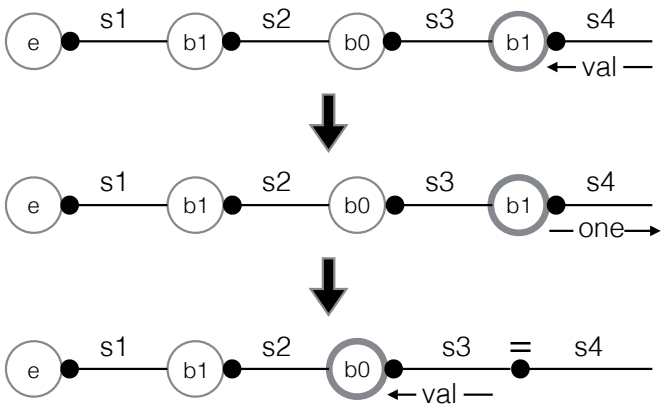

FIGURE 3.2: Binary counter system representing $5 = (101)_2$ with messages triggered when val message is received on s_4.

A more precise analysis. This requires that *in the type* either the number of bits in the representation of a number or its value can be referred. 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. These measures are indicated 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 [52]) with respect to type-checking and inference. Here, it is treated as a meta-level notation to denote families of types. Following the reasoning above, we obtain the following type:

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

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

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

The type system verifies these types against the implementation of $b0$, $b1$, and e (see Figures 3.3 and 3.4, potential annotations marked in red). I will briefly explain the type derivation of $b0$, as shown in Figure 3.3 after the $\%$. After receiving the inc message, the $b0$ process receives a unit potential on s using the $\triangleleft^{1}$ type constructor. This constructor is accompanied by the get construct (line 4) which receives the unit potential which is stored in the process, as indicated by the number on the turnstile. The type on the right exactly matches $b1$'s type in the signature, thereby making the call to $b1$ valid. Similarly, $b0$ receives $2\lceil \log(2n+1)\rceil + 2$ units of potential after receiving the val.

The cost model of interest in this chapter counts the total number of messages exchanged in the system. This is realized formally by consuming a unit of potential before every message sent. The corresponding construct is work $\{1\}$, as indicated in line 7. This consumes a unit

```
1:  (t : ctr[n]) ⊢^0 b0 :: (s : ctr[2n])
2:    s ← b0 ← t =
3:      case s
4:        (inc ⇒ get s {1} ;                          % (t : ctr[n]) ⊢^1 s : ctr[2n + 1]
5:                s ← b1 ← t
6:        | val ⇒ get s {2 ⌈log(2n + 1)⌉ + 2} ;       % (t : ctr[n]) ⊢^{2⌈log(2n+1)⌉+2} s : bits
7:                work {1} ;                          % (t : ctr[n]) ⊢^{2⌈log(2n+1)⌉+2−1} s : bits
8:                s.zero ;                            % (t : ctr[n]) ⊢^{2⌈log(2n+1)⌉+1} s : bits
9:                work {1} ;                          % (t : ctr[n]) ⊢^{2⌈log(2n+1)⌉+1−1} s : bits
10:               t.val ;                             % (t : ◁^{2⌈log(n+1)⌉+2} bits) ⊢^{2⌈log(2n+1)⌉} s : bits
11:               pay t {2 ⌈log(n + 1)⌉ + 2} ;        % (t : bits) ⊢^{2⌈log(2n+1)⌉−2⌈log(n+1)⌉−2} s : bits
12:               s ← t)                              % (t : bits) ⊢^0 s : bits
```

FIGURE 3.3: Implementation for $b0$ process with its type derivation.

of potential, as indicated by the type on the right. A unit potential is similarly consumed on line 9 before sending the val message on t (line 10). The dual to the get construct is pay. This is used to send potential on a channel, as indicated on line 11, consuming potential stored in the process. Finally, the $b0$ process remains with no potential and can successfully terminate by forwarding. Note that a process is not allowed to terminate while it stores potential as that would violate the linearity constraint on the potential. The derivations for $b1$ and e are similar and described in Figure 3.4.

The typing rules reduce the well-typedness of these processes to arithmetic inequalities which can be solved 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.

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

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

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 *close* along c, indicating termination.

3.2 Operational Cost Semantics

The cost semantics for standard session types is augmented to track the total work performed by the system. 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 executes the work $\{c\}$ construct, its counter w is incremented by c. When a process terminates, the respective predicate is removed from the configuration, but its work done is preserved. A

$(t : \mathsf{ctr}[n]) \vdash^{1} b1 :: (s : \mathsf{ctr}[2n+1])$
$s \leftarrow b1 \leftarrow t =$
case s
$(\mathsf{inc} \Rightarrow \mathsf{get}\ s\ \{1\}\ ;$ % $(t : \mathsf{ctr}[n]) \vdash^{2} s : \mathsf{ctr}[2n+2]$
$\mathsf{work}\ \{1\}\ ;$ % $(t : \mathsf{ctr}[n]) \vdash^{2-1} s : \mathsf{ctr}[2n+2]$
$t.\mathsf{inc}\ ;$ % $(t : \triangleleft^{1}\mathsf{ctr}[n+1]) \vdash^{1} s : \mathsf{ctr}[2n+2]$
$\mathsf{pay}\ t\ \{1\}\ ;$ % $(t : \mathsf{ctr}[n+1]) \vdash^{1-1} s : \mathsf{ctr}[2n+2]$
$s \leftarrow b0 \leftarrow t$
$\mid \mathsf{val} \Rightarrow \mathsf{get}\ s\ \{2\lceil\log(2n+2)\rceil+2\}\ ;$ % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+2)\rceil+2} s : \mathsf{bits}$
$\mathsf{work}\ \{1\}\ ;$ % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+2)\rceil+2-1} s : \mathsf{bits}$
$s.\mathsf{one}\ ;$ % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+2)\rceil+1} s : \mathsf{bits}$
$\mathsf{work}\ \{1\}\ ;$ % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+2)\rceil+1-1} s : \mathsf{bits}$
$t.\mathsf{val}\ ;$ % $(t : \triangleleft^{2\lceil\log(n+1)\rceil+2}\mathsf{bits}) \vdash^{2\lceil\log(2n+2)\rceil} s : \mathsf{bits}$
$\mathsf{pay}\ t\ \{2\lceil\log(n+1)\rceil+2\}\ ;$ % $(t : \mathsf{bits}) \vdash^{2\lceil\log(2n+2)\rceil-2\lceil\log(n+1)\rceil-2} s : \mathsf{bits}$
$s \leftarrow t)$ % $(t : \mathsf{bits}) \vdash^{0} s : \mathsf{bits}$

$\cdot \vdash^{0} e :: (s : \mathsf{ctr}[0])$
$s \leftarrow e =$
case s
$(\mathsf{inc} \Rightarrow \mathsf{get}\ s\ \{1\}\ ;$ % $\cdot \vdash^{1} s : \mathsf{ctr}[0+1]$
$t \leftarrow e\ ;$ % $(t : \mathsf{ctr}[0]) \vdash^{1} s : \mathsf{ctr}[1]$
$s \leftarrow b1 \leftarrow t$
$\mid \mathsf{val} \Rightarrow \mathsf{get}\ s\ \{2\lceil\log(0+1)\rceil+2\}\ ;$ % $\cdot \vdash^{2\lceil\log(0+1)\rceil+2} s : \mathsf{bits}$
$\mathsf{work}\ \{1\}\ ;$ % $\cdot \vdash^{2-1} s : \mathsf{bits}$
$s.\$\ ;$ % $\cdot \vdash^{1} s : \mathbf{1}$
$\mathsf{work}\ \{1\}\ ;$ % $\cdot \vdash^{1-1} s : \mathbf{1}$
$\mathsf{close}\ s)$ % $\cdot \vdash^{0} s : \mathbf{1}$

FIGURE 3.4: Implementations for $b1$ and e processes with their type derivations.

process can terminate either by sending a close message, or by forwarding. In either case, the process' work is conveniently preserved in the msg predicate to pass it on to the client process.

The cost semantics is parametric in the cost model. That is, the programmer can specify the resource they intend to measure. This is realized by the cost model by inserting a work construct before the respective expressions. For instance, inserting a work $\{1\}$ before each send will count the total number of messages exchanged.

The semantics is defined in Figure 3.5. Each rule consumes the propositions to the left of $\mapsto$ and produces the proposition to its right. The rules cut$C$ and def$C$ spawn a new process with 0 work (as it has not done any work so far), while Q_c continues with the same amount of work. A forwarding process transfers its work to a corresponding message and terminates after identifying the channels, as described in rules id^{+}C and id^{-}C. All other communication rules create a message with work 0, which is then later received by its recipient, thereby transferring

$$
\begin{array}{lll}
(\mathsf{cut}C) & \mathsf{proc}(d, w, x \leftarrow P_x\ ;\ Q_x) \ \mapsto\ \mathsf{proc}(c, 0, [c/x]P_x), \mathsf{proc}(d, w, [c/x]Q_x) & (c \text{ fresh}) \\
(\mathsf{def}C) & \mathsf{proc}(d, w, x \leftarrow f \leftarrow \overline{e}\ ;\ Q) \ \mapsto & \\
 & \mathsf{proc}(c, 0, [c/x, \overline{e}/\Delta]P), \mathsf{proc}(d, w, [c/x]Q) & (c \text{ fresh}) \\
(\mathsf{id}^+C) & \mathsf{msg}(d, w, M), \mathsf{proc}(c, w', c \leftarrow d) \ \mapsto\ \mathsf{msg}(c, w + w', [c/d]M) & \\
(\mathsf{id}^-C) & \mathsf{proc}(c, w, c \leftarrow d), \mathsf{msg}(e, w', M(c)) \ \mapsto\ \mathsf{msg}(e, w + w', [d/c]M(c)) & \\
(\oplus S) & \mathsf{proc}(c, w, c.k\ ;\ P) \ \mapsto\ \mathsf{proc}(c', w, [c'/c]P), \mathsf{msg}(c, 0, c.k\ ;\ c \leftarrow c') & (c' \text{ fresh}) \\
(\oplus C) & \mathsf{msg}(c, w, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, w', \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \ \mapsto & \\
 & \mathsf{proc}(d, w + w', [c'/c]Q_k) & \\
(\& S) & \mathsf{proc}(d, w, c.k\ ;\ Q) \ \mapsto\ \mathsf{msg}(c', 0, c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, w, [c'/c]Q) & (c' \text{ fresh}) \\
(\& C) & \mathsf{proc}(c, w, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}), \mathsf{msg}(c', w', c.k\ ;\ c' \leftarrow c) \ \mapsto & \\
 & \mathsf{proc}(c', w + w', [c'/c]Q_k) & \\
(\otimes S) & \mathsf{proc}(c, w, \mathsf{send}\ c\ e\ ;\ P) \ \mapsto & \\
 & \mathsf{proc}(c', w, [c'/c]P), \mathsf{msg}(c, 0, \mathsf{send}\ c\ e\ ;\ c \leftarrow c') & (c' \text{ fresh}) \\
(\otimes C) & \mathsf{msg}(c, w, \mathsf{send}\ c\ e\ ;\ c \leftarrow c'), \mathsf{proc}(d, w', x \leftarrow \mathsf{recv}\ c\ ;\ Q) \ \mapsto & \\
 & \mathsf{proc}(d, w + w', [c', e/c, x]Q) & \\
(\multimap S) & \mathsf{proc}(d, w, \mathsf{send}\ c\ e\ ;\ Q) \ \mapsto & \\
 & \mathsf{msg}(c', 0, \mathsf{send}\ c\ e\ ;\ c' \leftarrow c), \mathsf{proc}(d, w, [c'/c]Q) & (c' \text{ fresh}) \\
(\multimap C) & \mathsf{proc}(c, w, x \leftarrow \mathsf{recv}\ c), \mathsf{msg}(c', w', \mathsf{send}\ c\ e\ ;\ c' \leftarrow c) \ \mapsto & \\
 & \mathsf{proc}(c', w + w', [c', d/c, x]P) & \\
(\mathbf{1}S) & \mathsf{proc}(c, w, \mathsf{close}\ c) \ \mapsto\ \mathsf{msg}(c, w, \mathsf{close}\ c) & \\
(\mathbf{1}C) & \mathsf{msg}(c, w, \mathsf{close}\ c), \mathsf{proc}(d, w', \mathsf{wait}\ c\ ;\ Q) \ \mapsto\ \mathsf{proc}(d, w + w', Q) &
\end{array}
$$

FIGURE 3.5: Cost semantics tracking total work for programs

the work done by the message (which it gathered by possibly interacting with forwarding processes). The standard semantics rules can be obtained by simply deleting the work counters.

Work counter can be incremented only by executing the work construct.

$$(\mathsf{work}) \quad \mathsf{proc}(c, w, \mathsf{work}\ \{w'\}\ ;\ P) \ \mapsto\ \mathsf{proc}(c, w + w', P)$$

Finally, the two type constructors $\triangleright$ and its dual $\triangleleft$ are used to exchange potential. The potential is only a theoretical construct, and potentials have no role to play at runtime.

$$
\begin{array}{lll}
(\triangleright S) & \mathsf{proc}(c, w, \mathsf{pay}\ c\ \{r\}\ ;\ P) \ \mapsto & \\
 & \mathsf{proc}(c', w, [c'/c]P), \mathsf{msg}(c, 0, \mathsf{pay}\ c\ \{r\}\ ;\ c \leftarrow c') & (c' \text{ fresh}) \\
(\triangleright C) & \mathsf{msg}(c, w, \mathsf{pay}\ c\ \{r\}\ ;\ c \leftarrow c'), \mathsf{proc}(d, w', \mathsf{get}\ c\ \{r\}\ ;\ Q) \ \mapsto & \\
 & \mathsf{proc}(d, w + w', [c'/c]Q) & \\
(\triangleleft S) & \mathsf{proc}(d, w, \mathsf{pay}\ c\ \{r\}\ ;\ Q) \ \mapsto & \\
 & \mathsf{msg}(c', 0, \mathsf{pay}\ c\ \{r\}\ ;\ c' \leftarrow c), \mathsf{proc}(d, w, [c'/c]Q) & (c' \text{ fresh}) \\
(\triangleleft C) & \mathsf{proc}(c, w, \mathsf{get}\ c\ \{r\}), \mathsf{msg}(c', w', \mathsf{pay}\ c\ \{r\}\ ;\ c' \leftarrow c) \ \mapsto & \\
 & \mathsf{proc}(c', w + w', [c'/c]P) &
\end{array}
$$

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.

3.3 Type System

The typing judgment has the form

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

Intuitively, the judgment describes a process in state P using the context Δ and signature Σ and providing service along channel x of type A. In other words, P is the provider for channel $x : A$, and a client for all the channels in Δ. 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 $V = A_V$ and process definitions $\Delta \vdash^{q} f = P :: (x : A)$. The equation $V = S_V$ is used to define the type variable V as S_V. We treat such definitions *equirecursively*. For instance, $\mathsf{ctr}[n] = \&\{\mathsf{inc} : \triangleleft^{1}\mathsf{ctr}[n+1], \mathsf{val} : \triangleleft^{2\lceil \log(n+1) \rceil + 2}\mathsf{bits}\}$ exists in the signature for all $n \in \mathbb{N}$ for the binary counter system. The process definition $\Delta \vdash^{q} f = P :: (x : A)$ defines a (possibly recursive) process named f implemented by P providing along channel $x : A$ and using the channels Δ as a client, storing potential q. Messages are typed exactly as processes.

Figure 3.6 describes the usual typing rules for our system. The interesting rules here are spawn and id. The spawn splits the potential $r = p + q$, and provides potential p to the spawned process, and q to the continuation. A forwarding process $x \leftarrow y$ must be typed with no potential as it is about to terminate. The rest of the rules are standard and I am omitting their discussion. Deleting the potential annotation from the process typing judgment recovers the typing rules for standard session types.

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

$$\frac{r = p + q \quad \Delta \vdash^{p} f = P :: (x : A) \in \Sigma \quad \Delta_1 =_\alpha \Delta \quad \Sigma\ ;\ \Delta_2, (x : A) \vdash^{q} Q_x :: (z : C)}{\Sigma\ ;\ \Delta_1, \Delta_2 \vdash^{r} (x \leftarrow f \leftarrow y\ ;\ Q_x) :: (z : C)}\ \mathsf{spawn}$$

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

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

$$\frac{\Sigma\ ;\ \Delta \vdash^{q} P_\ell :: (x : A_\ell) \qquad (\forall \ell \in L)}{\Sigma\ ;\ \Delta \vdash^{q} \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R$$

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

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

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

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

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

$$\frac{q = 0}{\Sigma\ ;\ \cdot \vdash^{q} \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Sigma\ ;\ \Delta \vdash^{q} Q :: (z : C)}{\Sigma\ ;\ \Delta, (x : \mathbf{1}) \vdash^{q} \mathsf{wait}\ x\ ;\ Q :: (z : C)}\ \mathbf{1}L$$

FIGURE 3.6: Typing rules for resource-aware session types

In addition, the language has explicit rules for consuming and transfer of potential. Executing the work $\{w\}$ construct consumes w units from the potential stored in a process. Thus, a process must have at least w units of potential to execute this construct. This is expressed in the rule with the annotation $q + w$ in the conclusion.

$$\frac{\Sigma\ ;\ \Delta \vdash^{q} P :: (x : A)}{\Sigma\ ;\ \Delta \vdash^{q+w} \mathsf{work}\ \{w\}\ ;\ P :: (x : A)}\ \mathsf{work}$$

Similarly, executing a pay $x\ \{r\}$ consumes r units from the process potential, while get $x\ \{r\}$

provides r units to the process potential. The main innovation here is the introduction of the two dual type operators, $\triangleright$ and $\triangleleft$. The $\triangleright$ operator expresses that the provider must pay potential which is received by its client. Dually, the $\triangleleft$ operator requires that the provider receives potential paid by the client. The type guarantees that the potential paid by the sender equals what is gained by the recipient, thereby preserving the total potential of a configuration.

$$\frac{\Sigma\ ;\ \Delta \vdash^{q} P :: (x : A)}{\Sigma\ ;\ \Delta \vdash^{q+r} \mathsf{pay}\ x\ \{r\}\ ;\ P :: (x : \triangleright^{r} A)}\ \triangleright R$$

$$\frac{\Sigma\ ;\ \Delta, (x : A) \vdash^{q+r} Q :: (z : C)}{\Sigma\ ;\ \Delta, (x : \triangleright^{r} A) \vdash^{q} \mathsf{get}\ x\ \{r\}\ ;\ Q :: (z : C)}\ \triangleright L$$

$$\frac{\Sigma\ ;\ \Delta \vdash^{q+r} P :: (x : A)}{\Sigma\ ;\ \Delta \vdash^{q} \mathsf{get}\ x\ \{r\}\ ;\ P :: (x : \triangleleft^{r} A)}\ \triangleleft R$$

$$\frac{\Sigma\ ;\ \Delta, (x : A) \vdash^{q} Q :: (z : C)}{\Sigma\ ;\ \Delta, (x : \triangleleft^{r} A) \vdash^{q+r} \mathsf{pay}\ x\ \{r\}\ ;\ Q :: (z : C)}\ \triangleleft L$$

3.4 Soundness

This section demonstrates the soundness of the resource-aware type system with respect to the operational 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 evolves into a set of processes interacting via typed channels. Such a configuration is typed w.r.t. a well-formed signature. A signature Σ is *well formed* if (a) every type definition $V = S_V$ is contractive, and (b) every process definition $\Delta \vdash^{q} f = P :: (x : A)$ in Σ is well typed according to the process typing judgment, i.e. $\Sigma\ ;\ \Delta \vdash^{q} P :: (x : A)$. Note that the same process name f can have different resource-aware types in the signature Σ. We pick the appropriate type while applying the spawn rule.

I use the following judgment to type a configuration.

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

It states that Σ is well-formed and that the configuration $\mathcal{C}$ 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. I call E the energy of the configuration. The configuration typing judgment is defined using the rules presented in Figure 3.7. The rule empty defines that an empty configuration is well-typed with energy 0. The rule compose composes two configurations $\mathcal{C}$ and $\mathcal{C}'$; $\mathcal{C}$ provides service on the channels in Δ' while $\mathcal{C}'$ uses

$$\frac{}{\Sigma\ ;\ \Delta \overset{0}{\vDash} (\cdot) :: \Delta}\ \textsf{empty} \qquad \frac{\Sigma\ ;\ \Delta \overset{E}{\vDash} \mathcal{C} :: \Delta' \qquad \Sigma\ ;\ \Delta' \overset{E'}{\vDash} \mathcal{C}' :: \Delta''}{\Sigma\ ;\ \Delta \overset{E+E'}{\vDash} (\mathcal{C}\ \mathcal{C}') :: \Delta''}\ \textsf{compose}$$

$$\frac{\Sigma\ ;\ \Delta_1 \vdash^{q} P :: (x : A)}{\Sigma\ ;\ \Delta, \Delta_1 \overset{q+w}{\vDash} (\mathsf{proc}(x, w, P)) :: (\Delta, (x : A))}\ \textsf{proc}$$

$$\frac{\Sigma\ ;\ \Delta_1 \vdash^{q} M :: (x : A)}{\Sigma\ ;\ \Delta, \Delta_1 \overset{q+w}{\vDash} (\mathsf{msg}(x, w, M)) :: (\Delta, (x : A))}\ \textsf{msg}$$

FIGURE 3.7: Typing rules for a configuration

the channels in Δ'. The energy of the composed configuration $\mathcal{C}\ \mathcal{C}'$ is obtained by summing up their individual energies. The rule 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 msg creates a configuration out of a single message.

Soundness Theorem 3.1 is the main theorem of the chapter. 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.

Theorem 3.1 (Soundness). *Consider a well-typed configuration $\mathcal{C}$ w.r.t. a well-formed signature Σ such that $\Sigma; \Delta_1 \overset{E}{\vDash} \mathcal{C} :: \Delta_2$. If $\mathcal{C} \mapsto \mathcal{C}'$, then $\Sigma; \Delta_1 \overset{E}{\vDash} \mathcal{C}' :: \Delta_2$.*

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 preservation theorem is a corollary since soundness implies that the configuration $\mathcal{C}'$ 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 energy (equal to initial potential) is an upper bound on the total work performed by an evaluation starting in that configuration.

Corollary 3.2 (Upper Bound). *If $\Sigma; \Delta_1 \overset{E}{\vDash} \mathcal{C} :: \Delta_2$, and $\mathcal{C} \mapsto^* \mathcal{C}'$, then $E \geq W'$, 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 $\Sigma; \Delta_1 \overset{E}{\vDash} \mathcal{C} :: \Delta_2$, then $\Sigma; \Delta_1 \overset{E}{\vDash} \mathcal{C}' :: \Delta_2$. Also, $E = P' + W'$, 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 $W' \leq P' + W' = E$. In particular, if $W = 0$, we get that $P = P + W = E \geq W'$, where P and W are the potential and work of the initial configuration respectively. □

The progress theorem is a direct consequence of progress in SILL [99]. 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 this language.

3.5 Case Study: Stacks and Queues

As an illustration of the type system, I 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. Their common interface type is expressed 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, I add potential annotations leading to the store_A type to obtain two different resource-aware types for stacks and queues. The cost model again counts the total number of messages exchanged.

Stacks The type for stacks is defined as follows.

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

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 none and 2 units of potential respectively.

Queues 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 $n > 0$).

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

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. A queue is also implemented by a sequence of *elem* processes, connected via channels, terminated by the *empty* process, similar to a stack.

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. The label ins is annotated by 0 for stack_A and with $2n$ for queue_A. The label del has the same annotation in both types. Hence, an efficiency comparison can be performed by simply observing the resource-aware session types.

Queues as two stacks 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, the the first list is copied over to the second list , 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. The type of the queue is

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{\ &\mathsf{ins} : \triangleleft^{6}(A \multimap \mathsf{queue}_A[n+1]),\\ &\mathsf{del} : \triangleleft^2 \oplus \{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{queue}_A[n \dot{-} 1]\}\}\end{aligned}$$

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. 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.

3.6 Related Work

In the context of process calculi, capabilities [98] and static analyses [71] 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 [105]. 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., [35, 74]). The 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 [61] and polynomial bounds [60] for sequential functional programs. It can also be integrated with program logics to derive bounds for imperative programs [16, 31]. Moreover, it has been used to derive bounds for term-rewrite systems [63] and object-oriented programs [62]. A recent work also considers bounds on the parallel evaluation cost (also called *span*) of functional programs [57]. 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 sent along channels. As a result, the formulation and proof of the soundness theorem is quite different from the soundness of sequential AARA.

I am only aware of a couple of other works that study resource bounds for concurrent programs. Gimenez et al. [47] 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. [12, 14] 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.

3.7 Future Directions

I briefly mention some important future directions with regard to work analysis. The main direction that I plan to explore as part of my proposed work is an implementation of resource-aware session types. This has been detailed in Chapter 6. In addition, I describe some other important future directions that are outside the scope of this thesis.

Work Inference To completely automate the type system, it is crucial to infer the work bounds, not just check them. This entails inserting pay or get constructors with every continuation with a parametric value and then obtaining constraints on these parameters. Then a solver tries to solve these constraints while minimizing the number of such constructors that need to be inserted. If an algorithmic version of this inference is implemented, a programmer will simply write the original simple session-typed program, and the inference engine will infer the resource-aware type, along with the resource bound.

Process Scheduling Inferring work bounds has several applications. One such direction to explore is the use of resource bounds in process scheduling. For instance, oracle schedulers [8] can use a priori knowledge of the runtime of each parallel thread to calculate thread creation overheads and enhance efficiency. Thus, resource-aware session types can be used to design an efficient scheduling algorithm that maximizes throughput.

Chapter 4

Time Analysis

Analyzing the complexity of concurrent, message-passing processes poses additional challenges over sequential programs. To begin with, we need information about the possible interactions between processes to enable compositional and local reasoning about concurrent cost. In addition to the structure of communication, the timing of messages is of central interest for analyzing concurrent cost. With information on message timing we may analyze not only properties such as the rate or latency with which a stream of messages can proceed through a pipeline, but also the span of a parallel computation, which can be defined as the time of the final response message assuming maximal parallelism.

There are several possible ways to enrich session types with timing information. A challenge is to find a balance between precision and flexibility. We would like to express precise times according to a global clock as in synchronous data flow languages whenever that is possible. However, sometimes this will be too restrictive. For example, we may want to characterize the response time of a concurrent queue where enqueue and dequeue operations arrive at unpredictable intervals.

In this chapter, I develop a type system that captures the parallel complexity of session-typed message-passing programs by adding *temporal modalities next* ($\bigcirc A$), *always* ($\Box A$), and *eventually* ($\Diamond A$), interpreted over a linear model of time. When considered as types, the temporal modalities express properties of concurrent programs such as the *message rate* of a stream, the *latency* of a pipeline, the *response time* of concurrent data structure, or the *span* of a fork/join parallel program, all in the same uniform manner. The results complement my prior work on expressing the *work* of session-typed processes in the same base language [38]. Together, they form a foundation for analyzing the parallel complexity of session-typed processes.

The type system is constructed conservatively over the base language of session types, which makes it quite general and easily able to accommodate various concrete cost models. The language contains standard session types and process expressions, and their typing rules remain unchanged. They correspond to processes that do not induce cost and send all messages at the constant time 0.

To model computation cost, a new syntactic form delay is introduced, which advances time by one step. A particular cost semantics is specified by taking an ordinary, non-temporal program and adding delays capturing the intended cost. For example, if only the blocking operations should cost one unit of time, a delay is added before the continuation of every receiving construct. If sends should have unit cost as well, a delay is added immediately after each send operation. Processes that contain delays cannot be typed using standard session types.

To type processes with non-zero cost, I first introduce the type $\bigcirc A$, which is inhabited only by the process expression $(\mathsf{delay}\ ;\ P)$. This forces time to advance on all channels that P can communicate along. The resulting types prescribe the *exact* time a message is sent or received and sender and receiver are precisely synchronized.

As an example, consider a stream of bits terminated by \$, expressed as the recursive type

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$$

where $\oplus$ stands for *internal choice* and $\mathbf{1}$ for *termination*, ending the session. A simple cost model for asynchronous communication prescribes a cost of one unit of time for every receive operation. A stream of bits then needs to delay every continuation to give the recipient time to receive the message, expressing a *rate* of one. This can be captured precisely with the temporal modality $\bigcirc A$:

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$

A transducer *neg* that negates each bit it receives along channel x and passes it on along channel y would be typed as

$$x : \mathsf{bits} \vdash \mathit{neg} :: (y : \bigcirc\mathsf{bits})$$

expressing a *latency* of one. A process *negneg* that puts two negations in sequence has a latency of two, compared with *copy* which passes on each bit, and *id* which terminates and identifies the channel y with the channel x, short-circuiting the communication.

$$x : \mathsf{bits} \vdash \mathit{negneg} :: (y : \bigcirc\bigcirc\mathsf{bits}) \qquad x : \mathsf{bits} \vdash \mathit{copy} :: (y : \bigcirc\mathsf{bits}) \qquad x : \mathsf{bits} \vdash \mathit{id} :: (y : \mathsf{bits})$$

All these processes have the same extensional behavior, but different latencies. They also have the same rate since after the pipelining delay, the bits are sent at the same rate they are received, as expressed in the common type bits used in the context and the result.

While precise and minimalistic, the resulting system is often too precise for typical concurrent programs such as pipelines or servers. I therefore introduce the dual type formers $\Diamond A$ and $\Box A$ to talk about varying time points in the future. Remarkably, even if part of a program is typed using these constructs, we can still make precise and useful statements about other aspects.

For example, consider a transducer *compress* that shortens a stream by combining consecutive 1 bits so that, for example, 00110111 becomes 00101. For such a transducer, we cannot bound the latency statically, even if the bits are received at a constant rate like in the type bits. So we have to express that after seeing a 1 bit we will *eventually* see either another bit or the end of the stream. For this purpose, we introduce a new type sbits with the same message alternatives as bits, but different timing. In particular, after sending b1, either the next bit or end-of-stream is *eventually* sent ($\Diamond$sbits), rather than immediately.

$$\mathsf{sbits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{sbits}, \mathsf{b1} : \bigcirc\Diamond\mathsf{sbits}, \$: \bigcirc\mathbf{1}\}$$
$$x : \mathsf{bits} \vdash \textit{compress} :: (y : \bigcirc\mathsf{sbits})$$

We write $\bigcirc\Diamond$sbits instead of $\Diamond$sbits for the continuation type after b1 to express that there will always be a delay of at least one; to account for the unit cost of receive in this particular cost model.

The dual modality, $\Box A$, is useful to express, for example, that a server providing A is *always* ready, starting from "now". As an example, consider the following temporal type of an interface to a process of type $\Box\mathsf{queue}_A$ with elements of type $\Box A$. It expresses that there must be at least four time units between successive enqueue operations and that the response to a dequeue request is immediate, only one time unit later ($\&$ stands for external choice, the dual to internal choice).

$$\begin{aligned}\mathsf{queue}_A = \&\{\ &\mathsf{enq} : \bigcirc(\Box A \multimap \bigcirc^3\Box\mathsf{queue}_A),\\ &\mathsf{deq} : \bigcirc\oplus\{\ \mathsf{none} : \bigcirc\mathbf{1}, \mathsf{some} : \bigcirc(\Box A \otimes \bigcirc\Box\mathsf{queue}_A)\ \}\ \}\end{aligned}$$

As an example of a *parametric* cost analysis, the following type can be given to a process that appends inputs l_1 and l_2 to yield l, where the message rate on all three lists is $r + 2$ units of time (that is, the interval between consecutive list elements needs to be at least 2).

$$l_1 : \mathsf{list}_A[n], l_2 : \bigcirc^{(r+4)n+2}\mathsf{list}_A[k] \vdash \textit{append} :: (l : \bigcirc\bigcirc\mathsf{list}_A[n+k])$$

It expresses that *append* has a latency of two units of time and that it inputs the first message from l_2 after $(r + 4)n + 2$ units of time, where n is the number of elements sent along l_1.

To analyze the span of a fork/join parallel program, we capture the time at which the (final) answer is sent. For example, the type $\mathsf{tree}[h]$ describes the span of a process that computes the parity of a binary tree of height h with boolean values at the leaves. The session type expresses that the result of the computation is a single boolean that arrives at time $5h+3$ after the parity request.

$$\mathsf{tree}[h] = \&\{\ \mathsf{parity} : \bigcirc^{5h+3}\mathsf{bool}\ \}$$

In summary, the main contributions of the chapter are (1) a generic framework for parallel cost analysis of asynchronously communicating session-typed processes rooted in a novel combination of temporal and linear logic, (2) a soundness proof of the type system with respect to a timed operational semantics, showing progress and type preservation (3) instantiations of the framework with different cost models, e.g. where either just receives, or receives and sends, cost one time unit each, and (4) examples illustrating the scope of my method. My technique for proving progress and preservation does not require dependency graphs and may be of independent interest. I further provide decidable systems for *time reconstruction* and *subtyping* that greatly simplify the programmer's task. They also enhance modularity by allowing the same program to be assigned temporally different types, depending on the context of use.

4.1 The Temporal Modality Next ($\bigcirc A$)

This section introduces *actual cost* by explicitly advancing time. Remarkably, all the rules presented so far in Chapter 2 remain literally unchanged. They correspond to the cost-free fragment of the language in which time never advances. In addition, I have a new type construct $\bigcirc A$ (read: *next* A) with a corresponding process construct $(\mathsf{delay}\ ;\ P)$, which advances time by one unit. In the corresponding typing rule

$$\frac{\Delta \vdash P :: (x : A)}{\bigcirc\Delta \vdash (\mathsf{delay}\ ;\ P) :: (x : \bigcirc A)}\ \bigcirc LR$$

I abbreviate $y_1{:}\bigcirc A_1, \ldots, y_m{:}\bigcirc A_m$ by $\bigcirc(y_1{:}A_1, \ldots, y_m{:}A_m)$. Intuitively, when $(\mathsf{delay}\ ;\ P)$ idles, time advances on *all* channels connected to P. Computationally, I delay the process for one time unit without any external interactions. To understand this computation, I introduce semantic objects $\mathsf{proc}(c, t, P)$ and $\mathsf{msg}(c, t, M)$ which mean that process P or message M provide along channel c and are at an integral time t.

$$(\bigcirc C) \quad \mathsf{proc}(c, t, \mathsf{delay}\ ;\ P) \ \mapsto\ \mathsf{proc}(c, t+1, P)$$

There is a subtle point about forwarding: A process $\mathsf{proc}(c, t, c \leftarrow d)$ may be ready to forward a message *before* a client reaches time t while in all other rules the times must match exactly. We can avoid this mismatch by transforming uses of forwarding $x \leftarrow y$ at type $\bigcirc^n S$ where $S \neq \bigcirc(-)$ to $(\mathsf{delay}^n\ ;\ x \leftarrow y)$. In this discussion I have used the following notation which will be useful later:

$$\begin{array}{lcllcl} \bigcirc^0 A & = & A & \mathsf{delay}^0\ ;\ P & = & P \\ \bigcirc^{n+1} A & = & \bigcirc\bigcirc^n A & \mathsf{delay}^{n+1}\ ;\ P & = & \mathsf{delay}\ ;\ \mathsf{delay}^n\ ;\ P \end{array}$$

4.1.1 Modeling a Cost Semantics

My system allows us to represent a variety of different abstract cost models in a straightforward way. I will mostly use two different abstract cost models. In the first, called $\mathcal{R}$, I assign unit cost to every receive (or wait) action while all other operations remain cost-free. We may be interested in this since receiving a message is the only blocking operation in the asynchronous semantics. A second one, called $\mathcal{RS}$ and considered in Section 4.4, assigns unit cost to both send and receive actions.

To capture $\mathcal{R}$ I take a source program and insert a delay operation before the continuation of every receive. I write this delay as tick in order to remind the reader that it arises systematically from the cost model and is never written by the programmer. In all other respects, tick is just a synonym for delay.

For example, the copy process would become

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$$

$$y : \mathsf{bits} \vdash \mathit{copy} :: (x : \mathsf{bits}) \qquad \% \textit{ No longer correct!}$$
$$x \leftarrow \mathit{copy} \leftarrow y =$$
$$\quad \mathsf{case}\ y\ (\ \mathsf{b0} \Rightarrow \mathsf{tick}\ ;\ x.\mathsf{b0}\ ;\ x \leftarrow \mathit{copy} \leftarrow y$$
$$\qquad\quad \mid \mathsf{b1} \Rightarrow \mathsf{tick}\ ;\ x.\mathsf{b1}\ ;\ x \leftarrow \mathit{copy} \leftarrow y$$
$$\qquad\quad \mid \$ \Rightarrow \mathsf{tick}\ ;\ x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{tick}\ ;\ \mathsf{close}\ x\ )$$

As indicated in the comment, the type of *copy* is now no longer correct because the bits that arrive along y are delayed by one unit before they are sent along x. We can observe this concretely by starting to type-check the first branch

$$y : \mathsf{bits} \vdash \mathit{copy} :: (x : \mathsf{bits})$$
$$x \leftarrow \mathit{copy} \leftarrow y =$$
$$\quad \mathsf{case}\ y\ (\ \mathsf{b0} \Rightarrow \qquad\qquad \% \ y : \mathsf{bits} \vdash x : \mathsf{bits}$$
$$\qquad\qquad\quad \mathsf{tick}\ ;\ \ldots)$$

We see that the delay tick does not type-check, because neither x nor y have a type of the form $\bigcirc(-)$. We need to redefine the type bits so that the continuation type after every label is delayed by one, anticipating the time it takes to receive the label b0, b1, or \$. Similarly, we capture in the type of *copy* that its *latency* is one unit of time.

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$

$$y : \mathsf{bits} \vdash \mathit{copy} :: (x : \bigcirc\mathsf{bits})$$

With these declarations, we can now type-check the definition of *copy*. I show the intermediate type of the used and provided channels after each interaction.

```
x ← copy ← y =
  case y  ( b0 ⇒                    % y : ○bits ⊢ x : ○bits
            tick ;                  % y : bits ⊢ x : bits
            x.b0 ;                  % y : bits ⊢ x : ○bits
            x ← copy ← y            % well-typed by type of copy
          | b1 ⇒                    % y : ○bits ⊢ x : ○bits
            tick ;                  % y : bits ⊢ x : bits
            x.b1 ;                  % y : bits ⊢ x : ○bits
            x ← copy ← y
          | $ ⇒                     % y : ○1 ⊢ x : ○bits
            tick ;                  % y : 1 ⊢ x : bits
            x.$ ;                   % y : 1 ⊢ x : ○1
            wait y ;                % · ⊢ x : ○1
            tick ;                  % · ⊢ x : 1
            close x )
```

Armed with this experience, we now consider the increment process *plus1*. Again, we expect the latency of the increment to be one unit of time. Since we are interested in detailed type-checking, I show the transformed program, with a delay tick after each receive.

```
bits = ⊕{b0 : ○bits, b1 : ○bits, $ : ○1}
y : bits ⊢ plus1 :: (x : ○bits)
x ← plus1 ← y =
  case y  ( b0 ⇒ tick ; x.b1 ; x ← y                % type error here!
          | b1 ⇒ tick ; x.b0 ; x ← plus1 ← y
          | $ ⇒ tick ; x.$ ; wait y ; tick ; close x )
```

The branches for b1 and $ type-check as before, but the branch for b0 does not. I make the types at the crucial point explicit:

```
x ← plus1 ← y =
  case y  ( b0 ⇒ tick ; x.b1 ;          % y : bits ⊢ x : ○bits
                 x ← y                  % ill-typed, since bits ≠ ○bits
          | ... )
```

The problem here is that identifying x and y removes the delay mandated by the type of *plus1*. A solution is to call *copy* to reintroduce the latency of one time unit.

```
y : bits ⊢ plus1 :: (x : ○bits)
x ← plus1 ← y =
  case y  ( b0 ⇒ tick ; x.b1 ; x ← copy ← y
```

$$
\begin{array}{l}
\quad \mid \mathsf{b1} \Rightarrow \mathsf{tick}\ ;\ x.\mathsf{b0}\ ;\ x \leftarrow \mathit{plus1} \leftarrow y \\
\quad \mid \$ \Rightarrow \mathsf{tick}\ ;\ x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{tick}\ ;\ \mathsf{close}\ x\ )
\end{array}
$$

In order to write *plus2* as a pipeline of two increments we need to delay the second increment explicitly in the program and stipulate, in the type, that there is a latency of two.

$$
\begin{array}{l}
y : \mathsf{bits} \vdash \mathit{plus2} :: (x : \bigcirc\bigcirc\mathsf{bits}) \\
x \leftarrow \mathit{plus2} \leftarrow y = \\
\quad z \leftarrow \mathit{plus1} \leftarrow y\ ; \qquad\qquad \% \ z : \bigcirc\mathsf{bits} \vdash x : \bigcirc\bigcirc\mathsf{bits} \\
\quad \mathsf{delay}\ ; \qquad\qquad \% \ z : \mathsf{bits} \vdash x : \bigcirc\mathsf{bits} \\
\quad x \leftarrow \mathit{plus1} \leftarrow z
\end{array}
$$

Programming with so many explicit delays is tedious, but fortunately a source program without all these delay operations (but explicitly temporal session types) can be transformed automatically in two steps: (1) insert the delays mandated by the cost model (here: a tick after each receive), and (2) perform *time reconstruction* to insert the additional delays so the result is temporally well-typed or issue an error message if this is impossible (see [39]).

4.1.2 The Interpretation of a Configuration

Let us reconsider the program to produce the number $6 = (110)_2$ under the cost model $\mathcal{R}$ where each receive action costs one unit of time. There are no receive operations in this program, but time reconstruction must insert a delay after each send in order to match the delays mandated by the type bits.

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$

$$
\begin{array}{l}
\cdot \vdash \mathit{six} :: (x : \mathsf{bits}) \\
x \leftarrow \mathit{six} = x.\mathsf{b0}\ ;\ \mathsf{delay}\ ;\ x.\mathsf{b1}\ ;\ \mathsf{delay}\ ;\ x.\mathsf{b1}\ ;\ \mathsf{delay}\ ;\ x.\$\ ;\ \mathsf{delay}\ ;\ \mathsf{close}\ x
\end{array}
$$

Executing $\mathsf{proc}(c_0, 0, c_0 \leftarrow \mathit{six})$ then leads to the following configuration

$$
\begin{array}{l}
\mathsf{msg}(c_4, 4, \mathsf{close}\ c_4), \\
\mathsf{msg}(c_3, 3, c_3.\$\ ;\ c_3 \leftarrow c_4), \\
\mathsf{msg}(c_2, 2, c_2.\mathsf{b1}\ ;\ c_2 \leftarrow c_3), \\
\mathsf{msg}(c_1, 1, c_1.\mathsf{b1}\ ;\ c_1 \leftarrow c_2), \\
\mathsf{msg}(c_0, 0, c_0.\mathsf{b0}\ ;\ c_0 \leftarrow c_1)
\end{array}
$$

These messages are at increasing times, which means any client of c_0 will have to immediately (at time 0) receive b0, then (at time 1) b1, then (at time 2) b1, etc. In other words, the time stamps on messages predict *exactly* when the message will be received. Of course, if there is a client in parallel this state may never be reached because, for example, the first b0 message along channel c_0 may be received before the continuation of the sender produces the message

b1. So different configurations may be reached depending on the *scheduler* for the concurrent processes. It is also possible to give a time-synchronous semantics in which all processes proceed *in parallel* from time 0 to time 1, then from time 1 to time 2, etc.

4.2 The Temporal Modalities Always ($\Box A$) and Eventually ($\Diamond A$)

The strength and also the weakness of the system so far is that its timing is very precise. Consider a process *compress* that combines runs of consecutive 1's to a single 1. For example, compressing 11011100 should yield 10100. First, in the cost-free the process is defined as

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$$

$$y : \mathsf{bits} \vdash \textit{compress} :: (x : \mathsf{bits})$$
$$y : \mathsf{bits} \vdash \textit{skip1s} :: (x : \mathsf{bits})$$

$$\begin{array}{l} x \leftarrow \textit{compress} \leftarrow y = \\ \quad \mathsf{case}\ y\ \ (\ \mathsf{b0} \Rightarrow x.\mathsf{b0}\ ;\ x \leftarrow \textit{compress} \leftarrow y \\ \qquad\quad \mid \mathsf{b1} \Rightarrow x.\mathsf{b1}\ ;\ x \leftarrow \textit{skip1s} \leftarrow y \\ \qquad\quad \mid \$ \Rightarrow x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{close}\ x\ ) \end{array}$$

$$\begin{array}{l} x \leftarrow \textit{skip1s} \leftarrow y = \\ \quad \mathsf{case}\ y\ \ (\ \mathsf{b0} \Rightarrow x.\mathsf{b0}\ ;\ x \leftarrow \textit{compress} \leftarrow y \\ \qquad\quad \mid \mathsf{b1} \Rightarrow x \leftarrow \textit{skip1s} \leftarrow y \\ \qquad\quad \mid \$ \Rightarrow x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{close}\ x\ ) \end{array}$$

The problem is that program cannot be typed under the cost mode $\mathcal{R}$, where every receive takes one unit of time. Actually worse: there is no way to insert next-time modalities into the type and additional delays into the program so that the result is well-typed. This is because if the input stream is unknown we cannot predict how long a run of 1's will be, but the length of such a run will determine the delay between sending a bit 1 and the following bit 0.

The best we can say is that after a bit 1 *compress* will *eventually* send either a bit 0 or the end-of-stream token \$. This is the purpose of the type $\Diamond A$. We capture this timing in the type sbits (for *slow bits*).

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$
$$\mathsf{sbits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{sbits}, \mathsf{b1} : \bigcirc\Diamond\mathsf{sbits}, \$: \bigcirc\mathbf{1}\}$$

$$y : \mathsf{bits} \vdash \textit{compress} :: (x : \bigcirc\mathsf{sbits})$$
$$y : \mathsf{bits} \vdash \textit{skip1s} :: (x : \bigcirc\Diamond\mathsf{sbits})$$

The next section introduces the process constructs and typing rules so that *compress* and *skip1s* programs can be revised to have the right temporal semantics.

4.2.1 Eventually A

A process providing $\Diamond A$ promises only that it will eventually provide A. There is a somewhat subtle point here: since not every action may require time and because we do not check termination separately, $x : \Diamond A$ expresses only that *if the process providing x terminates* it will eventually provide A. Thus, it expresses non-determinism regarding the (abstract) *time* at which A is provided, rather than a strict liveness property. Therefore, $\Diamond A$ is somewhat weaker than one might be used to from LTL [87]. When restricted to a purely logical fragment, without unrestricted recursion, the usual meaning is fully restored so I feel the terminology is justified. Imposing termination, for example along the lines of Fortier and Santocanale [43] or Toninho et al. [100] is an interesting item for future work but not necessary for our present purposes.

When a process offering $c : \Diamond A$ is ready, it will send a now! message along c and then continue at type A. Conversely, the client of $c : \Diamond A$ will have to be ready and waiting for the now! message to arrive along c and then continue at type A. I use $(\text{when? } c \ ; \ Q)$ for the corresponding client. These explicit constructs are a conceptual device and may not need to be part of an implementation. They also make type-checking processes entirely syntax-directed and trivially decidable.

The typing rules for now! and when? are somewhat subtle.

$$\frac{\Delta \vdash P :: (x : A)}{\Delta \vdash (\text{now! } x \ ; \ P) :: (x : \Diamond A)} \ \Diamond R$$

$$\frac{\Delta \ \text{delayed}^{\Box} \quad \Delta, x : A \vdash Q :: (z : C) \quad C \ \text{delayed}^{\Diamond}}{\Delta, x : \Diamond A \vdash (\text{when? } x \ ; \ Q) :: (z : C)} \ \Diamond L$$

The $\Diamond R$ rule just states that, without constraints, we can at any time decide to communicate along $x : \Diamond A$ and then continue the session at type A. The $\Diamond L$ rule expresses that the process must be ready to receive a now! message along $x : \Diamond A$, but there are two further constraints. Because the process $(\text{when? } x \ ; \ Q)$ may need to wait an indefinite period of time, the rule must make sure that communication along z and any channel in Δ can also be postponed an indefinite period of time. The predicate $C \ \text{delayed}^{\Diamond}$ describes that C must have the form $\bigcirc^* \Diamond C'$ to require that C may be delayed a fixed finite number of time steps and then must be allowed to communicate at an arbitrary time in the future. Similarly, for every channel $y : B$ in Δ, B must have the form $\bigcirc^* \Box B$, where $\Box$ (as the dual of $\Diamond$) is introduced in Section 4.2.

In the operational semantics, the central restriction is that when? is ready *before* the now! message arrives so that the continuation can proceed immediately as promised by the type.

$$(\Diamond S) \quad \mathsf{proc}(c, t, \text{now! } c \ ; \ P) \ \mapsto \ \mathsf{proc}(c', t, [c'/c]P), \mathsf{msg}(c, t, \text{now! } c \ ; \ c \leftarrow c') \quad (c' \text{ fresh})$$
$$(\Diamond C) \quad \mathsf{msg}(c, t, \text{now! } c \ ; \ c \leftarrow c'), \mathsf{proc}(d, s, \text{when? } c \ ; \ Q) \ \mapsto \ \mathsf{proc}(d, t, [c'/c]Q) \quad (t \geq s)$$

To rewrite the *compress* process in our cost model $\mathcal{R}$, I first insert tick before all the actions that must be delayed according to our cost model. Then I insert appropriate additional delay, when?,

and now! actions. While *compress* turns out to be straightforward, *skip1s* creates a difficulty after it receives a b1:

$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$
$\mathsf{sbits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{sbits}, \mathsf{b1} : \bigcirc\Diamond\mathsf{sbits}, \$: \bigcirc\mathbf{1}\}$

$y : \mathsf{bits} \vdash \mathit{compress} :: (x : \bigcirc\mathsf{sbits})$
$y : \mathsf{bits} \vdash \mathit{skip1s} :: (x : \bigcirc\Diamond\mathsf{sbits})$

$x \leftarrow \mathit{compress} \leftarrow y =$
  case y (b0 $\Rightarrow$ tick ; x.b0 ; $x \leftarrow \mathit{compress} \leftarrow y$
  | b1 $\Rightarrow$ tick ; x.b1 ; $x \leftarrow \mathit{skip1s} \leftarrow y$
  | \$ $\Rightarrow$ tick ; x.\$; wait y ; tick ; close x)

$x \leftarrow \mathit{skip1s} \leftarrow y =$
  case y (b0 $\Rightarrow$ tick ; now! x ; x.b0 ; $x \leftarrow \mathit{compress} \leftarrow y$
  | b1 $\Rightarrow$ tick ; % $y : \mathsf{bits} \vdash x : \Diamond\mathsf{sbits}$
  $x' \leftarrow \mathit{skip1s} \leftarrow y$; % $x' : \bigcirc\Diamond\mathsf{sbits} \vdash x : \Diamond\mathsf{sbits}$
  $x \leftarrow \mathit{idle} \leftarrow x'$ % with $x' : \bigcirc\Diamond\mathsf{sbits} \vdash \mathit{idle} :: (x : \Diamond\mathsf{sbits})$
  | \$ $\Rightarrow$ tick ; now! x ; x.\$; wait y ; tick ; close x)

At the point where I would like to call *skip1s* recursively, I have

$y : \mathsf{bits} \vdash x : \Diamond\mathsf{sbits}$
but $y : \mathsf{bits} \vdash \mathit{skip1s} :: (x : \bigcirc\Diamond\mathsf{sbits})$

which prevents a tail call since $\bigcirc\Diamond\mathsf{sbits} \neq \Diamond\mathsf{sbits}$. Instead *skip1s* is called to obtain a new channel x' and then use another process called *idle* to go from $x' : \bigcirc\Diamond\mathsf{sbits}$ to $x : \Diamond\mathsf{sbits}$. Intuitively, it should be possible to implement such an idling process: $x : \Diamond\mathsf{sbits}$ expresses *at some time in the future, including possibly right now* while $x' : \bigcirc\Diamond\mathsf{sbits}$ says *at some time in the future, but not right now.*

To type the idling process, the $\bigcirc LR$ rule needs to be generalized to account for the interactions of $\bigcirc A$ with $\Box A$ and $\Diamond A$. After all, they speak about the same underlying model of time.

4.2.2 Interactions of $\bigcirc A$ and $\Diamond A$

Recall the left/right rule for $\bigcirc$:

$$\frac{\Delta \vdash P :: (x : A)}{\bigcirc\Delta \vdash (\mathsf{delay}\ ;\ P) :: (x : \bigcirc A)}\ \bigcirc LR$$

If the succedent were $x : \Diamond A$ instead of $x : \bigcirc A$, we should still be able to delay since we can freely choose when to interact along x. We could capture this in the following rule (superseded

later by a more general form of $\bigcirc LR$):

$$\frac{\Delta \vdash P :: (x : \Diamond A)}{\bigcirc\Delta \vdash (\mathsf{delay}\ ;\ P) :: (x : \Diamond A)}\ \bigcirc\Diamond$$

I keep $\Diamond A$ as the type of x since I want to retain the full flexibility of using x at any time in the future after the initial delay. I will generalize the rule once more in the next section to account for interactions with $\Box A$.

With this, I can define and type the idling process parametrically over A:

$x' : \bigcirc\Diamond A \vdash \mathit{idle} :: (x : \Diamond A)$
$x \leftarrow \mathit{idle} \leftarrow x' = \mathsf{delay}\ ;\ x \leftarrow x'$

This turns out to be an example of subtyping (see [39]), which means that the programmer actually will not have to explicitly define or even reference an idling process. The programmer simply writes the original *skip1s* process (without referencing the *idle* process) and our subtyping algorithm will use the appropriate rule to typecheck it successfully.

4.2.3 Always A

The last temporal modality, written as $\Box A$ (read: *always* A), is dual to $\Diamond A$. If a process P provides $x : \Box A$ it means it is ready to receive a now! message along x at any point in the future. In analogy with the typing rules for $\Diamond A$, but flipped to the other side of the sequent, we obtain

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

The operational rules just reverse the role of provider and client from the rules for $\Diamond A$.

$$(\Box S) \quad \mathsf{proc}(d, t, \mathsf{now!}\ c\ ;\ Q) \mapsto \mathsf{msg}(c', t, \mathsf{now!}\ c\ ;\ c' \leftarrow c), \mathsf{proc}(d, t, [c'/c]Q) \quad (c' \text{ fresh})$$
$$(\Box C) \quad \mathsf{proc}(c, s, \mathsf{when?}\ c\ ;\ P), \mathsf{msg}(c', t, \mathsf{now!}\ c\ ;\ c' \leftarrow c) \mapsto \mathsf{proc}(c', t, [c'/c]P) \quad (s \leq t)$$

As an example for the use of $\Box A$, and also to introduce a new kind of example, I specify and implement a counter process that can receive inc and val messages. When receiving an inc it will increment its internally maintained counter, when receiving val it will produce a finite bit stream representing the current value of the counter. In the cost-free setting the type is

$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$
$\mathsf{ctr} = \&\{\mathsf{inc} : \mathsf{ctr}, \mathsf{val} : \mathsf{bits}\}$

A counter is implemented by a chain of processes, each holding one bit (either *bit0* or *bit1*) or signaling the end of the chain (*empty*). For this purpose we implement three processes:

$d : \mathsf{ctr} \vdash \mathit{bit0} :: (c : \mathsf{ctr})$
$d : \mathsf{ctr} \vdash \mathit{bit1} :: (c : \mathsf{ctr})$
$\cdot \vdash \mathit{empty} :: (c : \mathsf{ctr})$

```
c ← bit0 ← d =
  case c ( inc ⇒ c ← bit1 ← d                % increment by continuing as bit1
         | val ⇒ c.b0 ; d.val ; c ← d )      % send b0 on c, send val on d, identify c and d

c ← bit1 ← d =
  case c ( inc ⇒ d.inc ; c ← bit0 ← d        % send inc (carry) on d, continue as bit1
         | val ⇒ c.b1 ; d.val ; c ← d )      % send b1 on c, send val on d, identify c and d

c ← empty =
  case c ( inc ⇒ e ← empty ;                 % spawn a new empty process with channel e
                 c ← bit1 ← e                % continue as bit1
         | val ⇒ c.$ ; close c )             % send $ on c and close c
```

Using our standard cost model $\mathcal{R}$ there is a problem: the *carry bit* (the d.inc message sent in the *bit1* process) is sent only on every other increment received because *bit0* continues as *bit1 without* a carry, and *bit1* continues as *bit0 with* a carry. So it will actually take 2^k increments received at the lowest bit of the counter (which represents the interface to the client) before an increment reaches the kth process in the chain. This is not a constant number, so the behavior cannot be characterized exactly using only the next time modality. Instead, I require, from a certain point on, a counter is always ready to receive either an inc or val message.

$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$
$\mathsf{ctr} = \square\&\{\mathsf{inc} : \bigcirc\mathsf{ctr}, \mathsf{val} : \bigcirc\mathsf{bits}\}$

In the program, the ticks are mandated by our cost model and some additional delay, when?, and now! actions are present to satisfy the stated types. The two marked lines may look incorrect, but are valid based on the generalization of the $\bigcirc LR$ rule in Section 4.2.

$d : \bigcirc\mathsf{ctr} \vdash \mathit{bit0} :: (c : \mathsf{ctr})$
$d : \mathsf{ctr} \vdash \mathit{bit1} :: (c : \mathsf{ctr})$
$\cdot \vdash \mathit{empty} :: (c : \mathsf{ctr})$

```
c ← bit0 ← d =
  when? c ;                                  % d : ○ctr ⊢ c : &{...}
  case c ( inc ⇒ tick ;                      % d : ctr ⊢ c : ctr
                 c ← bit1 ← d
         | val ⇒ tick ;                      % d : ctr ⊢ c : bits
                 c.b0 ;                      % d : ctr ⊢ c : ○bits
                 now! d ; d.val ;            % d : ○bits ⊢ c : ○bits
                 c ← d )
```

```
c ← bit1 ← d =
  when? c ;                          % d : ctr ⊢ c : &{...}
  case c ( inc ⇒ tick ;              % d : ctr ⊢ c : ctr      (see Section 4.2)
                 now! d ; d.inc ;    % d : ○ctr ⊢ c : ctr
                 c ← bit0 ← d
         | val ⇒ tick ;              % d : ctr ⊢ c : bit      (see Section 4.2)
                 c.b1 ;              % d : ctr ⊢ c : ○bits
                 now! d ; d.val ;    % d : ○bits ⊢ c : ○bits
                 c ← d )
c ← empty =
  when? c ;                          % · ⊢ c : &{...}
  case c ( inc ⇒ tick ;              % · ⊢ c : ctr
                 e ← empty ;         % e : ctr ⊢ c : ctr
                 c ← bit1 ← e
         | val ⇒ tick ; c.$ ;        % · ⊢ c : ○1
                 delay ; close c )
```

4.2.4 Interactions Between Temporal Modalities

Just as $\bigcirc A$ and $\Diamond A$ interacted in the rules since their semantics is based on the same underlying notion of time, so do $\bigcirc A$ and $\Box A$. Executing a delay allows any channel of type $\Box A$ that is used and leaves its type unchanged because we are not obligated to communicate along it at any particular time. To cover all the cases, I introduce a new notation, writing $[A]_L^{-1}$ and $[A]_R^{-1}$ on types and extend it to contexts. Depending on one's point of view, this can be seen as stepping forward or backward by one unit of time.

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

Here, S stands for any basic session type constructor as in Table 2.1. We use this notation in the general rule $\bigcirc LR$ which can be found in Figure 4.1 together with the final set of rules for $\Box A$ and $\Diamond A$. In conjunction with the rules in Chapter 2 this completes the system of temporal session types where all temporal actions are explicit. The rule $\bigcirc LR$ only applies if both $[\Delta]_L^{-1}$ and $[x : A]_R^{-1}$ are defined.

A type A is called *patient* if it does not force communication along a channel $x : A$ at any particular point in time. Because the direction of communication is reversed between the two sides of a sequent, a type A is patient if it has the form $\bigcirc^* \Box A'$ if it is among the antecedents, and $\bigcirc^* \Diamond A'$ if it is in the succedent. The judgments A delayed$^{\Box}$ and A delayed$^{\Diamond}$ are a shorthand for

$$\frac{[\Omega]_L^{-1} \vdash P :: [x : A]_R^{-1}}{\Omega \vdash (\mathsf{delay}\ ;\ P) :: (x : A)}\ \bigcirc LR \qquad \frac{}{\bigcirc^* \Box A\ \mathsf{delayed}^{\Box}} \qquad \frac{}{\bigcirc^* \Diamond A\ \mathsf{delayed}^{\Diamond}}$$

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

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

FIGURE 4.1: Explicit Temporal Typing Rules

patient types. Further, $A\ \mathsf{delayed}^{\Box}$ is extended to contexts $\Delta\ \mathsf{delayed}^{\Box}$ if for every declaration $(x : A) \in \Delta$, $A\ \mathsf{delayed}^{\Box}$ holds.

4.3 Preservation and Progress

The main theorems that exhibit the deep connection between our type system and the timed operational semantics are the usual *type preservation* and *progress*, sometimes called *session fidelity* and *deadlock freedom*, respectively.

4.3.1 Configuration Typing

A key question is how we type configurations $\mathcal{C}$. Configurations consist of multiple processes and messages, so they both *use* and *provide* a collection of channels. And even though we treat a configuration as a multiset, typing imposes a partial order on the processes and messages where a provider of a channel appears to the left of its client.

$$\mathsf{Configuration} \quad \mathcal{C} \quad ::= \quad \cdot \mid \mathcal{C}\ \mathcal{C}' \mid \mathsf{proc}(c, t, P) \mid \mathsf{msg}(c, t, M)$$

The predicates $\mathsf{proc}(c, t, P)$ and $\mathsf{msg}(c, t, M)$ *provide* c. I stipulate that no two distinct processes or messages in a configuration provide the same channel c. Also recall that messages M are simply processes of a particular form and are typed as such. The possible messages (of which there is one for each type constructor) can be read of from the operational semantics. They are summarized here for completeness.

$$M \quad ::= \quad (c.k\ ;\ c \leftarrow c') \mid (c.k\ ;\ c' \leftarrow c) \mid \mathsf{close}\ c \mid (\mathsf{send}\ c\ d\ ;\ c' \leftarrow c) \mid (\mathsf{send}\ c\ d\ ;\ c \leftarrow c')$$

The typing judgment has the form $\Delta' \vDash \mathcal{C} :: \Delta$ meaning that if composed with a configuration that provides Δ', the result will provide Δ.

$$\frac{}{\Delta \vDash (\cdot) :: \Delta}\ \mathsf{empty} \qquad \frac{\Delta_0 \vDash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \vDash \mathcal{C}_2 :: \Delta_2}{\Delta_0 \vDash (\mathcal{C}_1\ \mathcal{C}_2) :: \Delta_2}\ \mathsf{compose}$$

To type processes and messages, I begin by considering *preservation*: I would like to achieve that if $\Delta' \vDash \mathcal{C} :: \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$ then still $\Delta' \vDash \mathcal{C}' :: \Delta$. Without the temporal modalities, this is guaranteed by the design of the sequent calculus: the right and left rules match just so that cut reduction (which is the basis for reduction in the operational semantics) leads to a well-typed deduction. The key here is what happens with time. Consider the special case of delay. When we transition from delay ; P to P we strip one $\bigcirc$ modality from Δ and A, but because we also advance time from t to $t+1$, the $\bigcirc$ modality is restored, keeping the interface type invariant.

When we also consider types $\Box A$ and $\Diamond A$ the situation is a little less straightforward because of their interaction with $\bigcirc$. I reuse the idea of the solution, allowing the subtraction of time from a type, possibly stopping when I meet a $\Box$ or $\Diamond$.

$$\begin{array}{lcllcl} [A]_L^{-0} & = & A & [A]_R^{-0} & = & A \\ [A]_L^{-(t+1)} & = & [[A]_L^{-t}]_L^{-1} & [A]_R^{-(t+1)} & = & [[A]_R^{-t}]_R^{-1} \end{array}$$

This is extended to channel declarations in the obvious way. Additionally, the imprecision of $\Box A$ and $\Diamond A$ may create temporal gaps in the configuration that need to be bridged by a weak form of subtyping $A <: B$,

$$\frac{m \leq n}{\bigcirc^m \Box A <: \bigcirc^n \Box A}\ \Box_{\mathsf{weak}} \qquad \frac{m \geq n}{\bigcirc^m \Diamond A <: \bigcirc^n \Diamond A}\ \Diamond_{\mathsf{weak}} \qquad \frac{}{A <: A}\ \mathsf{refl}$$

This relation is specified to be reflexive and clearly transitive. I extend it to contexts Δ in the obvious manner. In the final rules, I also account for some channels that are not used by P or M but just passed through.

$$\frac{\Delta' <: \Delta \quad [\Delta]_L^{-t} \vdash P :: [c : A]_R^{-t} \quad A <: A'}{\Delta_0, \Delta' \vDash \mathsf{proc}(c, t, P) :: (\Delta_0, c : A')}\ \mathsf{proc}$$

$$\frac{\Delta' <: \Delta \quad [\Delta]_L^{-t} \vdash M :: [c : A]_R^{-t} \quad A <: A'}{\Delta_0, \Delta' \vDash \mathsf{msg}(c, t, M) :: (\Delta_0, c : A')}\ \mathsf{msg}$$

4.3.2 Type Preservation

With the four rules for typing configurations (empty, compose, proc and msg), type preservation is relatively straightforward. We need some standard lemmas about being able to split a configuration and be able to move a provider (whether process or message) to the right in a typing derivation until it rests right next to its client. Regarding time shifts, we need the following properties.

Lemma 4.1 (Time Shift)**.**

(i) If $[A]_L^{-t} = [B]_R^{-t}$ *and both are defined then* $A = B$.

(ii) $[[A]_L^{-t}]_L^{-s} = [A]_L^{-(t+s)}$ *and if either side is defined, the other is as well.*

(iii) $[[A]_R^{-t}]_R^{-s} = [A]_R^{-(t+s)}$ *and if either side is defined, the other is as well.*

Theorem 4.2 (Type Preservation). *If* $\Omega' \vDash \mathcal{C} :: \Omega$ *and* $\mathcal{C} \mapsto \mathcal{D}$ *then* $\Omega' \vDash \mathcal{D} :: \Omega$.

Proof. By case analysis on the transition rule, applying inversion to the given typing derivation, and then assembling a new derivation of $\mathcal{D}$. □

Type preservation on basic session types is a simple special case of this theorem.

4.3.3 Global Progress

A process or message is said to be *poised* if it is trying to communicate along the channel that it provides. A poised process is comparable to a value in a sequential language. A configuration is poised if every process or message in the configuration is poised. Conceptually, this implies that the configuration is trying to communicate externally, i.e. along one of the channel it provides. The progress theorem then shows that either a configuration can take a step or it is poised. To prove this I show first that the typing derivation can be rearranged to go strictly from right to left and then proceed by induction over this particular derivation.

The question is how can we prove that processes are either at the same time (for most interactions) or that the message recipient is ready before the message arrives (for when?, now!, and some forwards)? The key insight here is in the following lemma.

Lemma 4.3 (Time Inversion).

(i) *If* $[A]_R^{-s} = [A]_L^{-t}$ *and either side starts with a basic session type constructor then* $s = t$.

(ii) *If* $[A]_L^{-t} = \Box B$ *and* $[A]_R^{-s} \neq \bigcirc(-)$ *then* $s \leq t$ *and* $[A]_R^{-s} = \Box B$.

(iii) *If* $[A]_R^{-t} = \Diamond B$ *and* $[A]_L^{-s} \neq \bigcirc(-)$ *then* $s \leq t$ *and* $[A]_L^{-s} = \Diamond B$.

Theorem 4.4 (Global Progress). *If* $\cdot \vDash \mathcal{C} :: \Omega$ *then either*

(i) $\mathcal{C} \mapsto \mathcal{C}'$ *for some* $\mathcal{C}'$, *or*

(ii) $\mathcal{C}$ *is poised.*

Proof. By induction on the right-to-left typing of $\mathcal{C}$ so that either $\mathcal{C}$ is empty (and therefore poised) or $\mathcal{C} = (\mathcal{D}\ \mathsf{proc}(c, t, P))$ or $\mathcal{C} = (\mathcal{D}\ \mathsf{msg}(c, t, M))$. By induction hypothesis, $\mathcal{D}$ can either take a step (and then so can $\mathcal{C}$), or $\mathcal{D}$ is poised. In the latter case, we analyze the cases for P and M, applying multiple steps of inversion and Lemma 4.3 to show that in each case either $\mathcal{C}$ can take a step or is poised. □

4.4 Further Examples

This section presents example analyses of some of the properties that we can express in the type system, such as the response time of concurrent data structures and the span of a fork/join parallel program.

In some examples I use parametric definitions, both at the level of types and processes. For example, stack_A describes stacks parameterized over a type A, $\mathsf{list}_A[n]$ describes lists of n elements, and $\mathsf{tree}[h]$ describes binary trees of height h. Process definitions are similarly parameterized. They exist as families of ordinary definitions and calculated accordingly, at the metalevel, which is justified since they are only implicitly quantified across whole definitions. This common practice (for example, in work on interaction nets [47]) avoids significant syntactic overhead, highlighting conceptual insight. It is of course possible to internalize such parameters (see, for example, work on refinement of session types [52] or explicitly polymorphic session types [29, 51]).

4.4.1 Response Times: Stacks and Queues

To analyze response times, I present concurrent stacks and queues. A stack data structure provides a client with a choice between a push and a pop. After a push, the client has to send an element, and the provider will again behave like a stack. After a pop, the provider will reply either with the label none and terminate (if there are no elements in the stack), or send an element and behave again like a stack. In the cost-free model, this is expressed in the following session type.

$$\begin{aligned}\mathsf{stack}_A = \&\{\, &\mathsf{push} : A \multimap \mathsf{stack}_A,\\ &\mathsf{pop} : \oplus\{\, \mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{stack}_A \,\}\,\}\end{aligned}$$

A stack is implemented as a chain of processes. The bottom to the stack is defined by the process *empty*, while a process *elem* holds a top element of the stack as well as a channel with access to the top of the remainder of the stack.

$$x : A, t : \mathsf{stack}_A \vdash \mathit{elem} :: (s : \mathsf{stack}_A)$$
$$\cdot \vdash \mathit{empty} :: (s : \mathsf{stack}_A)$$

The cost model I would like to consider here is $\mathcal{RS}$ where both receives and sends cost one unit of time. Because a receive costs one unit, every continuation type must be delayed by one tick of the clock, which is denoted by prefixing continuations by the $\bigcirc$ modality. This delay is not an artifact of the implementation, but an inevitable part of the cost model—one reason I have distinguished the synonyms tick (delay of one, due to the cost model) and delay (delay of one, to correctly time the interactions). In this section of examples I will make the same distinction

for the next-time modality: I write $\text{`}A$ for a step in time mandated by the cost model, and $\bigcirc A$ for a delay necessitated by a particular set of process definitions.

As a first approximation,

$$\begin{aligned}\mathsf{stack}_A = \&\{\ &\mathsf{push} : \text{`}(A \multimap \text{`}\mathsf{stack}_A),\\ &\mathsf{pop} : \text{`}\oplus\{\ \mathsf{none} : \text{`}\mathbf{1}, \mathsf{some} : \text{`}(A \otimes \text{`}\mathsf{stack}_A)\ \}\ \}\end{aligned}$$

There are several problems with this type. The stack is a data structure and has little or no control over *when* elements will be pushed onto or popped from the stack. Therefore a type $\Box\mathsf{stack}_A$ should be used to indicate that the client can choose the times of interaction with the stack. While the elements are held by the stack time advances in an indeterminate manner. Therefore, the elements stored in the stack must also have type $\Box A$, not A (so that they are always available).

$$\begin{aligned}\mathsf{stack}_A = \&\{\ &\mathsf{push} : \text{`}(\Box A \multimap \text{`}\Box\mathsf{stack}_A),\\ &\mathsf{pop} : \text{`}\oplus\{\ \mathsf{none} : \text{`}\mathbf{1}, \mathsf{some} : \text{`}(\Box A \otimes \text{`}\Box\mathsf{stack}_A)\ \}\ \}\end{aligned}$$

$$x : \Box A, t : \Box\mathsf{stack}_A \vdash \mathit{elem} :: (s : \Box\mathsf{stack}_A)$$
$$\cdot \vdash \mathit{empty} :: (s : \Box\mathsf{stack}_A)$$

This type expresses that the data structure is very efficient in its response time: there is no additional delay after it receives a push and then an element of type $\Box A$ before it can take the next request, and it will respond immediately to a pop request. It may not be immediately obvious that such an efficient implementation actually exists in the $\mathcal{RS}$ cost model, but it does. I use the implicit form from [39] omitting the tick constructs after each receive and send, and also the when? before each case that goes along with type $\Box A$.

```
s ← elem ← x t =
  case s  ( push ⇒ y ← recv s ;
                   s' ← elem ← x t ;        % previous top of stack, holding x
                   s ← elem ← y s'          % new top of stack, holding y
          | pop ⇒  s.some ;
                   send s x ;               % send channel x along s
                   s ← t )                  % s is now provided by t, via forwarding
s ← empty =
  case s  ( push ⇒ y ← recv s ;
                   e ← empty ;              % new bottom of stack
                   s ← elem ← y e
          | pop ⇒  s.none ;
                   close s )
```

The specification and implementation of a queue is very similar. The key difference in the implementation is that when a new element is received, it is passed along the chain of processes until it reaches the end. So instead of

$s' \leftarrow \mathit{elem} \leftarrow x\ t$; % *previous top of stack, holding* x
$s \leftarrow \mathit{elem} \leftarrow y\ s'$ % *new top of stack, holding* y

I write

t.enq ;
send $t\ y$; % *send* y *to the back of the queue*
$s \leftarrow \mathit{elem} \leftarrow x\ t$

in the push branch of *elem* process. These two send operations take two units of time, which must be reflected in the type: after a channel of type $\square A$ has been received, there is a delay of an additional two units of time before the provider can accept the next request.

$$\mathsf{queue}_A = \&\{\ \mathsf{enq} : \text{`}(\square A \multimap \text{`}\bigcirc\bigcirc\square\mathsf{queue}_A),\ \mathsf{deq} : \text{`}\oplus\{\ \mathsf{none} : \text{`}\mathbf{1}, \mathsf{some} : \text{`}(\square A \otimes \text{`}\square\mathsf{queue}_A)\ \}\ \}$$

$$x : \square A, t : \bigcirc\bigcirc\square\mathsf{queue}_A \vdash \mathit{elem} :: (s : \square\mathsf{queue}_A)$$
$$\cdot \vdash \mathit{empty} :: (s : \square\mathsf{queue}_A)$$

Time reconstruction will insert the additional delays in the *empty* process through subtyping, using $\square\mathsf{queue}_A \leq \bigcirc\bigcirc\square\mathsf{queue}_A$. I have syntactically expanded the tail call so the second use of subtyping is more apparent.

$s \leftarrow \mathit{empty} =$
  case s (enq $\Rightarrow$ $y \leftarrow$ recv s ; % $y : \square A \vdash s : \bigcirc\bigcirc\square\mathsf{queue}_A$
  $e \leftarrow \mathit{empty}$; % $y : \square A, e : \square\mathsf{queue}_A \vdash s : \bigcirc\bigcirc\square\mathsf{queue}_A$
  $s' \leftarrow \mathit{elem} \leftarrow y\ e$; % $\square\mathsf{queue}_A \leq \bigcirc\bigcirc\square\mathsf{queue}_A$ (on e)
  $s \leftarrow s'$ % $\square\mathsf{queue}_A \leq \bigcirc\bigcirc\square\mathsf{queue}_A$ (on s')
  | deq $\Rightarrow$ s.none ;
  close s)

The difference between the *response times* of stacks and queues in the cost model is minimal: both are constant, with the queue being two units slower. This is in contrast to the total work [38] which is constant for the stack but linear in the number of elements for the queue.

This difference in response times can be realized by typing clients of both stacks and queues. Compare clients S_n and Q_n that insert n elements into a stack and queue, respectively, send the result along channel d, and then terminate. I show only their type below, omitting the implementations.

$x_1 : \Box A, \ldots, x_n : \Box A, s : \Box\mathsf{stack}_A \vdash S_n :: (d : \bigcirc^{2n} (\Box\mathsf{stack}_A \otimes \text{`}\mathbf{1}))$
$x_1 : \Box A, \ldots, x_n : \Box A, s : \Box\mathsf{queue}_A \vdash Q_n :: (d : \bigcirc^{4n} (\Box\mathsf{queue}_A \otimes \text{`}\mathbf{1}))$

The types demonstrate that the total execution time of S_n is only $2n + 1$, while it is $4n + 1$ for Q_n. The difference comes from the difference in response times. Note that we can infer precise execution times, even in the presence of the $\Box$ modality in the stack and queue types.

4.4.2 Span Analysis: Trees

I use trees to illustrate an example that is typical for fork/join parallelism and computation of *span*. In order to avoid integers, I just compute the parity of a binary tree of height h with boolean values at the leaves. I do not show the obvious definition of *xor*, which in the $\mathcal{RS}$ cost model requires a delay of four from the first input.

$\mathsf{bool} = \oplus\{\, \mathsf{b0} : \text{`}\mathbf{1}, \mathsf{b1} : \text{`}\mathbf{1} \,\}$

$a : \mathsf{bool}, b : \bigcirc^2 \mathsf{bool} \vdash \mathit{xor} :: (c : \bigcirc^4 \mathsf{bool})$

In the definition of *leaf* and *node* I have explicated the delays inferred by time reconstruction, but not the tick delays. The type of $\mathsf{tree}[h]$ gives the *span* of this particular parallel computation as $5h + 2$. This is the time it takes to compute the parity under maximal parallelism, assuming that *xor* takes 4 cycles as shown in the type above.

$\mathsf{tree}[h] = \&\{\, \mathsf{parity} : \text{`}\bigcirc^{5h+2} \mathsf{bool} \,\}$

$\cdot \vdash \mathit{leaf} :: (t : \mathsf{tree}[h])$

$t \leftarrow \mathit{leaf} =$

```
  case t ( parity ⇒                 % · ⊢ t : ○^{5h+2} bool
                   % delay^{5h+2}   % · ⊢ t : bool
                   t.b0 ;           % · ⊢ t : 1
                   close t )
```

$l : \bigcirc^1\mathsf{tree}[h], r : \bigcirc^3 \mathsf{tree}[h] \vdash \mathit{node} :: (t : \mathit{tree}[h+1])$

$t \leftarrow \mathit{node} \leftarrow l\ r =$

```
  case t ( parity ⇒                 % l : tree[h], r : ○^2 tree[h] ⊢ t : ○^{5(h+1)+2} bool
                   l.parity ;       % l : ○^{5h+2} bool, r : ○^1 tree[h] ⊢ t : ○^{5(h+1)+1} bool
                   % delay          % l : ○^{5h+1} bool, r : tree[h] ⊢ t : ○^{5h+5} bool
                   r.parity ;       % l : ○^{5h} bool, r : ○^{5h+2} bool ⊢ t : ○^{5h+4} bool
                   % delay^{5h}     % l : bool, r : ○^2 bool ⊢ t : ○^4 bool
                   t ← xor ← l r )
```

The type $l : \bigcirc\,\mathsf{tree}[h]$ comes from the fact that, after receiving a parity request, it is first sent out the parity request to the left subtree l. The type $r : \bigcirc^3\,\mathsf{tree}[h]$ is determined from the delay

of 2 between the two inputs to *xor*. The magic number 5 in the type of tree was derived in reverse from setting up the goal of type-checking the *node* process under the constraints already mentioned. It can also be thought of as 4+1, where 4 is the time to compute the exclusive or at each level and 1 as the time to propagate the parity request down each level.

As is often done in abstract complexity analysis, I can also impose an alternative cost model. For example, I may count only the number of calls to *xor* while all other operations are cost free. Then I would have

$$a : \mathsf{bool}, b : \mathsf{bool} \vdash \mathit{xor} :: (c : \bigcirc\mathsf{bool}) \qquad \cdot \vdash \mathsf{leaf} :: (t : \mathsf{tree}[h])$$
$$\mathsf{tree}[h] = \&\{\, \mathsf{parity} : \bigcirc^h \mathsf{bool} \,\} \qquad l : \mathsf{tree}[h], r : \mathsf{tree}[h] \vdash \mathsf{node} :: (t : \mathsf{tree}[h+1])$$

with the same code but different times and delays from before. The reader is invited to reconstruct the details.

4.5 Related Work

Most closely related is work on space and time complexity analysis of interaction nets by Gimenez and Moser [47], which is a parallel execution model for functional programs. While also inspired by linear logic and, in particular, proof nets, it treats only special cases of the additive connectives and recursive types and does not have analogues of the $\Box$ and $\Diamond$ modalities. It also does not provide a general source-level programming notation with a syntax-directed type system. On the other hand it incorporates sharing and space bounds, which are beyond the scope of this work.

Session types and process calculi. Another related thread is the research on timed multiparty session types [24] for modular verification of real-time choreographic interactions. Their system is based on explicit global timing interval constraints, capturing a new class of communicating timed automata, in contrast to our system based on binary session types in a general concurrent language. Therefore, their system has no need for general $\Box$ and $\Diamond$ modalities, the ability to pass channels along channels, or the ability to identify channels via forwarding. Their work is complemented by an expressive dynamic verification framework in real-time distributed systems [83], which I do not consider. Semantics counting communication costs for work and span in session-typed programs were given by Silva et al. [94], but no techniques for analyzing them were provided.

In addition to the work on timed multiparty session types, time has been introduced into the π-calculus (see, for example, Saeedloei and Gupta [92]) or session-based communication primitives (see, for example, López et al. [76]) but generally these works do not develop a type system. Kobayashi [70] extends a (synchronous) π-calculus with means to count parallel reduction steps. He then provides a type system to verify time-boundedness. This is more general

in some dimension than our work because of a more permissive underlying type and usage system, but it lacks internal and external choice, genericity in the cost model, and provides bounds rather than a fine gradation between exact and indefinite times. Session types can also be derived by a Curry-Howard interpretation of *classical linear logic* [104] but I am not aware of temporal extensions. I conjecture that there is a classical version of our system where $\Box$ and $\Diamond$ are dual and $\bigcirc$ is self-dual.

Reactive programming. Synchronous data flow languages such as Lustre [54], Esterel [22], or Lucid Synchrone [88] are time-synchronous with uni-directional flow and thus may be compared to the fragment of our language with internal choice ($\oplus$) and the next-time modality ($\bigcirc A$), augmented with existential quantification over basic data values like booleans and integers (which we have omitted here only for the sake of brevity). The global clock would map to our underlying notion of time, but data-dependent local clocks would have to be encoded at a relatively low level using streams of option type, compromising the brevity and elegance of these languages. Furthermore, synchronous data flow languages generally permit sharing of channels, which, although part of many session-typed languages [19, 28], require further investigation with temporal modalities. On the other hand, I support a number of additional types such as external choice ($\&$) for bidirectional communication and higher-order channel-passing ($A \multimap B$, $A \otimes B$). In the context of functional reactive programming, a Nakano-style [82] temporal modality has been used to ensure productivity [72]. A difference in my work is that I consider concurrent processes and that the types prescribe the timing of messages.

Computational interpretations of $\bigcirc A$. A first computational interpretation of the next-time modality under a proofs-as-programs paradigm was given by Davies [41]. The basis is natural deduction for a (non-linear!) intutionistic linear-time temporal logic with only the next-time modality. Rather than capturing cost, the programmer could indicate *staging* by stipulating that some subexpressions should be evaluated "at the next time". The natural operational semantics then is a logically-motivated form of *partial evaluation* which yields a residual program of type $\bigcirc A$. This idea was picked up by Feltman et al. [42] to instead *split* the program statically into two stages where results from the first stage are communicated to the second. Again, neither linearity (in the sense of linear logic), nor any specific cost semantics appears in this work.

Other techniques. Inferring the cost of concurrent programs is a fundamental problem in resource analysis. Hoffmann and Shao [57] introduce the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. Their main limitation is that they can only handle parallel computation; they don't support message-passing or shared memory based concurrency. Blelloch and Reid-Miller [23] use pipelining [84] to improve the complexity of parallel algorithms. However, they use futures [55], a parallel language construct to implement pipelining without the programmer having to specify them

explicitly. The runtime of algorithms is determined by analyzing the work and depth in a language-based cost model. The work relates to ours in the sense that pipelines can have delays, which can be data dependent. However, the algorithms they analyze have no message-passing concurrency or other synchronization constructs. Albert et al. [13] devised a static analysis for inferring the parallel cost of distributed systems. They first perform a block-level analysis to estimate the serial cost, then construct a distributed flow graph (DFG) to capture the parallelism and then obtain the parallel cost by computing the maximal cost path in the DFG. However, the bounds they produce are modulo a points-to and serial cost analysis. Hence, an imprecise points-to analysis will result in imprecise parallel cost bounds. Moreover, since their technique is based on static analysis, it is not compositional and a whole program analysis is needed to infer bounds on each module. Recently, a bounded linear typing discipline [46] modeled in a semiring was proposed for resource-sensitive compilation. It was then used to calculate and control execution time in a higher-order functional programming language. However, this language did not support recursion.

4.6 Future Directions

I have presented a system of temporal session types that can accommodate and analyze concurrent programs with respect to a variety of different cost models. Types can vary in precision, based on desired and available information, and includes latency, rate, response time, and span of computations. It is constructed in a modular way, on top of a system of basic session types, and therefore lends itself to easy generalization. I have illustrated the type system through a number of simple programs on streams of bits, binary counters, lists, stacks, queues, and trees. I mention some of the further challenges that need to be addressed in this domain of temporal session types.

Inference Inference of time bounds will make resource-aware session types more practical and usable. The most severe difficulty here is the $\bigcirc$ operator. Computing the number of $\bigcirc$ operators to insert at a program point is non-trivial. The idea here would be similar to work inference. The inference engine first inserts a parametric amount of delay and then the type-checker determines the constraints on the inserted delays. A solver then tries to determine the value of said delays. The $\Box$ and $\Diamond$ operators do not involve any parameters and they should be easier to handle.

Dependent Types Time bounds are often dependent on the type refinements applied to session types. However, these dependencies bring along their own challenges. They require an arithmetic solver engine in the type checker, along with a system that allows parameters in types and process definitions. Moreover, they exacerbate the non-determinism in subtyping.

Chapter 5

Programming Digital Contracts

Digital contracts are computer protocols that describe and enforce the execution of a contract. With the rise of blockchains and cryptocurrencies such as Bitcoin [81], Ethereum [106], and Tezos [50], digital contracts have become popular in the form of smart contracts, which provide potentially distrusting parties with programmable money and an enforcement mechanism that does not rely on third parties. Smart contracts have been used to implement auctions [1], investment instruments [79], insurance agreements [67], supply chain management [75], and mortgage loans [78]. In general, digital contracts hold the promise to reduce friction, lower cost, and broaden access to financial infrastructure.

Smart contracts have not only shed light on the benefits of digital contracts but also on their potential risks. Like all software, smart contracts can contain bugs and security vulnerabilities [17], which can have direct financial consequences. A well-known example, is the attack on The DAO [79], resulting in a multi-million dollar theft by exploiting a contract vulnerability. Maybe even more important than the direct financial consequences is the potential erosion of trust as a result of such failures.

Contract languages today are derived from existing general-purpose languages like JavaScript (Ethereum's Solidity [1]), Go (in the Hyperledger project [27]), or OCaml (Tezos' Liquidity [3]). While this makes contract languages look familiar to software developers, it is inadequate to accommodate the domain-specific requirements of digital contracts.

- Instead of centering contracts on their interactions with users, the high-level protocol of the intended interactions with a contract is buried in the implementation code, hampering understanding, formal reasoning, and trust.
- Resource (or *gas*) usage of digital contracts is of particular importance for transparency and consensus. However, obliviousness of resource usage in existing contract languages makes it hard to predict the cost of executing a contract and prevent denial-of-service vulnerabilities.
- Existing languages fail to enforce linearity of assets, endangering the validity of a contract when assets get duplicated or deleted, accidentally or maliciously [77].

As a result, developing a correct smart contract is no easier than developing bug-free software in general. Additionally, vulnerabilities are harder to fix, because changes in the code may proliferate into changes in the contract itself.

This chapter presents the *type-theoretic foundations* of Nomos, a programming language for digital contracts whose genetics match the domain-specific requirements to provide strong static guarantees that facilitate the design of correct contracts. In particular, Nomos' type system makes explicit the protocols governing a contract, provides static bounds on the resource cost of interacting with a contract, and enforces a linear treatment of a contract's assets.

To express and enforce the protocols underlying a contract, Nomos is based on *resource-aware session types* [38]. The types describe the protocol of interaction between users and contracts and serve as a high-level description of the functionality of the contract. Type checking can be automated and guarantees that Nomos programs follow the given protocol. In this way, the key functionality of the contract is visible in the type, and contract development is centered on the interaction of the contract with the world. In addition to the interaction, resource-aware types also make the transaction cost visible in the type. This makes transactions transparent in their resource usage, and type checking again guarantees that the transactions do not exceed the resource usage prescribed by the type.

To eliminate a class of bugs in which the internal state of a contract loses track of its assets or performs unintended transactions, Nomos integrates a linear type system [103] into a functional language. Linear type systems use the ideas of Girard's linear logic [48] to ensure that certain data is neither duplicated nor discarded by a program. Programming languages such as Rust [6] have demonstrated that substructural type systems are practical in industrial-strength languages. Moreover, linear types are compatible with session types, which are themselves based on linear logic [19, 28, 86, 99, 104].

In addition to the design of the Nomos language, this chapter makes the following technical *contributions*.

1. Linear session types that support controlled sharing [19, 20] have been integrated into a conventional functional type system. To leave the logical foundation intact, the integration is achieved by a *contextual monad* [99] (Section 5.2) that gives process expressions first-class status in the functional language. Moreover, shared session types [19] are recast to accommodate the explicit notions of *contracts* and clients (Section 5.3).

2. I prove the type soundness of Nomos with respect to a novel asynchronous cost semantics using progress and preservation (Section 5.5).

3. I translated all examples used in this paper into Concurrent C0 [105], which serves as proof-of-concept for evaluating the performance of digital contract languages based on session types. Our preliminary results indicate that the performance of language based on session types is adequate for implementation of digital contracts (Section 5.6).

5.1 Nomos by Example

Nomos is a programming language based on resource-aware [38] and shared [19] session types for writing safe digital contracts. This section uses a simple auction contract to showcase the most significant features of the language.

Explicit Protocols of Interaction Digital contracts, like ordinary contracts, follow a predefined protocol. For instance, an auction contract follows the protocol that the bidders first submit their bids to the auctioneer, and then the highest bidder receives the lot while all other bidders receive their bids back. In existing smart contract languages like Solidity [1], this protocol is neither made explicit in the contract program nor enforced statically. Without such an explicit protocol, there is no guarantee that the parties involved in the contract will follow the protocol. As a result, contracts in these languages have to resort to explicit runtime checks to prevent undesirable behavior. This is a common source of bugs in contracts as accounting for all possible unwanted behavior is challenging, especially in a distributed system with distrusting parties.

Contracts in Nomos, on the other hand, are typed with a *session type* [19, 28, 64–66, 86, 99, 104], which specifies the contract's protocol of interaction. Type-checking then makes sure that the program implements the protocol defined by the session type correctly. For instance, consider the following protocol prescribed by the auction session type.

$$\begin{aligned}\mathsf{auction} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{11} \oplus \{\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction},\\ \mathsf{cancel} : \triangleright^{8} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction}\},\\ \mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{lot} \otimes \triangleright^{3} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction},\\ \mathsf{lost} : \mathsf{money} \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction}\},\\ \mathsf{cancel} : \triangleright^{8} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction}\}\}\end{aligned}$$

Since there exist multiple bidders in an auction, I use a *shared* session type [19] to define the auction protocol. To guarantee that bidders interact with the auction in mutual exclusion, the session type demarcates the parts of the protocol that become a *critical section*. The $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ type modality denotes the beginning of a critical section, the $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ modality its end. Programmatically, $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ translates into an *acquire* of the auction session and $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ into the *release* of the session. Shared session types guarantee that inside a critical section there exists exactly one client, whose interaction is described by a *linear* session type.

Once a client has acquired the auction session, the auction will indicate whether it is still running (running) or not (ended). This protocol is expressed by the internal choice type constructor ($\oplus$), describing the provider's (aka contract's) choice. An external choice ($\&$), on the other hand, leaves the choice to the client. For example, in case the auction is still running, the client can choose between placing a bid (bid) or backing out (cancel). If the client chooses to place a bid, they have to indicate their identifier, followed by a payment, after which they

```
1:  (b : bids) ; (M : money), (l : lot) ⊢
                                    run :: (sa : auction)
2:    sa ← run b ← M l =
3:      la ← accept sa ;
4:      la.running ;
5:      case la
6:        (bid ⇒ r ← recv la ;
7:                m ← recv la ;
8:                sa ← detach la ;
9:                m.value ;
10:               v ← recv m ;
11:               b′ = addbid b (r, v) ;
12:               M.add ;
13:               send M m ;
14:               sa ← run b′ ← M l)
15:       | cancel ⇒ sa ← detach la ;
16:                   sa ← run b ← M l
```

FIGURE 5.1: Auction process in running mode

release the session. Nomos session types allow exchange of both functional values (e.g. id), using the arrow ($\supset$) constructor, and linear values, using the lolli ($\multimap$) constructor. Using a linear type to represent digital money (money) makes sure that such a value can neither be duplicated nor lost. Should the auction have ended, the client can check whether they have won by providing their identifier. The auction will answer with either won or lost. In the former case, the auction will send the lot (commodity being auctioned, represented as a linear type), in the latter case, it will return the client's bid. The tensor ($\otimes$) constructor is the dual to $\multimap$ and denotes the exchange of linear value from the contract to the client.

Figure 5.1 implements the contract providing session type auction. In Nomos, session types are implemented by *processes*, revealing the concurrent, message-passing nature of session-typed languages. The implementation shows the process *run* representing the running auction. It first *accepts* an acquire request by a client (line 3) and then sends the message running (line 4) indicating the auction status. The process then waits for the clients choice. Should the client choose to make a bid, the process waits to receive the client's identifier (line 6) followed by money equivalent to the client's bid (line 7). After this linear exchange, the process leaves the critical section by issuing a *detach* (line 8), matching the client's release request. Internally, the process stores the pair of the client's identifier and their bid in the data structure bids (lines 9 - 11), and the total funds of the auction as a linear resource provided by channel M of type money (lines 12 - 13). The ended protocol of the contract is governed by a different process, responsible for distributing the bids back to the clients.

Re-Entrancy Vulnerabilities This condition is created when a client can call a contract function and potentially re-enter before the previous call is completed. In existing languages, transferring funds from the contract to the client also transfers execution control over to the

client, who can then call into the same transfer function recursively, eventually leading to all funds being transferred from the contract to the client. This vulnerability was exposed by the infamous DAO attack [79], where $60 million worth of funds were stolen. The message passing framework of session types eliminates this vulnerability. While session types provide multiple clients access to a contract, the acquire-release discipline ensures that clients interact with the contract in mutual exclusion and according to the protocol defined by the type.

Resource Cost Another important aspect of digital contracts is their *gas usage*. The state of all the contracts is stored on the *blockchain*, a distributed ledger which records the history of all transactions. Executing a new contract function and updating the blockchain state requires new blocks to be added to the blockchain. This is done by *miners* who charge a fee based on the *gas* usage of the function, indicating the cost of its execution. Precisely computing this cost is important because the sender of a transaction must pay this fee to the miners. If the sender does not pay the required fee, the function will be rejected by the miners.

Resource-aware session types [38] are adept for statically analyzing the resource cost of a process. They operate by assigning an initial potential to each process. This potential is consumed by each operation that the process executes or can be transferred between processes to share and amortize cost. The cost of each operation is defined by a cost model. Resource-aware session types express the potential as part of the session type, making the resource analysis static. For instance, in the auction contract, I can require the client to pay potential for the operations that the contract must execute, both while placing and collecting their bids. If the cost model assigns a cost of 1 to each contract operation, then the maximum cost of a session is 11 (taking the max of all branches). Thus, I require the client to send 11 units of potential at the start of a session.

The $\triangleleft$ type constructor prescribes that the client must send potential to the client. The amount of potential is marked as a superscript to $\triangleleft$. Thus, $\triangleleft^{11}$ in the auction type indicates that the client initiates the session by sending 11 units of potential, consumed by the contract during execution. Dually, the $\triangleright$ constructor prescribes that the contract must send potential to the client. This is used by the contract to return the leftover potential to the client at the end of session in each branch. For instance, in the cancel branch in the auction type, the contract returns 8 units of potential to the client using the $\triangleright^8$ type constructor. This is analogous to gas usage in smart contracts, where the sender initiates a transaction with some initial gas, and the leftover gas at the end of transaction is returned to the sender. If the cost model assigns a cost to each operation as equivalent to their gas cost, the total initial potential of a process plus the potential it receives during a session reflects the upper bound on the gas usage.

Linear Resources Nomos integrates a linear type system that tracks the resources stored in a process. The type system enforces that these resources are never duplicated or discarded, but only exchanged between processes. The type system forbids a process from terminating while

it stores any linear resource. For instance, in the auction contract, money and lot are treated as linear resources. Technically, such resources are handled in the same way as channels.

Bringing It All Together A main contribution of this chapter is combining these features in a single language while retaining type safety. I also introduce *modes* for each channel; i) P for purely linear channels (mode for M and l in auction), ii) S for a shared session type (mode for sa), iii) L for a shared session type in linear mode (mode for la), and iv) C for a channel offered by a client process. The mode of a channel assigns a specific typing judgment (described in Section 5.3) to the process offering it. This prevents the linear channels from forming a cycle (which would break linearity), and is crucial while proving preservation.

Remarkably, the rules for the standard session types are simply extended in the usual way to apply to Nomos. The non-trivial aspects are the addition of the functional layer, and shared session types, which will be the topic of the next sections.

5.2 Adding a Functional Layer

Digital contracts combine linear channels and coins with conventional data structures, such as integers, lists, or dictionaries to enable contracts to maintain state. For instance, the auction contract introduced in Section 5.1 contains a list of bids that is not treated linearly.

To reflect and track different classes of data in the type system, I take inspiration from prior work [86, 99] and incorporate processes into a functional core via a linear *contextual monad* that isolates session-based concurrency. To this end, I introduce a separate functional context to the typing of a process. The linear contextual monad encapsulates open concurrent computations, which can be passed in functional computations but also transferred between processes in the form of *higher-order processes*, providing a uniform integration of higher-order functions and processes.

The types are separated into a functional and concurrent part, mutually dependent on each other. The functional types τ are given by the type grammar below.

$$\begin{array}{rcl} \tau & ::= & \tau \to \tau \mid \tau + \tau \mid \tau \times \tau \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{list}^q_\tau \\ & \mid & \{A_P \leftarrow \overline{A_P}\}_P \mid \{A_S \leftarrow \overline{A_P}\}_S \mid \{A_C \leftarrow \overline{A_S}\ ;\ \overline{A}\}_C \end{array}$$

The types are standard, except for the potential annotation $q \in \mathbb{N}$ in list types, which I explain in Section 5.4, and the contextual monadic types in the last line, which are the topic of this section. The terms in the functional layer are standard. Figure 5.2 provides their grammar for completeness. I also define a standard type judgment for the functional part of the language.

$$\Psi \Vdash^p M : \tau \quad \text{term } M \text{ has type } \tau \text{ in functional context } \Psi$$

$$
\begin{array}{rcl}
M, N & ::= & \lambda x : \tau . M_x \mid M\ N \\
& \mid & l \cdot M \mid r \cdot M \mid \mathsf{case}\ M\ (l \hookrightarrow M_l, r \hookrightarrow M_r) \\
& \mid & \langle M, N \rangle \mid M \cdot l \mid M \cdot r \\
& \mid & n \mid \mathsf{true} \mid \mathsf{false} \\
& \mid & [] \mid M :: N \mid \mathsf{match}\ M\ ([] \to M_1, x :: xs \to M_2) \\
& \mid & \{c \leftarrow P_{c,\overline{a}} \leftarrow \overline{a}\} \mid \{c \leftarrow P_{c,\overline{a},\overline{d}} \leftarrow \overline{a}\ ;\ \overline{d}\}
\end{array}
$$

FIGURE 5.2: Standard expressions from the functional layer

Contextual Monad The main novelty in the functional types are the three type formers for contextual monads, denoting the type of a process expression. The type $\{A_P \leftarrow \overline{A_P}\}_P$ denotes a process offering a *purely linear* session type A_P and using the purely linear type $\overline{A_P}$. The corresponding introduction form in the functional language is the monadic value constructor $\{c_\mathsf{P} \leftarrow P \leftarrow \overline{d_\mathsf{P}}\}$, denoting a runnable process offering along channel c_P that uses channels $\overline{d_\mathsf{P}}$, all at mode P. The corresponding typing rule for the monad is

$$
\frac{\Delta = \overline{d_\mathsf{P} : D} \qquad \Psi\ ;\ \cdot\ ;\ \Delta \vdash^{g} P :: (x_\mathsf{P} : A)}{\Psi \Vdash^{g} \{x_\mathsf{P} \leftarrow P \leftarrow \overline{d_\mathsf{P}}\} : \{A \leftarrow \overline{D}\}_P}\ \{\}I_P
$$

The monadic *bind* operation implements process composition and acts as the elimination form for values of type $\{A_P \leftarrow \overline{A_P}\}_P$. The bind operation, written as $c_\mathsf{P} \leftarrow M \leftarrow \overline{d_\mathsf{P}}\ ;\ Q_c$, composes the process underlying the monadic value M, which offers along channel c_P and uses channels $\overline{d_\mathsf{P}}$, with Q_c, which uses c_P. The typing rule for the monadic bind is

$$
\frac{\begin{array}{c}\Delta = \overline{d_\mathsf{P} : D} \quad \Psi \curlyvee (\Psi_1, \Psi_2) \quad \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{D}\} \\ r = p + q \quad \Psi_2\ ;\ \cdot\ ;\ \Delta', (x_\mathsf{P} : A) \vdash^{g} Q :: (z_\mathsf{P} : C)\end{array}}{\Psi\ ;\ \cdot\ ;\ \Delta, \Delta' \vdash^{r} x_\mathsf{P} \leftarrow M \leftarrow \overline{d_\mathsf{P}}\ ;\ Q :: (z_\mathsf{P} : C)}\ \{\}E_{PP}
$$

The linear context is split between the monad M and continuation Q, enforcing linearity. Similarly, the potential in the functional context is split using the sharing judgment ($\curlyvee$), explained in Section 5.4. The shared context Γ is empty and will be discussed in Section 5.3. The effect of executing a bind is the spawn of the purely linear process corresponding to the monad M, and the parent process continuing with Q. The corresponding operational semantics rules are given as follows:

$$
\frac{N \Downarrow V \mid \mu}{\mathsf{proc}(d_\mathsf{P}, w, x_\mathsf{P} \leftarrow M \leftarrow \overline{a}\ ;\ Q) \ \mapsto\ \mathsf{proc}(d_\mathsf{P}, w + \mu, x_\mathsf{P} \leftarrow V \leftarrow \overline{a}\ ;\ Q)}\ \mathsf{step}
$$

$$
\begin{array}{c}
\mathsf{proc}(d_\mathsf{P}, w, x_\mathsf{P} \leftarrow \{x'_\mathsf{P} \leftarrow P_{x'_\mathsf{P}, \overline{y}} \leftarrow \overline{y}\} \leftarrow \overline{a}\ ;\ Q) \ \mapsto \\
\mathsf{proc}(c_\mathsf{P}, 0, P_{c_\mathsf{P}, \overline{a}}), \quad \mathsf{proc}(d_m, w, [c_\mathsf{P}/x_\mathsf{P}]Q)
\end{array}
$$

The second rule spawns the process P offering along a globally fresh channel c_P, and using channels $\overline{a}$. The continuation process Q acts as a client for this fresh channel c_P. The other two

$\boxed{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x : A)}$ Process P uses functional values in Ψ, and provides A along x.

$$\frac{\Delta = \overline{d : D} \qquad \Psi\ ;\ \cdot\ ;\ \Delta \vdash^{q} P :: (x_{\mathsf{S}} : A_S)}{\Psi \Vdash^{q} \{x_{\mathsf{S}} \leftarrow P \leftarrow \overline{d_{\mathsf{P}}}\} : \{A_S \leftarrow \overline{D}\}_S}\ \{\}I_S$$

$$\frac{\Gamma = \overline{a : A} \qquad \Delta = \overline{d : D} \qquad \Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x_{\mathsf{C}} : A)}{\Psi \Vdash^{q} \{x_{\mathsf{C}} \leftarrow P \leftarrow \overline{a}\ ;\ \overline{d}\} : \{A \leftarrow \overline{A}\ ;\ \overline{D}\}_C}\ \{\}I_C$$

$$\frac{\begin{array}{c} r = p + q \qquad \Delta = \overline{d_{\mathsf{P}} : D} \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \\ \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{D}\} \qquad \Psi_2\ ;\ \cdot\ ;\ \Delta', (x_{\mathsf{P}} : A) \vdash^{q} Q :: (z_m : C) \end{array}}{\Psi\ ;\ \cdot\ ;\ \Delta, \Delta' \vdash^{r} x_{\mathsf{P}} \leftarrow M \leftarrow \overline{d_{\mathsf{P}}}\ ;\ Q :: (z_m : C)}\ \{\}E_{P(m=S,L)}$$

$$\frac{\begin{array}{c} r = p + q \qquad \Delta = \overline{d_{\mathsf{P}} : D} \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \\ \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{D}\} \qquad \Psi_2\ ;\ \Gamma\ ;\ \Delta', (x_{\mathsf{P}} : A) \vdash^{q} Q :: (z_{\mathsf{C}} : C) \end{array}}{\Psi\ ;\ \Gamma\ ;\ \Delta, \Delta' \vdash^{r} x_{\mathsf{P}} \leftarrow M \leftarrow \overline{d_{\mathsf{P}}}\ ;\ Q :: (z_{\mathsf{C}} : C)}\ \{\}E_{PC}$$

$$\frac{\begin{array}{c} r = p + q \qquad \Delta = \overline{d_{\mathsf{P}} : D} \qquad (A_S, A_S)\ \mathsf{esync} \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \\ \Psi_1 \Vdash^{p} M : \{A_S \leftarrow \overline{D}\}_S \qquad \Psi_2\ ;\ \Gamma, (x_{\mathsf{S}} : A_S)\ ;\ \Delta' \vdash^{q} Q :: (z_{\mathsf{C}} : C) \end{array}}{\Psi\ ;\ \Gamma\ ;\ \Delta, \Delta' \vdash^{r} x_{\mathsf{S}} \leftarrow M \leftarrow \overline{d_{\mathsf{P}}}\ ;\ Q :: (z_{\mathsf{C}} : C)}\ \{\}E_{SC}$$

$$\frac{\begin{array}{c} r = p + q \qquad \Gamma \supseteq \overline{a_{\mathsf{S}} : A} \qquad \Delta = \overline{d : D} \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \\ \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{A}\ ;\ \overline{D}\}_C \qquad \Psi_2\ ;\ \Gamma\ ;\ \Delta', (x_{\mathsf{C}} : A) \vdash^{q} Q :: (z_{\mathsf{C}} : C) \end{array}}{\Psi\ ;\ \Gamma\ ;\ \Delta, \Delta' \vdash^{r} x_{\mathsf{C}} \leftarrow M \leftarrow \overline{a_{\mathsf{S}}}\ ;\ \overline{d}\ ;\ Q :: (z_{\mathsf{C}} : C)}\ \{\}E_{CC}$$

FIGURE 5.3: Introduction and elimination rules for the contextual monad.

monadic types correspond to spawning a shared process ($\{A_S \leftarrow \overline{A_P}\}_S$) and a client process ($\{A_C \leftarrow \overline{A_S}\ ;\ \overline{A}\}_C$) at mode S and C respectively. Their rules are analogous to $\{\}I_P$ and $\{\}E_{PP}$ and presented in Figure 5.3.

Value Communication Communicating a *value* of the functional language along a channel is expressed at the type level by adding the following two types.

$$A ::= \ldots \mid \tau \supset A \mid \tau \wedge A$$

The type $\tau \supset A$ prescribes receiving a value of type τ with continuation type A, while its dual $\tau \wedge A$ prescribes sending a value of type τ with continuation A. The corresponding typing rules for arrow ($\supset R, \supset L$) and product ($\wedge R, \wedge L$) are given in Figure 5.4. Receiving a value adds it to the functional context Ψ, while sending it requires proving that the value has type τ. Again, I defer the discussion of the resource annotation on the turnstile to later.

$$\boxed{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x : A)}$$ Process P uses functional values in Ψ, and provides A along x.

$$\frac{r = p + q \qquad \Psi \Vdash^{p} M : \tau \qquad \Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x_m : A)}{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{r} \mathsf{send}\ x_m\ M\ ;\ P :: (x_m : \tau \wedge A)}\ \wedge R$$

$$\frac{\Psi, (y : \tau)\ ;\ \Gamma\ ;\ \Delta, (x_m : A) \vdash^{q} Q :: (z_k : C)}{\Psi\ ;\ \Gamma\ ;\ \Delta, (x_m : \tau \wedge A) \vdash^{q} y \leftarrow \mathsf{recv}\ x_m\ ;\ Q :: (z_k : C)}\ \wedge L$$

$$\frac{\Psi, (y : \tau)\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x_m : B)}{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} y \leftarrow \mathsf{recv}\ x_m\ ;\ P :: (x_m : \tau \supset A)}\ \supset R$$

$$\frac{\begin{array}{c} r = p + q \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \qquad \Psi_1 \Vdash^{p} M : \tau \\ \Psi_2\ ;\ \Gamma\ ;\ \Delta, (x_m : A) \vdash^{q} Q :: (z_k : C) \end{array}}{\Psi\ ;\ \Gamma\ ;\ \Delta, (x_m : \tau \supset A) \vdash^{r} \mathsf{send}\ x_m\ M\ ;\ Q :: (z_k : C)}\ \supset L$$

FIGURE 5.4: Introduction and elimination rules for the contextual monad.

Tracking Linear Assets As an illustration, consider the protocol of interacting with the provider of nomisma, the cryptocurrency associated with the blockchain. Nomisma is used to store funds in the system, both in contracts and in clients, and can also be exchanged between them. The protocol is defined by the following type nomisma.

$$\begin{array}{lll} \mathsf{nomisma} = \&\{\mathsf{value} : \mathsf{int} \wedge \mathsf{nomisma}, & & \% \text{ send value} \\ \quad \mathsf{add} : \mathsf{nomisma} \multimap \mathsf{nomisma} & & \% \text{ receive money} \\ \quad \mathsf{subtract} : \mathsf{int} \supset \mathsf{nomisma} \otimes \mathsf{nomisma} & & \% \text{ receive int, send money} \\ \quad \mathsf{coins} : \mathsf{list}_{\mathsf{coin}}\} & & \% \text{ send list of coins} \end{array}$$

Nomisma acts as an abstraction over the funds stored. The *wallet* process implemented in Figure 5.5 provides this abstraction. It contains an integer n in its functional context (storing the integral value of the funds), a list of coins ($\mathsf{list}_{\mathsf{coin}}$) in its linear context (storing the actual funds), and provides a service of type nomisma. The type coin stands for a unit of currency and can be added to the type system as an abstract channel type that does not allow for interactions. Thus, there is no way to create or destroy a message or value depending on this type. The nomisma type prescribes the following protocol: querying for the value of the current funds (value), adding and subtracting to and from the current funds (add and subtract, resp.), and retrieving the list of coins constituting the current funds (coins). If the *wallet* process receives the message value, it sends back the integer n, and recurses (lines 4 and 5). If it receives the message add followed by a channel of type nomisma (line 6), it queries the value of the received nomisma m'_{P} (line 7), stores it in v (line 8), queries the coins stored in m'_{P} (line 9), and appends them to its internal list of coins (line 10). Similarly, if the *wallet* process receives the message subtract followed by an integer, it sends a channel of type nomisma corresponding to the subtracted funds. If the received integer n' is greater than the stored funds n (line 13), the

```
1:  (n : int) ; (l_P : list_coin) ⊢ wallet :: (m_P : nomisma)
2:    m_P ← wallet n ← l_P =
3:      case m_P
4:        (value ⇒ send m_P n ;
5:                  m_P ← wallet n ← l_P
6:        | add ⇒ m'_P ← recv m_P ;
7:                m'_P.value ;
8:                v ← recv m'_P ;
9:                m'_P.coins ;
10:               k_P ← append ← l_P m'_P ;
11:               m_P ← wallet (n + v) ← k_P
12:       | subtract ⇒ n' ← recv m_P ;
13:                    if (n' > n)
14:                      then raise "Insufficient funds!"
15:                    else
16:                      l'_P ← remove n' ← l_P ;
17:                      k_P ← recv l'_P ;
18:                      m'_P ← wallet n' ← k_P ;
19:                      send m_P m'_P ;
20:                      m_P ← wallet (n − n') ← l'_P
21:       | coins ⇒ m_P ← l_P)
```

FIGURE 5.5: Wallet process abstracting over the stored funds

wallet process raises an exception. Otherwise, it removes n' coins using the $remove$ process (line 16), creating a nomisma abstraction using the *wallet* process (line 18) and sending it over (line 19). Finally, if the *wallet* receives the message coins, it simply sends its internal list along the offered channel.

5.3 Sharing Contracts

Multi-user support is fundamental to digital contract development. Linear session types, as defined in Chapter 2, unfortunately preclude such sharing because they restrict processes to exactly one client; only one bidder for the auction, for instance (who will always win!). To support multi-user contracts, Nomos is based on *shared* session types [19]. Shared session types impose an acquire-release discipline on shared processes to guarantee that multiple clients interact with a contract in *mutual exclusion* of each other. When a client acquires a shared contract, it obtains a private linear channel along which it can communicate with the contract undisturbed by any other clients. Once the client releases the contract, it loses its private linear channel and only retains a shared reference to the contract.

A key idea of shared session types is to lift the acquire-release discipline to the type level. Generalizing the idea of type *stratification* [21, 86, 91], session types are stratified into a linear and shared layer with two *adjoint modalities* going back and forth between them:

$$
\begin{array}{rcl}
A_P & ::= & V \mid \oplus\{\ell : A_P\}_{\ell \in L} \mid \&\{l_i : A_P\}_{i \in I} \mid \mathbf{1} \\
& \mid & A_P \multimap A_P \mid A_P \otimes A_P \mid \tau \supset A_P \mid \tau \wedge A_P \\
\\
A_L & ::= & V \mid \oplus\{\ell : A_L\}_{\ell \in L} \mid \&\{l_i : A_L\}_{i \in I} \mid \mathbf{1} \\
& \mid & A_P \multimap A_L \mid A_P \otimes A_L \mid \tau \supset A_L \mid \tau \wedge A_L \mid \downarrow^{\mathsf{S}}_{\mathsf{L}} A_S \\
\\
A_S & ::= & \uparrow^{\mathsf{S}}_{\mathsf{L}} A_L \\
\\
A_C & ::= & A_P
\end{array}
$$

FIGURE 5.6: Grammar for shared session types

$$
\begin{array}{rcll}
A_S & ::= & \uparrow^{\mathsf{S}}_{\mathsf{L}} A_L & \text{shared session type} \\
A_L & ::= & \ldots \mid \downarrow^{\mathsf{S}}_{\mathsf{L}} A_S & \text{linear session types}
\end{array}
$$

The $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ type modality translates into an *acquire*, while the dual $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ type modality into a *release*.

Whereas mutual exclusion is one key ingredient to guarantee session fidelity (a.k.a. type preservation) for shared session types, the other key ingredient is the requirement that a session type is *equi-synchronizing*. A session type is equi-synchronizing if it imposes the invariant on a process to be released back to the same type at which the process was previously acquired, should it ever have been acquired.

Nomos integrates shared and linear session types with a functional language, yielding the following typing judgment:

$$\Psi \ ; \ \Gamma \ ; \ \Delta \vdash^{q} P :: (x : A)$$

This denotes a process P providing service of type A along channel x and using the functional variables in Ψ, shared channels from Γ and linear channels from Δ. The stratification of channels into layers arises from a difference in structural properties that exist for types at a mode. Shared propositions (mode S) exhibit weakening, contraction and exchange, thus can be discarded or duplicated, while linear propositions (mode $\mathsf{L}, \mathsf{C}, \mathsf{P}$) only exhibit exchange.

Allowing Shared Contracts to Rely on Linear Resources As exemplified by the auction contract, a digital contract typically amounts to a process that is shared at the outset, but oscillates between shared and linear to interact with clients, one at a time. Crucial for this pattern is the ability of a contract to maintain its linear resources (e.g., money) regardless of its mode. Unfortunately, current shared session types [19] do not allow a shared process to rely on any linear resources, requiring any linear resources to be consumed before becoming shared. This precaution is logically motivated [89] and also crucial for type preservation.

A key novelty of my work is to lift this restriction while *maintaining* type preservation. The main concern regarding type preservation is to prevent a process from acquiring its client, which would result in a cycle in the linear process tree. To this end, I factorize the above typing judgment according to the *three roles* that arise in digital contract programs: *contracts*,

clients, and *linear assets*. Since contracts are shared and thus can oscillate between shared and linear, the processes are governed by the following four typing judgments:

$$\begin{array}{lll} \Psi\ ;\ \cdot\ ;\ \Delta_{\mathsf{P}} & \vdash^{g} & P :: (x_{\mathsf{P}} : A_P) \\ \Psi\ ;\ \cdot\ ;\ \Delta_{\mathsf{P}} & \vdash^{g} & P :: (x_{\mathsf{S}} : A_S) \\ \Psi\ ;\ \cdot\ ;\ \Delta_{\mathsf{P}} & \vdash^{g} & P :: (x_{\mathsf{L}} : A_L) \\ \Psi\ ;\ \Gamma\ ;\ \Delta & \vdash^{g} & P :: (x_{\mathsf{C}} : A_C) \end{array}$$

The first typing judgment is for typing linear assets. These type a purely linear process P using a purely linear context Δ_{P} and offering type A_P along channel x_{P}. The mode P of the channel indicates that a purely linear session is offered. An example of such a process is the *wallet* process in Section 5.2. The second and third typing judgment are for typing contracts. The second judgment shows the type of a contract process P using a purely linear channel context Δ_{P} and offering type A_S on channel x_{S}. Once this shared channel is acquired by a client, the shared process transitions to its linear phase, whose typing is governed by the third judgment. The context remains purely linear Δ_{P}, while the offered channel transitions to shared linear mode L. Finally, the fourth typing judgment types a client process. This is the only judgment with a shared channel context Γ, allowing only a client process to acquire a contract process.

This factorization and the fact that contracts, as the only shared processes, do not have a shared channel context Γ, upholds preservation while allowing shared contract processes to rely on linear resources. Figure 5.6 shows the abstract syntax of types in Nomos.

Shared session types introduce new typing rules into the system, concerning the *acquire-release* constructs (see Figure 5.7). An acquire is applied to the shared channel x_{S} along which the shared process offers and yields a linear channel x_{L} when successful. A contract process can *accept* an acquire request along its offering shared channel x_{S}. After the accept is successful, the shared contract process transitions to its linear phase, now offering along the linear channel x_{L}.

The synchronous dynamics of the *acquire-accept* pair is

$$\begin{array}{l} \mathsf{proc}(a_{\mathsf{S}}, w', x_{\mathsf{L}} \leftarrow \mathsf{accept}\ a_{\mathsf{S}}\ ;\ P_{x_{\mathsf{L}}}), \mathsf{proc}(c_{\mathsf{C}}, w, x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ a_{\mathsf{S}}\ ;\ Q_{x_{\mathsf{L}}}) \ \mapsto \\ \mathsf{proc}(a_{\mathsf{L}}, w', P_{a_{\mathsf{L}}}), \mathsf{proc}(c_{\mathsf{C}}, w, Q_{a_{\mathsf{L}}}) \end{array}$$

This rule exploits the invariant that a contract process' providing channel a can come at two different modes, a linear one a_{L}, and a shared one a_{S}. The linear channel a_{L} is substituted for the channel variable x_{L} occurring in the process terms P and Q.

The dual to acquire-accept is *release-detach*. A client can *release* linear access to a contract process, while the contract process *detaches* from the client. The corresponding typing rules are presented in Figure 5.7. The effect of releasing the linear channel x_{L} is that the continuation Q loses access to x_{L}, while a new reference to x_{S} is made available in the shared context Γ.

$\boxed{\Psi \;;\; \Gamma \;;\; \Delta \vdash^{g} P :: (x : A)}$ Process P uses shared channels in Γ and offers A along x.

$$\frac{\Psi \;;\; \Gamma \;;\; \Delta, (x_{\mathsf{L}} : A_L) \vdash^{g} Q :: (z_{\mathsf{C}} : C)}{\Psi \;;\; \Gamma, (x_{\mathsf{S}} :\uparrow^{\mathsf{S}}_{\mathsf{L}} A_L) \;;\; \Delta \vdash^{g} x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ x_{\mathsf{S}} \;;\; Q :: (z_{\mathsf{C}} : C)} \uparrow^{\mathsf{S}}_{\mathsf{L}} L$$

$$\frac{\Psi \;;\; \cdot \;;\; \Delta \vdash^{g} P :: (x_{\mathsf{L}} : A_L)}{\Psi \;;\; \cdot \;;\; \Delta \vdash^{g} x_{\mathsf{L}} \leftarrow \mathsf{accept}\ x_{\mathsf{S}} \;;\; P :: (x_{\mathsf{S}} :\uparrow^{\mathsf{S}}_{\mathsf{L}} A_L)} \uparrow^{\mathsf{S}}_{\mathsf{L}} R$$

$$\frac{\Psi \;;\; \Gamma, (x_{\mathsf{S}} : A_S) \;;\; \Delta \vdash^{g} Q :: (z_{\mathsf{C}} : C)}{\Psi \;;\; \Gamma \;;\; \Delta, (x_{\mathsf{L}} :\downarrow^{\mathsf{S}}_{\mathsf{L}} A_S) \vdash^{g} x_{\mathsf{S}} \leftarrow \mathsf{release}\ x_{\mathsf{L}} \;;\; Q :: (z_{\mathsf{C}} : C)} \downarrow^{\mathsf{S}}_{\mathsf{L}} L$$

$$\frac{\Psi \;;\; \Gamma \;;\; \Delta \vdash^{g} P :: (x_{\mathsf{S}} : A_S)}{\Psi \;;\; \Gamma \;;\; \Delta \vdash^{g} x_{\mathsf{S}} \leftarrow \mathsf{detach}\ x_{\mathsf{L}} \;;\; P :: (x_{\mathsf{L}} :\downarrow^{\mathsf{S}}_{\mathsf{L}} A_S)} \downarrow^{\mathsf{S}}_{\mathsf{L}} R$$

FIGURE 5.7: Typing rules corresponding to the shared layer.

The contract, on the other hand, detaches from the client by transitioning its offering channel from linear mode x_{L} back to the shared mode x_{S}.

Operationally, the release-detach rule is inverse to the acquire-accept rule.

$$\mathsf{proc}(a_{\mathsf{L}}, w', x_{\mathsf{S}} \leftarrow \mathsf{detach}\ a_{\mathsf{L}} \;;\; P_{x_{\mathsf{S}}}), \mathsf{proc}(c_{\mathsf{C}}, w, x_{\mathsf{S}} \leftarrow \mathsf{release}\ a_{\mathsf{L}} \;;\; Q_{x_{\mathsf{S}}}) \;\mapsto\; \mathsf{proc}(a_{\mathsf{S}}, w', P_{a_{\mathsf{S}}}), \mathsf{proc}(c_{\mathsf{C}}, w, Q_{a_{\mathsf{S}}})$$

The auction contract introduced in Section 5.1 is an example of a shared type. A client of the auction contract is implemented in Figure 5.8 using the *bidder* process. The process holds access to the shared auction using channel a_{S}, and its linear assets (or nomisma) using channel N_{P}, along with its identifier and an integer corresponding to the amount it will bid. It first acquires the shared channel (line 3), and then branches based on the status of the auction. If the auction is running, the *bidder* sends the bid label, followed by its identifier (line 5). It then subtracts the bid amount from its assets (line 6), sends it to the auction (line 7). Finally, it releases the auction 8) and recurses (line 9). On the other hand, if the auction has ended, the *bidder* sends the collect label, followed by its identifier (line 10). The auction then responds with won or lost. If the *bidder* won the auction, it receives the lot (line 12), releases the auction (line 13) and terminates (line 14). While if it lost the auction, it receives its assets back (line 15), releases the auction (line 16), adds the received assets to its stored assets (line 17) and terminates (line 18).

5.4 Tracking Resource Usage

Resource usage is particularly important in digital contracts: Since multiple parties need to agree on the result of the execution of a contract, the computation is potentially performed

```
1:  (v : int), (r : id) ; (a_S : auction) ; (N_P : nomisma) ⊢
                          bidder :: (d_C : ⊕{won : lot, lost : nomisma})
2:    d_C ← bidder v r ← a_S ; N_P =
3:      a_L ← acquire a_S ;
4:      case a_L
5:        (running ⇒ a_L.bid ; send a_L r ;
6:                   N_P.subtract ; send N_P v ;
7:                   m_P ← recv N_P ; send a_L m_P ;
8:                   a_S ← release a_L ;
9:                   d_C ← bidder v r ← a_S ; N_P
10:       | ended ⇒ a_L.collect ; send a_L r ;
11:                 case a_L
12:                   (won ⇒ l_P ← recv a_L ;
13:                           a_S ← release a_L ;
14:                           d_C.won ; d_C ← l_P
15:                   | lost ⇒ m_P ← recv a_L ;
16:                           a_S ← release a_L ;
17:                           N_P.add ; send N_P m_P ;
18:                           d_C.lost ; d_C ← N_P))
```

FIGURE 5.8: A client of the auction process

multiple times or by a trusted third party. This immediately introduces the need to prevent denial of service attacks and to distribute the cost of the computation among the participating parties.

The predominant approach for smart contracts on blockchains like Ethereum is not to restrict the computation model but to introduce a cost model that defines the *gas* consumption of low level operations. Any transaction with a smart contract needs to be executed and validated before adding to the global distributed ledger, i.e., blockchain. This validation is performed by *miners*, who charge fees based on the gas consumption of the transaction. This fee has to be estimated and provided by the sender prior to the transaction. If the provided amount does not cover the gas cost, the money falls to the miner, the transaction fails, and the state of the contract is reverted back. Overestimates bare the risk of high losses if the contract has flaws or vulnerabilities.

It is not trivial to decide on the right amount for the fee since the gas cost of the contract does not only depend on the requested transaction but also on the (a priori unknown) state of the blockchain. Thus, precise and static estimation of gas cost facilitates transactions and reduces risks.

Functional Layer Numerous techniques have been proposed to statically derive resource bounds for functional programs [18, 32, 37, 73, 90]. Nomos adapts the work on automatic amortized resource analysis (AARA) [58, 61] that has been implemented in Resource Aware ML (RaML) [60]. RaML can automatically derive worst-case resource bounds for higher-order

```
1  let findbidh (b, r, acc, v) =
2    match b with
3    | [] -> (acc, v)
4    | hd::tl ->
5      tick(1.0);
6      let (addr, val) = hd in
7      if (addr = r)
8      then findbidh (tl, r, acc, (Some val))
9      else findbidh (tl, r, hd::acc, v)
10 let findbid (r, b) =
11   tick(1.0); findbidh (b, r, [], None)
```

FIGURE 5.9: A linear-time implementation of the function *findbid* from the auction contract

polymorphic programs with user-defined inductive types. The derived bounds are multivariate resource polynomials of the size parameters of the arguments. AARA is parametric in the resource metric and can deal with non-monotone resources like memory that can become available during the evaluation.

As an illustration, consider the function *findbid* in Figure 5.9. The argument b : bids can store the bids as a list of pairs of integers. The first component of a pair stores the identifier and the second component stores the bid. The function *findbid* searches the list for the identifier r and returns the corresponding bid value v and the list of bits without the pair (r, v) (to prevent bidders to withdraw twice).

I use *tick* annotations to define the resource usage of an expression in this article. I have annotated the code in Figure 5.9 to count the number of function calls. So the resource usage of an evaluation of *findbid* b r is $|b| + 1$.

The idea of AARA is to decorate base types with potential annotations that define a potential function as in amortized analysis. The typing rules ensure that the potential before evaluating an expression is sufficient to cover the cost of the evaluation and the potential defined by the return type. This posterior potential can then be used to pay for resource usage in the continuation of the program. For instance,the following is a resource-annotated type for *findbid*.

$$\mathit{findbid} : L^1(\mathsf{int} \times \mathsf{int}) \times \mathsf{int} \xrightarrow{1/0} L^0(\mathsf{int} \times \mathsf{int}) \times \mathsf{int\ option}$$

The type $L^1(\mathsf{int} \times \mathsf{int})$ assigns a unit potential to each element in the first argument of the function. The return value has no potential and thus has type $L^0(\mathsf{int} \times \mathsf{int})$. The annotation on the function arrow indicates that we need a unit potential to call the function and that no constant potential is left after the function call has returned.

In a larger program, we might want to call the function *findbid* again on the result of a call to the function. In this case, we would need to assign the type $L^1(\mathsf{int} \times \mathsf{int})$ to the resulting list and require $L^2(\mathsf{int} \times \mathsf{int})$ for the argument. In general, the type for the function can be

described with symbolic annotations with linear constraints between them. To derive a worst-case bound for a function the constraints can be solved by an off-the-shelf LP solver, even if the potential functions are polynomial. [58, 60].

Nomos simply adopts the standard typing judgment of AARA for functional programs.

$$\Psi \Vdash^{q} M : \tau$$

It states that under the resource-annotated, functional, context Ψ, with constant potential q, the expression M has the resource-aware type τ.

The operational *cost* semantics is defined by the judgment

$$M \Downarrow V \mid \mu$$

which states that the closed expression M evaluates to the value V with cost μ. The type soundness theorem states that if $\cdot \Vdash^{q} M : \tau$ and $M \Downarrow V \mid \mu$ then $q \geq \mu$.

Process Layer To bound resource usage of a process, Nomos features resource-aware session types (Chapter 3) for work analysis. The typing judgment for Nomos processes then becomes

$$\Psi \;;\; \Gamma \;;\; \Delta \vdash^{q} P :: (x_m : A)$$

The non-negative number q in the judgment denotes the potential that is stored in the process. Figure 5.10 shows the rules that interact with the potential annotations. In the rule $\triangleleft R$, process P storing potential q receives r units along the offered channel x_m using the *get* construct and the continuation executes with $p = q + r$ units of potential. In the dual rule $\triangleleft L$, a process storing potential $q = p + r$ sends r units along the channel x_m in its context using the *get* construct, and the continuation remains with p units of potential. The typing rules for the dual constructor $\triangleleft^{r} A$ are the exact inverse. Finally, executing the tick (r) construct consumes r potential from the stored process potential q, and the continuation remains with $p = q - r$ units, as described in the tick rule in Figure 5.10.

Integration Since both AARA for functional programs and resource-aware session types are based on the integration of the potential method into their type systems, their combination is natural. The two points of integration of the functional and process layer are (i) spawning a process, and (ii) sending/receiving a value from the functional layer. Recall the spawn rule $\{\}E_{PP}$ from Figure 5.3.

$$\frac{\begin{array}{c}\Delta = \overline{d_{\mathsf{P}} : D} \quad \Psi \curlyvee (\Psi_1, \Psi_2) \quad \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{D}\} \\ r = p + q \quad \Psi_2 \;;\; \cdot \;;\; \Delta', (x_{\mathsf{P}} : A) \vdash^{q} Q :: (z_{\mathsf{P}} : C)\end{array}}{\Psi \;;\; \cdot \;;\; \Delta, \Delta' \vdash^{r} x_{\mathsf{P}} \leftarrow M \leftarrow \overline{d_{\mathsf{P}}} \;;\; Q :: (z_{\mathsf{P}} : C)}\ \{\}E_{PP}$$

$\boxed{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x : A)}$ Process P has potential q and provides type A along channel x.

$$\frac{q = p + r \qquad \Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{p} P :: (x_m : A)}{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} \mathsf{pay}\ x_m\ \{r\}\ ;\ P :: (x_m : \triangleright^{r} A)}\ \triangleright R$$

$$\frac{p = q + r \qquad \Psi\ ;\ \Gamma\ ;\ \Delta, (x_m : A) \vdash^{p} P :: (z_k : C)}{\Psi\ ;\ \Gamma\ ;\ \Delta, (x_m : \triangleright^{r} A) \vdash^{q} \mathsf{get}\ x_m\ \{r\}\ ;\ P :: (z_k : C)}\ \triangleright L$$

$$\frac{p = q + r \qquad \Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{p} P :: (x_m : A)}{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} \mathsf{get}\ x_m\ \{r\}\ ;\ P :: (x_m : \triangleleft^{r} A)}\ \triangleleft R$$

$$\frac{q = p + r \qquad \Psi\ ;\ \Gamma\ ;\ \Delta, (x_m : A) \vdash^{p} P :: (z_k : C)}{\Psi\ ;\ \Gamma\ ;\ \Delta, (x_m : \triangleleft^{r} A) \vdash^{q} \mathsf{pay}\ x_m\ \{r\}\ ;\ P :: (z_k : C)}\ \triangleleft L$$

$$\frac{q = p + r \qquad \Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{p} P :: (x_m : A)}{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} \mathsf{tick}\ (r)\ ;\ P :: (x_m : A)}\ \mathsf{tick}$$

FIGURE 5.10: Typing rules corresponding to potential.

A process storing potential $r = p + q$ can spawn a process corresponding to the monadic value M, if M needs p units of potential to evaluate, while the continuation needs q units of potential to execute. Moreover, the functional context Ψ is shared in the two premises as Ψ_1 and Ψ_2 using the judgment $\Psi \curlyvee (\Psi_1, \Psi_2)$. This judgment, already explored in prior work [60] describes that the base types in Ψ is copied to both Ψ_1 and Ψ_2, but the potential is split up. For instance, $L^{q_1+q_2}(\tau) \curlyvee (L^{q_1}(\tau), L^{q_2}(\tau))$. Recalling the $\wedge R$ rule,

$$\frac{r = p + q \qquad \Psi \Vdash^{p} M : \tau \qquad \Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x_m : A)}{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{r} \mathsf{send}\ x_m\ M\ ;\ P :: (x_m : \tau \wedge A)}\ \wedge R$$

A process storing $r = p + q$ units of potential can send an expression M if evaluating M needs p units of potential, while the continuation needs q units of potential. Thus, the combination of the two type systems is smooth, assigning a uniform meaning to potential, both for the functional and process layer.

Operational Cost Semantics The resource usage of a process (or message) is tracked in semantic objects $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$ using the local counters w. This signifies that the process P (or message M) has performed *work* w so far. The rules of semantics that explicitly affect the work counter are

$$\frac{N \Downarrow V \mid \mu}{\mathsf{proc}(c_m, w, P[N]) \ \mapsto\ \mathsf{proc}(c_m, w + \mu, P[V])}\ \mathsf{internal}$$

This rule describes that if an expression N evaluates to V with cost μ, then the process $P[N]$ depending on monadic expression N steps to $P[V]$, while the work counter increments by μ, denoting the total number of internal steps taken by the process. At the process layer, the work increments on executing a *tick* operation.

$$\mathsf{proc}(c_m, w, \mathsf{tick}\ (\mu)\ ;\ P)\ \mapsto\ \mathsf{proc}(c_m, w + \mu, P)$$

A new process is spawned with $w = 0$, and a terminating process transfers its work to the corresponding message it interacts with before termination, thus preserving the total work performed by the system.

5.5 Type Soundness

The main theorems that exhibit the connections between our type system and the operational cost semantics are the usual *type preservation* and *progress*. First, I formalize the process typing judgment, $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{g} P :: (x_m : A)$ which is separated into 4 different categories, depending on the mode m. This mode asserts certain well-formedness conditions on the typing judgment. Remarkably, the process typing rules, despite being parametric in the mode, preserve these well-formedness conditions.

Lemma 5.1 (Invariants)**.** *The typing rules on the judgment* $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{g} (x_m : A)$ *preserve the following invariants i.e., if the conclusion satisfies the invariant, so do all the premises.* $\mathbf{L}(A)$ *denotes the language generated by the grammar of* A*.*

- *If* $m = \mathsf{P}$*, then* Γ *is empty and* $d_k \in \Delta \implies k = \mathsf{P}$ *for all* d_k *and* $A \in \mathbf{L}(A_P)$*.*
- *If* $m = \mathsf{S/L}$*, then* Γ *is empty and* $d_k \in \Delta \implies k = \mathsf{P}$ *for all* d_k *and* $A \in \mathbf{L}(A_S)$ *or* $A \in \mathbf{L}(A_L)$*.*
- *If* $m = \mathsf{C}$*, then* $A \in \mathbf{L}(A_C)$*.*

Configuration Typing At run-time, a program evolves into a number of processes and messages, represented by proc and msg predicates. This multiset of predicates is referred to as a *configuration* (abbreviated as Ω).

$$\Omega ::= \cdot \mid \Omega, \mathsf{proc}(c, w, P) \mid \Omega, \mathsf{msg}(c, w, M)$$

where $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$ are said to offer along channel c. A key question then is how to type these configurations. A configuration both uses and provides a collection of channels. The typing imposes a partial order among the processes and messages, requiring the provider of a channel to appear to the left of its client. I stipulate that no two distinct processes or messages in a well-formed configuration provide the same channel c.

$\boxed{\Sigma\ ;\ \Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta)}$ Configuration Ω provides shared channels Γ and linear channels Δ.

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

$$\frac{\Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta'_{\mathsf{P}}) \qquad \cdot\ ;\ \cdot\ ;\ \Delta'_{\mathsf{P}} \vdash^{q} P :: (x_{\mathsf{P}} : A_P)}{\Gamma_S \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{P}}, w, P) :: (\Gamma\ ;\ \Delta, (x_{\mathsf{P}} : A_P))}\ \mathsf{proc_P}$$

$$\frac{(x_{\mathsf{S}} : A_S) \in \Gamma_S \qquad (A_S, A_S)\ \mathsf{esync} \qquad \Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta'_{\mathsf{P}}) \qquad \cdot\ ;\ \cdot\ ;\ \Delta'_{\mathsf{P}} \vdash^{q} P :: (x_{\mathsf{S}} : A_S)}{\Gamma_S \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{S}}, w, P) :: (\Gamma, (x_{\mathsf{S}} : A_S)\ ;\ \Delta)}\ \mathsf{proc_S}$$

$$\frac{(x_{\mathsf{S}} : A_S) \in \Gamma_S \qquad (A_L, A_S)\ \mathsf{esync} \qquad \Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta'_{\mathsf{P}}) \qquad \cdot\ ;\ \cdot\ ;\ \Delta'_{\mathsf{P}} \vdash^{q} P :: (x_{\mathsf{L}} : A_L)}{\Gamma_S \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{L}}, w, P) :: (\Gamma, (x_{\mathsf{S}} : A_S)\ ;\ \Delta, (x_{\mathsf{L}} : A_L))}\ \mathsf{proc_L}$$

$$\frac{\Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta') \qquad \cdot\ ;\ \Gamma\ ;\ \Delta' \vdash^{q} P :: (x_{\mathsf{C}} : A_C)}{\Gamma_S \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{C}}, w, P) :: (\Gamma\ ;\ \Delta, (x_{\mathsf{C}} : A_C))}\ \mathsf{proc_C}$$

$$\frac{\Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta') \qquad \cdot\ ;\ \cdot\ ;\ \Delta' \vdash^{q} M :: (x_m : A)}{\Gamma_S \overset{E+q+w}{\vDash} \Omega, \mathsf{msg}(x_m, w, M) :: (\Gamma\ ;\ \Delta, (x_m : A))}\ \mathsf{msg}$$

FIGURE 5.11: Configuration typing rules.

The typing judgment for configurations has the form $\Sigma\ ;\ \Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta)$ defining a configuration Ω providing shared channels in Γ and linear channels in Δ. Additionally, we need to track the mapping between the shared channels and linear channels offered by a contract process, switching back and forth between them when the channel is acquired or released respectively. This mapping is stored in Γ_S. E is a natural number and stores the sum of the total potential and work as recorded in each process and message. I call E the energy of the configuration. Figure 5.11 presents the formal rules for typing a configuration.

Finally, Σ denotes a signature storing the type and function definitions. A signature is well-formed if a) every type definition $V = A_V$ is *contractive* [45] and b) every function definition $f = M : \tau$ is well-typed according to the expression typing judgment $\Sigma\ ;\ \cdot \Vdash^{p} M : \tau$. The signature does not contain process definitions; any process is encapsulated inside a function using the contextual monad.

Theorem 5.2 (Type Preservation)**.**

- *If a closed well-typed expression* $\cdot \Vdash^{q} N : \tau$ *evaluates to a value, i.e.,* $N \Downarrow V \mid \mu$*, then* $q \geq \mu$ *and* $\cdot \Vdash^{q-\mu} V : \tau$*.*
- *Consider a closed well-formed and well-typed configuration* Ω *such that* $\Sigma \; ; \; \Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta)$*. If the configuration takes a step, i.e.* $\Omega \mapsto \Omega'$*, then there exist* Γ'_S, Γ' *such that* $\Sigma \; ; \; \Gamma'_S \overset{E}{\vDash} \Omega' :: (\Gamma' \; ; \; \Delta)$*, i.e., the resulting configuration is well-typed. Additionally,* $\Gamma_S \subseteq \Gamma'_S$ *and* $\Gamma \subseteq \Gamma'$*.*

The preservation theorem is standard for expressions [60]. For processes, the proof proceeds by induction on the operational cost semantics and inversion on the process typing judgment.

A process $\mathsf{proc}(c_m, w, P)$ is poised if it is receiving a message on c_m. Dually, a message $\mathsf{msg}(c_m, w, M)$ is poised if it is sending along c_m. A configuration is poised if every message or process in the configuration is poised. Intuitively, this means that the configuration is trying to interact with the outside world along a channel in Γ or Δ. Additionally, a client process can be blocked if it is trying to acquire a contract process, which has already been acquired by some other client process. This can lead to the possibility of deadlocks.

Theorem 5.3 (Progress)**.** *Consider a closed well-formed and well-typed configuration* Ω *such that* $\Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta)$*. Either* Ω *is poised, or it can take a step, i.e.,* $\Omega \mapsto \Omega'$*, or some process in* Ω *is blocked along* a_{S} *for some shared channel* a_{S} *and there is a process* $\mathsf{proc}(a_{\mathsf{L}}, w, P) \in \Omega$*.*

The progress theorem is weaker than that for binary session types, where progress guarantees deadlock freedom due to absence of shared channels.

5.6 Preliminary Evaluation

To evaluate the practicality of Nomos, I implemented several digital contracts in Concurrent C0 [19, 105], a type-safe C-like imperative language with session types. The implementation uses an asynchronous semantics, and supports functional variables, shared and linear channels. Concurrent C0 does not have the type-theoretic foundation of Nomos and, in particular, does not support potential annotations on types. However, the language is similar enough to Nomos to draw conclusions about the efficiency of its session-type based approach, especially since the implemented contracts are not imperative.

Concurrent C0 programs are compiled to C. Each process is implemented as an operating system thread, as provided by the pthread library. Message passing is implemented via shared memory. Therefore, each channel is a shared data structure transitioning between linear and shared phases.

Contracts In the evaluation I used three different contract implementations, namely *auction*, *voting* and *bank account*. The auction contract was already introduced in Section 5.1.

The voting contract provides a ballot type.

$$\begin{aligned}\mathsf{ballot} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{11} \oplus \{&\mathsf{open} : \mathsf{id} \supset \oplus\{\mathsf{vote} : \mathsf{id} \supset \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{ballot},\\ &\qquad\qquad \mathsf{novote} : \triangleright^{3} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{ballot}\},\\ &\mathsf{closed} : \mathsf{id} \wedge \triangleright^{8} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{ballot}\}\end{aligned}$$

After a voter acquires the channel typed ballot, the contract sends a message open if the election is still open to voting and the voter responds with their id, which is verified by the contract. If the verification is successful, the contract sends vote, the voter replies with the id of the candidate they are voting for, and the session terminates. If the verification fails, the contract sends novote and terminates the session. If the election is over, the contract sends closed followed by the id of the winner of the election, and terminates the session.

The banking contract provides an account type, as follows.

$$\begin{aligned}\uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{13} \,\&\, \{&\mathsf{signup} : \mathsf{id} \supset \mathsf{pwd} \supset \triangleright^{5} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{account},\\ &\quad \mathsf{login} : \mathsf{id} \supset \mathsf{pwd} \supset \oplus\{\mathsf{failure} : \triangleright^{7} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{account},\\ &\qquad\qquad \mathsf{success} : \&\{\mathsf{deposit} : \mathsf{money} \multimap \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{account},\\ &\qquad\qquad\qquad \mathsf{balance} : \mathsf{int} \wedge \triangleright^{4} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{account},\\ &\qquad\qquad\qquad \mathsf{withdraw} : \mathsf{int} \supset \mathsf{money} \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{account}\}\}\}\end{aligned}$$

After acquiring the contract, a customer has the choice to create a new account by sending signup, followed by their id and password, and terminating. Or the customer can log in by sending login, followed by their id and password. The contract then authenticates the account, replying with failure and terminating the session, or success, depending on the authentication status. On receiving success, a customer has the choice of making a deposit, checking the balance, or making a withdraw and receiving money.

Evaluation I evaluate the execution time of each contract, varying number of transactions. Figure 5.12 plots the execution time in seconds (y-axis) vs the size of the input (x-axis). The experiments were performed on an Intel Core i5-5250U processor with 16 GB of main memory.

- Auction : I use the total number of bidders in the auction as input, varying them from 100 to 5000 in steps of 100. Each client first places a bid, followed by the winner of the auction being declared, and finally each client collects either the lot or their money.
- Voting : I use the total number of voters as the input size, varying them from 200 to 10000 in steps of 200. Each voter sends their vote to the contract, and at the end, the winner of the election is determined.

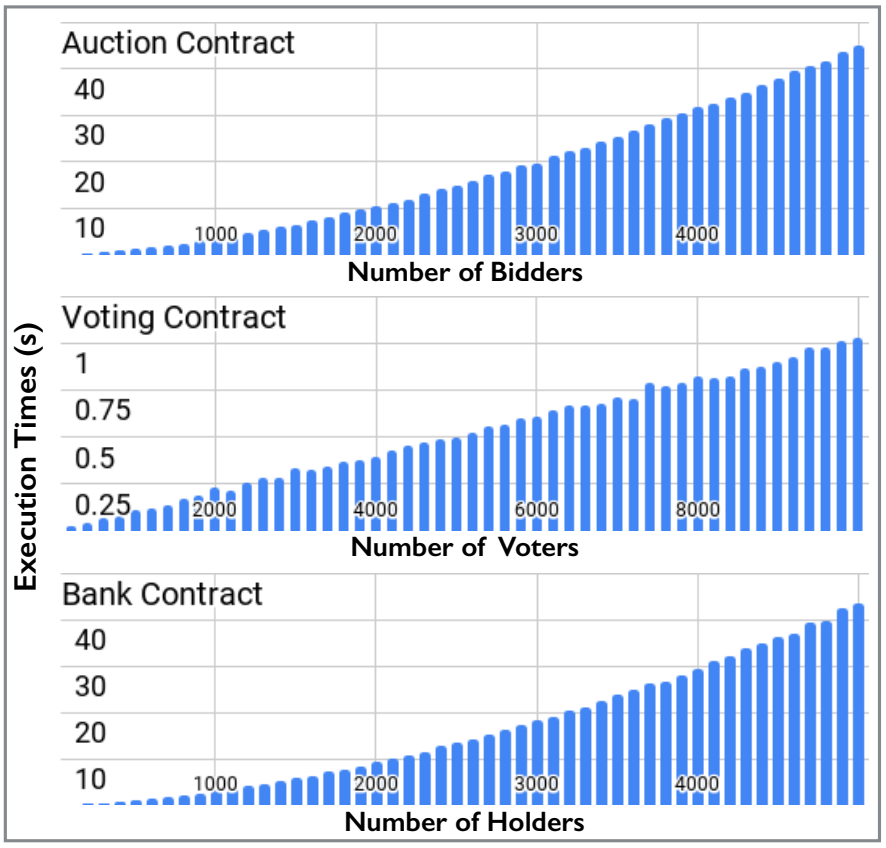

FIGURE 5.12: Contract execution times

- Bank : I use the total number of account holders as input size, varying them from 100 to 5000 in steps of 100. Each client first creates their account, then makes a deposit followed by a withdrawal, and finally checks their balance.

The execution times of the experiments vary between a few milliseconds to about 40 seconds. The data leads to the conclusion that the average time of one individual session with a contract varies from 0.12 ms (voting) to 4.5 ms (auction). The execution times for the auction and banking are significantly higher than the one of the voting protocol. This is because the former contracts involve exchange of funds. In concurrent C0, funds are represented by actual coin processes, resulting in a large number of processes in the system. In the Nomos implementation, I will introduce optimizations for the built-in coin type that does not spawn a process for every coin. As a result, the performance of the auction and bank-account would be similar to the voting contract.

5.7 Related Work

Existing smart contracts on Ethereum are predominantly implemented in Solidity [1], a statically typed object-oriented language influenced by Python and Javascript. Contracts in Solidity are similar to classes containing state variables and function declarations. However,

the language provides no information about the resource usage of a contract. Languages like Vyper [7] address resource usage by disallowing recursion and infinite-length loops, thus making estimation of gas usage decidable. However, both languages still suffer from re-entrancy vulnerabilities. Bamboo [2], on the other hand, makes state transitions explicit and avoids re-entrance by design. However, none of these languages describe and enforce communication protocols statically.

Domain specific languages have also been designed for other blockchains apart from Ethereum. Rholang [5] is formally modeled by the ρ-calculus, a reflective higher-order extension of the π-calculus. Michelson [4] is a purely functional stack-based language that has no side effects. Liquidity [3] is a high-level language that complies with the security restrictions of Michelson. Scilla [93] is an intermediate-level language where contracts are structured as communicating automata providing a continuation-passing style computational model to the language semantics. In contrast to this work, none of these languages use linear type systems to track assets stored in a contract.

Session types have been integrated into a functional language in prior work [99]. However, this integration does not account for resource usage, nor sharing. Similarly, shared session types [19] have previously not been integrated with a functional layer or tracked for resource usage. Moreover, existing shared session types [19] disallow shared processes to rely on any linear resources, a restriction we lift in Nomos. Resource usage has previously been explored separately for a functional language [60] and the process layer [38], but the two have never been integrated together.

5.8 Future Directions

Contract-to-Contract Communication The type theoretic foundations of Nomos does not allow direct contract-to-contract communication. This is because contracts are treated as special processes that can only depend on purely linear channels. Thus, contracts are not allowed to contain a reference to another shared channel. Contracts can have a proxy client through which they can indirectly communicate. It can be proved through a counter example that simply allowing contracts to interact compromises type preservation. I would like to explore how this restriction can be lifted without compromising type safety.

Error Handling Currently, the strong safety guarantees of Nomos only hold when both contracts and clients are well-typed. This raises the concern of how contracts implemented in Nomos can interact with clients that are not implemented in Nomos. Currently, the contracts do not have error handling mechanisms. One approach here is to introduce runtime monitoring. These monitors are wrapped around the contracts and observe the type of the data that is being exchanged on the channels. If they catch a type mismatch, they force the contract to safely return to a well-formed state and resume interaction with other clients.

Chapter 6

Implementation: Resource-Aware Session Types
Proposed Work

My main goal in this thesis proposal is evaluating and developing the *usability* of resource-aware session types, and their practicality with regard to digital contracts. As a first step, this requires a sound and efficient implementation. And this implementation has two components, both of which are part of my proposed work. The first proposal is an implementation of resource-aware session types. This problem is interesting, both from a theoretical and a practical standpoint. The two main challenges arise from *type dependencies* and *type equality*, which are the main topics of this chapter.

The resource bounds of a process often depends on the type refinements of a particular channel. For instance, consider the following type of queue.

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

The potential required for insertion is a function of n, the size of the queue, which is expressed as a type refinement for queue. However, introducing the refinement poses additional challenges for deciding validity of a type and checking type equality. For instance, the some branch in the queue_A type can only be reached if $n > 0$, and therefore $n - 1$ is well defined and non-negative. And this validity property, both for type and process definitions, needs to be incorporated in the implementation.

Another significant challenge is defining type equality in the presence of type dependencies. A sound definition of type equality is necessary for the implementation. Consider the forwarding

rule for a session-typed process.

$$\frac{}{y : A \vdash x \leftarrow y :: (x : A)}\ \mathsf{fwd}$$

To type a forward, the types of y and x must be equal. A similar need arises when type checking a spawn of a process or a cut. There, the types of the channel must match exactly with the type of the spawned process in the signature. Thus, type equality is central to the implementation.

A sound definition must take into account the dependencies. For instance, consider the types $\mathsf{queue}_A[m]$ and $\mathsf{queue}_A[n]$. Without refinements, the two types must be equal. But to compare the type refinements, we must compare m and n. The types should be equal iff $m = n$. Hence, type equality must have a mechanism of comparing natural numbers that support the usual arithmetic, such as addition, subtraction and multiplication. The proposal involves formalizing the details of defining type constraints, and incorporating them while defining type equality.

6.1 Design Decisions

In addition to the above theoretical challenges, I briefly describe the other design decisions I plan to follow in the implementation.

- The language will follow *bi-directional type checking* [85]. The programmer is required to declare the type of each process that is defined, but does not need to provide additional type annotations in the body of the process definition. As an example, consider the simple *copy* process that copies an incoming bit stream to an outgoing bit stream.

```
type copy = +{b0 : bits, b1 : bits, $ : 1}

proc copy : (x : bits) |- (y : bits) =
  case x
    ( b0 => y.b0 ; y <- copy <- x
    | b1 => y.b1 ; y <- copy <- x
    | $  => y.$ ; wait x ; close y )
```

  Thus, the programmer only specifies the type of the process, and the type checker infers the types of the sub-expressions in the process definition. The hope is that the programmer need not insert any internal type annotations.

- The type system is *structural* by design. When comparing types for equality, the type checker uses the structure of the types. This is in contrast with a *nominal* type system, where the type checker uses type names when comparing types for equality. Thus, the two types bits1 and bits2 defined as

```
type bits1 = +{b0 : bits1, b1 : bits1, $ : 1}

type bits2 = +{b0 : bits2, b1 : bits2, $ : 1}
```

have the same structure but different names. Thus, bits1 is equal to bits2 in a structural type system, and unequal in a nominal type system.

A structural type system is more natural for a concurrent message passing system. Two types that have the same communication behavior should be considered equal. A type name is irrelevant when considering its communication behavior.

- The language follows an *equi-recursive* treatment of the types. A type name is considered equivalent to its type definition, i.e., there are no explicit folds or unfolds. Thus, the types bits and $\oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: 1\}$ are considered equal. Again, this is more natural since the two types have the same communication behavior.

The above design decisions introduce their own challenges. First, I would like to minimize the amount of type annotations the programmer needs to introduce. Full type inference for this system is undecidable in the presence of dependent types. Even without refinements, inferring the types is very complicated, and unnecessary in my opinion. Similarly, since types are compared structurally for equality, type checking becomes more challenging since type structures can be possibly infinite. For instance, the type bits can represent possibly infinite bits. Moreover, equi-recursive types increase the complexity of type checking, since we cannot rely on explicit folds or unfolds to guide type checking.

6.2 Type Constraints

To enable type definitions to be valid, the types must incorporate constraints. Currently, since types can only depend on natural numbers, I will only allow arithmetic propositions as type constraints. For instance, consider the queue_A type again.

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

Since we know that the none branch is taken when $n = 0$ and some branch is taken when $n > 0$, I incorporate these using an *existential* type constructor.

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

The type constructor in full generality is written as $!\{\phi\}.\ A$ and denotes that the constraint ϕ must hold in the continuation type A. Theoretically, this requires the provider to send a proof of ϕ, while the client receives this proof, and continues execution *assuming* ϕ holds. The dual type constructor is the *universal* type $?\{\phi\}.\ A$ where the provider receives a proof of ϕ that is sent by the client. In this case, the client continues assuming ϕ holds. In the above queue_A type, since the some branch is taken only when $n > 0$ holds, it can be concluded that $n-1 \geq 0$ and well defined.

With the addition of these two new constructors, I can define a judgment for checking validity of type definitions. The judgment is written as $\Theta \vdash A\ \mathsf{valid}$ defining that the type A is valid under the type constraints Θ. The rules for this judgment are given below. The only non-trivial rule that involves dependencies are ?, ! and name. The former two rules simply add ϕ to the set of constraints, while the latter checks a type name (such as recursive reference to $\mathsf{queue}_A[n-1]$) is valid by ensuring all indices are valid natural numbers.

$$\frac{\Theta \vdash A_\ell\ \mathsf{valid} \qquad (\forall \ell \in L)}{\Theta \vdash \oplus\{\ell : A_\ell\}_{\ell \in L}\ \mathsf{valid}}\ \oplus \qquad \frac{\Theta \vdash A_\ell\ \mathsf{valid} \qquad (\forall \ell \in L)}{\Theta \vdash \&\{\ell : A_\ell\}_{\ell \in L}\ \mathsf{valid}}\ \&$$

$$\frac{\Theta \vdash A\ \mathsf{valid} \qquad \Theta \vdash B\ \mathsf{valid}}{\Theta \vdash A \otimes B\ \mathsf{valid}}\ \otimes \qquad \frac{\Theta \vdash A\ \mathsf{valid} \qquad \Theta \vdash B\ \mathsf{valid}}{\Theta \vdash A \multimap B\ \mathsf{valid}}\ \multimap$$

$$\frac{\Theta, \phi \vdash A\ \mathsf{valid}}{\Theta \vdash ?\{\phi\}.\ A\ \mathsf{valid}}\ ? \qquad \frac{\Theta, \phi \vdash A\ \mathsf{valid}}{\Theta \vdash !\{\phi\}.\ A\ \mathsf{valid}}\ !$$

$$\frac{}{\Theta \vdash \mathbf{1}\ \mathsf{valid}}\ \mathbf{1} \qquad \frac{\Theta \vDash e \geq 0}{\Theta \vdash V\overline{\{e\}}\ \mathsf{valid}}\ \mathsf{name}$$

$$\frac{\Theta \vDash r \geq 0 \qquad \Theta \vdash A\ \mathsf{valid}}{\Theta \vdash \triangleleft^r A\ \mathsf{valid}}\ \triangleleft \qquad \frac{\Theta \vDash r \geq 0 \qquad \Theta \vdash A\ \mathsf{valid}}{\Theta \vdash \triangleright^r A\ \mathsf{valid}}\ \triangleright$$

$$\frac{\Theta \vDash r \geq 0 \qquad \Theta \vdash A\ \mathsf{valid}}{\Theta \vdash \bigcirc^r(A)\ \mathsf{valid}}\ \bigcirc \qquad \frac{\Theta \vdash A\ \mathsf{valid}}{\Theta \vdash \Diamond A\ \mathsf{valid}}\ \Diamond \qquad \frac{\Theta \vdash A\ \mathsf{valid}}{\Theta \vdash \Box A\ \mathsf{valid}}\ \Box$$

Finally, a type definition is considered valid if the definition is a valid type under the constraint that all indices are natural numbers.

$$\frac{\overline{v \geq 0} \vdash A\ \mathsf{valid}}{\vDash V\overline{\{v\}} = A\ \mathsf{valid}}\ \mathsf{def}$$

The type constructors also introduce program syntax. When a process sends a proof of ϕ along channel x, it uses the expression $\mathsf{assert}\ x\ \{\phi\}$. On the other hand, the process receiving this proof assumes it using $\mathsf{assume}\ x\ \{\phi\}$. If the corresponding ϕ is unsatisfiable (dead branch), the process uses the expression $\mathsf{impossible}\ x\ \{\phi\}$. Figure 6.1 presents the relevant typing rules of the system with dependent types. This chapter mainly focuses on work type operators, since they are most relevant to Nomos. The extension to time operators is relatively straightforward.

The typing judgment has the following form: $\Sigma \ ; \ \mathfrak{v} \ ; \ \Theta \ ; \ \Delta \vdash^{q} P :: (x : A)$. Σ denotes the signature, $\mathfrak{v}$ denotes the free variables that exist in the types and processes, Θ denotes the constraints satisfied by the free variables. The judgment types the process P in context Δ offering along channel x a service of type A and storing potential q. The id rule states that the potential q must be 0 under the current constraints and types A and B should be equal according to the type equality definition. The spawn rule calls a process f with indices $\overline{\{e_c\}}$ which are substituted for the indices $\overline{\{v\}}$ from the signature for f. A similar substitution is performed on the potential p and type A of the provided channel in the signature. The $\triangleright$ and $\triangleleft$ rules require that the potential exchanged according to the type and process expression should be equal under the current constraints. The $?R$ rule verifies that the proposition ϕ in the syntax and ϕ' in the type hold and continues with P. The dual $?L$ rule assumes ϕ and adds it to the set of current constraints Θ when typechecking the continuation Q. Dead branches can be typechecked using $?L_{\mathsf{unsat}}$ where the constraints entail that ϕ is unsatisfiable. There is no continuation here since the branch is dead. The rules $!R, !L$ are inverse and switch the roles of the provider and client.

The rules of the cost semantics are similar to the previous constructs. An assertion in a process turns to an assertion message which is consumed by a process waiting on an assume. This is expressed using the following rules.

$$
\begin{array}{lll}
(?S) & \mathsf{proc}(c, w, \mathsf{assert}\ c\ \{\phi\}\ ;\ P) \mapsto & (\phi\ \mathsf{valid}) \\
 & \mathsf{proc}(c', w, [c'/c]P), \mathsf{msg}(c, 0, \mathsf{assert}\ c\ \{\phi\}\ ;\ c \leftarrow c') & (c'\ \mathsf{fresh}) \\
(?C) & \mathsf{msg}(c, w, \mathsf{assert}\ c\ \{\phi\}\ ;\ c \leftarrow c'), \mathsf{proc}(d, w', \mathsf{assume}\ c\ \{\phi\}\ ;\ Q) \mapsto & \\
 & \mathsf{proc}(d, w + w', [c'/c]Q) & \\
(?D) & \mathsf{proc}(d, w, \mathsf{impossible}\ c\ \{\phi\}) \mapsto & (\phi\ \mathsf{invalid}) \\
 & \mathsf{abort} &
\end{array}
$$

The rules for semantics for $!S, !C, !D$ are inverse and skipped.

As an illustration, consider the implementation of a queue as a sequence of *elem* processes terminated by the *empty* process. They will be programmed as presented in Figure 6.2. Apart from the usual implementation challenges such as implementing a parser, interpreter, type checker and runtime, a key challenge here is implementing polymorphism, represented as type A in Figure 6.2. Polymorphism is central to functional programming, and session types in particular. This will also impact the definition of type equality.

Another issue in the implementation is the explicit syntax of the various type constructors I have added for work and type constraints. Writing all the asserts, assumes, pay and get are tedious for the programmer. Since they can be inferred from the type of the process, I would like to also explore program reconstruction. Here, the programmer will simply omit the explicit syntax constructs and write the original session-typed program and the type checker will infer the locations to insert the explicit constructs. The *elem*$[n]$ process will then be implemented as

```
case s
( ins => y <- recv s ;
```

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash q = 0 \qquad \Sigma\ ;\ \mathfrak{v}\ ;\ \Theta \vDash A \equiv B}{\Sigma\ ;\ \mathfrak{v}\ ;\ \Theta\ ;\ y : A \vdash^{q} x \leftarrow y :: (x : B)}\ \mathsf{id}$$

$$\frac{\begin{array}{c}\mathfrak{v}\ ;\ \Theta \vDash r \geq [\overline{e_c}/\overline{v}]p \\ f\overline{\{v\}} : \overline{y_i : A_i} \vdash^{p} x' : A \in \Sigma \qquad \mathfrak{v}\ ;\ \Theta \vDash \Delta_1 \equiv_\alpha \overline{y_i : [e_c/v]A_i} \\ q = [\overline{e_c}/\overline{v}]p \qquad \Sigma\ ;\ \mathfrak{v}\ ;\ \Theta\ ;\ \Delta_2, (x : [\overline{e_c}/\overline{v}]A) \vdash^{r-q} Q_x :: (z : C)\end{array}}{\Sigma\ ;\ \mathfrak{v}\ ;\ \Theta\ ;\ \Delta_1, \Delta_2 \vdash^{r} (x \leftarrow f\overline{\{e_c\}} \leftarrow y\ ;\ Q_x) :: (z : C)}\ \mathsf{spawn}$$

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash q \geq c \qquad \mathfrak{v}\ ;\ \Theta \vDash c \geq 0 \qquad \mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q-c} P :: (x : A)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q} \mathsf{work}\ \{c\}\ ;\ P :: (x : A)}\ \mathsf{work}$$

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash q \geq r_1 \qquad \mathfrak{v}\ ;\ \Theta \vDash r_1 = r_2 \geq 0 \qquad \mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q-r_1} P :: (x : A)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q} \mathsf{pay}\ x\ \{r_1\}\ ;\ P :: (x : \triangleright^{r_2} A)}\ \triangleright R$$

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash r_1 = r_2 \geq 0 \qquad \mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : A) \vdash^{q+r_1} Q :: (z : C)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : \triangleright^{r_2} A) \vdash^{q} \mathsf{get}\ x\ \{r_1\}\ ;\ Q :: (z : C)}\ \triangleright L$$

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash r_1 = r_2 \geq 0 \qquad \mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q+r_1} P :: (x : A)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q} \mathsf{get}\ x\ \{r_1\}\ ;\ P :: (x : \triangleleft^{r_2} A)}\ \triangleleft R$$

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash q \geq r_1 \qquad \mathfrak{v}\ ;\ \Theta \vDash r_1 = r_2 \geq 0 \qquad \mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : A) \vdash^{q-r_1} Q :: (z : C)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : \triangleleft^{r_2} A) \vdash^{q} \mathsf{pay}\ x\ \{r_1\}\ ;\ Q :: (z : C)}\ \triangleleft L$$

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash \phi \qquad \mathfrak{v}\ ;\ \Theta \vDash \phi' \qquad \mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q} P :: (x : A)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q} \mathsf{assert}\ x\ \{\phi\}\ ;\ P :: (x : ?\{\phi'\}.\ A)}\ ?R$$

$$\frac{\mathfrak{v}\ ;\ \Theta, \phi' \vDash \phi \qquad \mathfrak{v}\ ;\ \Theta, \phi\ ;\ \Delta, (x : A) \vdash^{q} Q :: (z : C)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : ?\{\phi'\}.\ A) \vdash^{q} \mathsf{assume}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ ?L$$

$$\frac{\mathfrak{v}\ ;\ \Theta, \phi' \vDash \phi \qquad \mathfrak{v}\ ;\ \Theta, \phi \vDash \bot}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : ?\{\phi'\}.\ A) \vdash^{q} \mathsf{impossible}\ x\ \{\phi\} :: (z : C)}\ ?L_{\mathsf{unsat}}$$

$$\frac{\mathfrak{v}\ ;\ \Theta, \phi' \vDash \phi \qquad \mathfrak{v}\ ;\ \Theta, \phi\ ;\ \Delta \vdash^{q} P :: (x : A)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q} \mathsf{assume}\ x\ \{\phi\}\ ;\ P :: (x : !\{\phi'\}.\ A)}\ !R$$

$$\frac{\mathfrak{v}\ ;\ \Theta \vDash \phi \qquad \mathfrak{v}\ ;\ \Theta \vDash \phi' \qquad \mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : A) \vdash^{q} Q :: (z : C)}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta, (x : !\{\phi'\}.\ A) \vdash^{q} \mathsf{assert}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ !L$$

$$\frac{\mathfrak{v}\ ;\ \Theta, \phi' \vDash \phi \qquad \mathfrak{v}\ ;\ \Theta, \phi \vDash \bot}{\mathfrak{v}\ ;\ \Theta\ ;\ \Delta \vdash^{q} \mathsf{impossible}\ x\ \{\phi\} :: (x : !\{\phi'\}.\ A)}\ !R_{\mathsf{unsat}}$$

FIGURE 6.1: Selected typing rules with dependent indexed types

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

proc elem[n] : (x : A), (t : queue_A[n]) |- (s : queue_A[n+1]) =
  case s
  ( ins => y <- recv s ;
           get s {2*(n+1)} ;
           work {1} ;
           t.ins ;
           work {1} ;
           send t y ;
           s <- elem[n+1] <- x t
  | del => get s {2} ;
           work {1} ;
           s.some ;
           assert s {n+1 > 0} ;
           work {1} ;
           send s x ;
           s <- t )

proc empty : . |- (s : queue_A[0]) =
  case s
  ( ins => x <- recv s ;
           get s {2*(0)} ;
           s <- elem[0] <- x t
  | del => get s {2} ;
           work {1} ;
           s.none ;
           assert s {0 = 0} ;
           work {1} ;
           close s )
```

FIGURE 6.2: Queue program implementation

```
           t.ins ;
           send t y ;
           s <- elem[n+1] <- x t
  | del => s.some ;
           send s x ;
           s <- t )
```

which corresponds exactly to the usual process implementation and is far more compact. However, inferring the exact locations of asserts and assumes, as well as the work constructs can be challenging, especially in the presence of dependent types. These will again be impacted by the definition of type equality.

6.3 Type Equality

Type equality is a standard and well researched problem, even for session types [102]. Intuitively, two types should be considered equal if they have the same communication behavior. For instance, the $\mathsf{queue}_A[n]$ type defined as

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

can accept infinitely many ins messages, while only n del messages. Thus, any type equal to this type must accept the same number of ins and del messages. Thus, $\mathsf{queue}_A[m]$ and $\mathsf{queue}_A[n]$ are equal iff $m = n$.

On the other hand, consider the $\mathsf{ctr}[n]$ type defined as

$$\mathsf{ctr}[n] = \&\{\mathsf{inc} : \mathsf{ctr}[n+1]\}$$

A $\mathsf{ctr}[n]$ type can accept infinitely many inc messages for any $n \in \mathbb{N}$. Thus, $\mathsf{ctr}[m] = \mathsf{ctr}[n]$ for all $m, n \in \mathbb{N}$. Therefore, equality of types not only depends on the index value, but also the structure of the type.

Similarly, consider

$$\mathsf{ctr}'[n] = \&\{\mathsf{inc} : \mathsf{ctr}'[n+2]\}$$

Not only is $\mathsf{ctr}'[m] = \mathsf{ctr}'[n]$, but also $\mathsf{ctr}[m] = \mathsf{ctr}'[n]$ for any $m, n \in \mathbb{N}$, since the two types have the exact same communication behavior. Since the communication can be possibly infinite and arbitrary paths can be pruned away by refinements, equality checking is challenging.

The introduction of dependent types, along with a structural type system and equi-recursive treatment of types makes type equality undecidable. I will first present a proof of undecidability by reducing the problem to the non-halting problem for 2 counter machines. I will then conclude with some ideas on heuristics that can be used to approximately decide equality, and what can be done if the type checker cannot infer type equality.

6.3.1 Type Equality is Undecidable

A two counter machine $\mathcal{M}$ is defined as a sequence of instructions $\iota_1, \iota_2, \ldots, \iota_m$ where each instruction is one of the following.

- $\mathsf{inc}(c_j)$; goto k (increment counter j by 1 and go to instruction k)

- $\mathsf{zero}(c_j)?$ goto k : $\mathsf{dec}(j)$; goto l (if the value of the counter j is 0, go to instruction k, else decrement the counter by 1 and go to instruction l)
- halt (stop computation)

A configuration C of the machine $\mathcal{M}$ is defined as a triple (i, c_1, c_2), where i denotes the number of the current instruction and c_j's denote the value of the counters. A configuration C' is defined as the successor configuration of C, written as $C \mapsto C'$ if C' is the result of executing the i-th instruction on C. If $\iota_i = \mathsf{halt}$, then $C = (i, c_1, c_2)$ has no successor configuration.

The computation of $\mathcal{M}$ is the unique maximal sequence $\rho = \rho(0)\rho(1)\ldots$ such that $\rho(i) \mapsto \rho(i+1)$ and $\rho(0) = (1, 0, 0)$. Either ρ is infinite, or ends in (i, c_1, c_2) such that $\iota_i = \mathsf{halt}$.

The *halting problem* refers to determining whether the computation of a two counter machine $\mathcal{M}$ is finite. Both the halting problem and its dual, the non-halting problem are undecidable. I prove the undecidability of the type equality problem by reducing the non-halting problem of an instance of the two counter machine $\mathcal{M}$ to an instance of the type equality problem.

Consider the i-th instruction ι_i. It can have one of the three forms shown above. I will define a recursive type based on the type of instruction. First, consider an infinite type defined as $T_{\text{inf}} = \oplus\{\ell_1 : T_{\text{inf}}\}$, which sends infinitely many labels ℓ_1.

- Case ($\iota_i = \mathsf{inc}(c_1)$; goto k) : In this case, I define the type $A_i[c_1, c_2] = \oplus\{\ell_1 : A_k[c_1 + 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$. I also define $B_i[c_1, c_2] = \oplus\{\ell_1 : B_k[c_1 + 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$.
- Case ($\iota_i = \mathsf{inc}(c_2)$; goto k) : In this case, I define the type $A_i[c_1, c_2] = \oplus\{\ell_1 : A_k[c_1, c_2 + 1]\}$ for all $c_1, c_2 \in \mathbb{N}$. I also define $B_i[c_1, c_2] = \oplus\{\ell_1 : B_k[c_1, c_2 + 1]\}$ for all $c_1, c_2 \in \mathbb{N}$.
- Case ($\iota_i = \mathsf{zero}(c_1)?$ goto k : $\mathsf{dec}(j)$; goto l) : In this case, I define the type $A_i[c_1, c_2] = \oplus\{\ell_1 : ?\{c_1 = 0\}.\ A_k[c_1, c_2], \ell_2 : ?\{c_1 > 0\}.\ A_l[c_1 - 1, c_2]\}$ for all $c_1, c_2 \in \mathbb{N}$. I also define $B_i[c_1, c_2] = \oplus\{\ell_1 : ?\{c_1 = 0\}.\ B_k[c_1, c_2], \ell_2 : ?\{c_1 > 0\}.\ B_l[c_1 - 1, c_2]\}$.
- Case ($\iota_i = \mathsf{zero}(c_2)?$ goto k : $\mathsf{dec}(j)$; goto l) : In this case, I define the type $A_i[c_1, c_2] = \oplus\{\ell : ?\{c_2 = 0\}.\ A_k[c_1, c_2], \ell : ?\{c_2 > 0\}.\ A_l[c_1, c_2 - 1]\}$ for all $c_1, c_2 \in \mathbb{N}$. I also define $B_i[c_1, c_2] = \oplus\{\ell_1 : ?\{c_2 = 0\}.\ B_k[c_1, c_2], \ell : ?\{c_2 > 0\}.\ B_l[c_1, c_2 - 1]\}$.
- Case ($\iota_i = \mathsf{halt}$) : In this case, I define the type $A_i[c_1, c_2] = T_{\text{inf}}$ for all $c_1, c_2 \in \mathbb{N}$. I also define $B_i[c_1, c_2] = ?\{0 = 1\}.\ T_{\text{inf}}$ for all $c_1, c_2 \in \mathbb{N}$.

Suppose, the counter $\mathcal{M}$ is initialized in the state $(1, m, n)$. The type equality instance is $A_1[m, n] = B_1[m, n]$. The two types only differ at the halting instruction. Therefore, the types are equal iff $\mathcal{M}$ never halts, i.e., the halting instruction is never executed. Moreover, if $\mathcal{M}$ halts, then type $A_1[m, n]$ sends infinitely many labels ℓ_1, while $B_1[m, n]$ only sends finitely many labels, hence they have decidedly different communication behavior. Thus, an instance of the non-halting problem can be reduced to an instance of the type equality problem, thereby proving its undecidability.

6.3.2 Heuristics for Approximate Equality

This section presents some early ideas for heuristics that can be used to infer equality in the presence of dependencies. Since the problem is undecidable in general, these ideas can only be sound, and not complete.

- *Deleting refinements*: For two types to be structurally equal, I can restrict them to have the same structure in the absence of any refinements. Thus, I can delete all dependencies and check the bare bones structure of the type for equality. This problem is actually decidable, since the types are only finitely branching and the number of types are finite. If the types are structurally equal, I will only compare their refinements then.
- *Reflexivity*: I can encode reflexivity in the type equality algorithm. Whenever a type $A[m]$ is compared to $A[n]$, I simply check if $m = n$. If they are, I conclude that the types are equal, otherwise not. This approach fails in the ctr example earlier, but is still sound and should be evaluated for practicality.
- *Transformations*: I am also considering performing type and program transformations which might enable inferring type equality. I also require the type definitions to be contractive, i.e., each definition must start with a constructor. I can perform certain type transformations such that type definitions have a particular shape, making type equality tractable.
- *User Annotations*: I can also request the user to add type equality constraints that can be verified by the type checker. These can then be stored in a global signature, and used to boost the type equality algorithm. For instance, for the ctr example, the programmer can state that $\mathsf{ctr}[m] = \mathsf{ctr}'[n]$, which is verified and later used while inferring type equality.

In conclusion, type equality is a central problem for dependent session types. And to decide this type equality, type constraints are essential to validity of types. In addition to these questions, extending the language to Nomos has its own challenges. First, a functional language needs to be integrated using a linear contextual monad. Moreover, shared channels need to be introduced. They pose a challenge, both while introducing intuitive syntax and while implementing its typing. In addition to these, another future direction is to infer the potential annotations automatically, so that the programmer need not insert them.

6.4 Further Challenges

Apart from the aforementioned technical challenges, an implementation poses other questions that I plan to answer.

- Designing an intuitive surface syntax for work and time type constructors

- Providing informed error messages to guide the programmer with debugging. Error messages are central to any practical language implementation, particularly in the setting of quantitative type constructors. The main sources of error for the programmer are providing insufficient potential in the type for the execution of a process, or incorrect exchange of potential when processes communicate. These must be handled appropriately with intuitive error messages.

- Another important direction is work reconstruction. With the introduction of work operators, programmers have to write work constructs, such as pay and get explicitly, despite them being present in the type. This is tedious to write, as seen in the several examples presented in this chapter. I would like to explore if these can be omitted in the source program, and be inferred just from the type of the process. This problem is non-trivial because there are many program points in the source where these constructs can be inserted. Inserting them too early might prevent the process from spawning another process, and inserting them too late might lead to insufficient potential in the process.

- A central limitation of the current type system for time analysis is the non-determinism in subtyping. This makes type checking considerably inefficient since it involves backtracking while checking, for instance, that a forward is type correct. Backtracking also complicates providing reasonable error messages since a failure in a branch does not necessarily warrant a warning or error. Moreover, time reconstruction also becomes non-deterministic due to the interaction of the time operators. Thus, the reconstruction engine needs to explore all possibilities leading to an exponential blowup.

6.5 Implementation of Nomos

Extending the implementation to Nomos comes with its own obstacles. Here, I am more concerned with the practical aspects of the language. They are summarized below.

- Integration with RAML: At the core of Nomos lies the integration with a resource-aware functional programming language. RAML [59] is a tool that statically and automatically computes bounds on resource use of OCaml programs. RAML relies on off-the-shelf LP solving to determine the potential of the arguments, and works very well in practice. My objective is to reuse the RAML system to infer bounds for the functional layer, and use a similar LP solver architecture to infer bounds on the process layer.

- Shared channels: Nomos requires sharing of channels, as explored in prior work [19]. I would like to define an intuitive surface syntax to denote shared channels. Moreover, the type system needs to distinguish between shared and linear channels. This also creates a challenge in the runtime implementation. A mapping needs to be maintained between a shared channel and its linear counterpart. This is required while releasing a linear channel to its shared counterpart.

- Standard contracts: Once I have a working implementation for Nomos, I would like to evaluate its usability. I would like to implement the standard smart contracts in the literature. These include an online auction, a voting protocol, a bank account, an online lottery, etc. My goal is that contract implementation should be as simple and intuitive as possible. In this regard, I would like to explore the possibility of surface syntax to separate contracts from clients, and other libraries for the interaction with the functional layer.

6.6 Beyond Proposed Work

Beyond the implementation, I would like to explore the possibility of a compiler from Nomos to bytecode. In this regard, there has been some recent work on compiling OCaml to EVM (Ethereum bytecode). I am considering compiling down Nomos to OCaml and using this tool to translate to EVM. Ideally, I would like to deploy contracts implemented in Nomos on a blockchain, and evaluate that the safety guarantees hold in practice.

Another important goal here is inference of gas bounds. Since the gas bounds for standard contracts are often a constant, I would use the techniques of RAML to insert work constructs with a parametric value, and then infer them using an LP solver.

Chapter 7

Runtime Monitoring
Further Proposed Work

A central limitation of Nomos and other existing smart contract languages is that they provide no mechanisms for error handling. Malicious users often try to interact in an undesirable manner, in the hopes of exploiting a vulnerability that is not considered by the programmer. Such a vulnerability can then be used to drain assets from the contracts, if possible. This is the reason behind many hacks across different blockchains, leading to loss of several millions of dollars and trust in the blockchain community.

An important property of Nomos is that the types prescribe the communication protocol. In particular, it expresses the type of every exchange between a contract and client. A runtime monitor [49, 68] can then be inserted as an intermediary in the communication. This monitor can observe the type of the exchange, and confirm that it matches the type prescribed by the session type. This makes detecting an error easier.

However, handling an error is more complicated. The goal is that if an error happens in the middle of a session, then the contract safely and reliably returns back to a well-formed state so it can potentially interact with other clients. Again, this can be guided by the type. Consider the auction contract.

$$
\begin{aligned}
\mathsf{auction} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{11} \oplus \{\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction},\\
&\qquad\qquad \mathsf{cancel} : \triangleright^{8} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction}\},\\
&\quad \mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset\\
&\qquad\qquad \oplus\{\mathsf{won} : \mathsf{lot} \otimes \triangleright^{3} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction},\\
&\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction}\},\\
&\qquad \mathsf{cancel} : \triangleright^{8} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction}\}\}
\end{aligned}
$$

The type clearly demarcates the start and end of a session. The start is indicated by $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ modality, while the end is marked by the $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ modality. Therefore, an error should only occur in the middle

of these modalities. And in case an error is detected, the contract needs to revert to the start of the session. Although the program point to which the state needs to reach is clear from the contract definition, the key question is what to do with the partial communication. I can mark the state of each data structure before the session started and revert to those.

But how to handle the linear resources that were exchanged? In a linear type system, these cannot simply be discarded. Hence, the monitors need to devise a way to safely consume these linear resources. One possibility is to send them back to the client. However, a malicious client may not accept them, violating progress. Another mechanism would be to require the programmer of the contract to create a sink of linear resources. The monitor can then transfer these unconsumed linear resources to the sink, and then safely terminate. This allows the contract to return to a reliable state at the start of the session, and it can resume communication with other clients.

Since I expect contracts to be implemented in Nomos, I would require that they are well typed. Thus, a malicious client would be the most likely and serious attack in this situation. However, it may not be the only one. I plan to explore other sources of attacks on smart contracts, and design programming language techniques to counter them.

Chapter 8

Timeline

PhD Timeline for 2019-21

Feb – May 2019	Implement the simple linear session type system (includes lexer, parser, type checker and runtime)
May – Aug 2019	Internship at Facebook
Sep – Dec 2019	Add work and time operators to the type system Implement reconstruction and explore inference
Jan – May 2020	Add shared channels and functional context Evaluate usability, develop intuitive surface syntax
Jun – Nov 2020	Integrate error handling into Nomos
Nov – Dec 2020	Apply for faculty positions
Jan – Apr 2021	Write thesis

Bibliography

[1] Solidity by example. https://solidity.readthedocs.io/en/v0.3.2/solidity-by-example.html, march 2016. Accessed: 2018-11-04.

[2] Bamboo. https://github.com/cornellblockchain/bamboo, August 2018. Accessed: 2018-11-04.

[3] Welcome to liquidity's documentation! http://www.liquidity-lang.org/doc/index.html, August 2018. Accessed: 2018-11-04.

[4] The michelson language. https://www.michelson-lang.com/, August 2018. Accessed: 2018-11-04.

[5] Rholang. https://github.com/rchain/Rholang, August 2018. Accessed: 2018-11-04.

[6] The rust programming language. https://doc.rust-lang.org/book, 2018.

[7] Vyper. https://vyper.readthedocs.io/en/latest/index.html, August 2018. Accessed: 2018-11-04.

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

[9] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, pages 161–203, 2011.

[10] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. Automatic Inference of Resource Consumption Bounds. In *18th Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'12)*, 2012.

[11] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142 – 159, 2012.

[12] Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. Resource analysis: From sequential to concurrent and distributed programs. In *FM'15*, 2015.

[13] Elvira Albert, Jesús Correas, Einar Broch Johnsen, and Guillermo Román-Díez. Parallel cost analysis of distributed systems. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis*, pages 275–292, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48288-9.

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

[15] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*, 2010.

[16] Robert Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust - 6th International Conference, POST 2017*, pages 164–186, 2017. doi: 10.1007/978-3-662-54455-6_8. URL https://doi.org/10.1007/978-3-662-54455-6_8.

[18] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 152–164, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784753. URL http://doi.acm.org/10.1145/2784731.2784753.

[19] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP):37:1–37:29, 2017.

[20] Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. A universal session type for untyped asynchronous communication. In *29th International Conference on Concurrency Theory (CONCUR)*, 2018. To apper.

[21] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *8th International Workshop on Computer Science Logic (CSL)*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1994. An extended version appeared as Technical Report UCAM-CL-TR-352, University of Cambridge.

[22] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992. ISSN 0167-6423. doi: 10.1016/0167-6423(92)90005-V. URL http://dx.doi.org/10.1016/0167-6423(92)90005-V.

[23] Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97,

pages 249–259, New York, NY, USA, 1997. ACM. ISBN 0-89791-890-8. doi: 10.1145/258492.258517. URL http://doi.acm.org/10.1145/258492.258517.

[24] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, pages 419–434, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44584-6.

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

[26] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*, 2014.

[27] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, volume 310, 2016.

[28] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR)*, pages 222–236. Springer, 2010.

[29] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 330–349, Rome, Italy, March 2013. Springer LNCS 7792.

[30] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, pages 402–417, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-85361-9.

[31] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated Resource Analysis with Coq Proof Objects. In *29th International Conference on Computer-Aided Verification (CAV'17)*, 2017.

[32] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[33] Pavol Cerný, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[34] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044 – 1077, 2009. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2008.11.006. URL http://www.sciencedirect.com/science/article/pii/S089054010900100X. Special

issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

[35] Ezgi Çiçek, Deepak Garg, and Umut Acar. Refinement types for incremental computational complexity. In Jan Vitek, editor, *Programming Languages and Systems*, pages 406–431, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8.

[36] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[37] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784749. URL http://doi.acm.org/10.1145/2784731.2784749.

[38] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *33rd ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*, 2018.

[39] Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. In *23rd International Conference on Functional Programming (ICFP'18)*, 2018.

[40] Ankush Das, Stephanie Balzer, Jan Hoffmann, and Frank Pfenning. Resource-aware session types for digital contracts. Manuscript submitted for review, 2019.

[41] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press. URL http://www.cs.cmu.edu/afs/cs/user/rowan/www/papers/multbta.ps.Z.

[42] Nicolas Feltman, Carlo Angiuli, Umut Acar, and Kayvon Fatahalian. Automatically splitting a two-stage lambda calculus. In P. Thiemann, editor, *Proceedings of the 25th European Symposium on Programming (ESOP)*, pages 255–281, Eindhoven, The Netherlands, April 2016. Springer LNCS 9632.

[43] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: Semantics and cut elimination. In *22nd Conference on Computer Science Logic*, volume 23 of *LIPIcs*, pages 248–262, 2013.

[44] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sara Décova. Exceptional asynchronous session types: Session types without tiers. Principles of Programming Languages (to appear), 2019.

[45] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, Nov 2005. ISSN 1432-0525. doi: 10.1007/s00236-005-0177-z. URL https://doi.org/10.1007/s00236-005-0177-z.

[46] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, pages 331–350, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-642-54832-1. doi: 10.1007/978-3-642-54833-8_18. URL http://dx.doi.org/10.1007/978-3-642-54833-8_18.

[47] Stéphane Gimenez and Georg Moser. The complexity of interaction. In *POPL'16*, 2016.

[48] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[49] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. In A. Ahmed, editor, *European Symposium on Programming (ESOP'18)*, pages 771–798, Thessaloniki, Greece, April 2018. Springer LNCS 10801.

[50] L.M Goodman. Tezos — a self-amending crypto-ledger. https://tezos.com/static/papers/white_paper.pdf, 2014.

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

[52] Dennis Griffith and Elsa L. Gunter. Liquid pi: Inferrable dependent session types. In *NASA Formal Methods Symp.'13*, 2013.

[53] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, 2009.

[54] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991. ISSN 0018-9219. doi: 10.1109/5.97300.

[55] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985. ISSN 0164-0925. doi: 10.1145/4472.4478. URL http://doi.acm.org/10.1145/4472.4478.

[56] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)*, 2010.

[57] Jan Hoffmann and Zhong Shao. Automatic Static Cost Analysis for Parallel Programs. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[58] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th ACM Symp. on Principles of Prog. Langs. (POPL'11)*, 2011.

[59] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 781–786, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31424-7.

[60] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[61] Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, 2003.

[62] Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, 2006.

[63] Martin Hofmann and Georg Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, 2015.

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

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

[66] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008.

[67] Blockchain Insurance Industry Initiative. B3i. 2008.

[68] Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *43rd Annual Symposium on Principles of Programming Languages*, pages 582–594, St. Petersburg, Florida, January 2016. ACM Press.

[69] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, 2010.

[70] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.

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

[72] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric Semantics of Reactive Programs. In *26th IEEE Symposium on Logic in Computer Science, (LICS'11)*, pages 257–266, 2011.

[73] Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

[74] Ugo Dal Lago and Barbara Petit. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, 2013.

[75] Angwei Law. *Smart contracts and their application in supply chain management.* PhD thesis, Massachusetts Institute of Technology, 2017.

[76] Hugo A. López, Carlos Olarte, and Jorge A. Pérez. Towards a unified framework for declarative structure communications. In A. Beresford and S. Gay, editors, *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*, pages 1–15. EPTCS 17, March 2009.

[77] Lucius Gregory Meredith. Linear types can change the blockchain. *arXiv preprint arXiv:1506.01001*, 2015.

[78] Vincenzo Morabito. Smart contracts and licensing. In *Business Innovation Through Blockchain*, pages 101–124. Springer, 2017.

[79] David Z. Morris. Blockchain-based venture capital fund hacked for $ 60 million. http://fortune.com/2016/06/18/blockchain-vc-fund-hacked, June 2016.

[80] Dimitris Mostrous and Vasco Thudichum Vasconcelos. Affine sessions. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages*, pages 115–130, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43376-8.

[81] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[82] Hiroshi Nakano. A Modality for Recursion. In *15th IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 255–266, 2000.

[83] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. In *3rd International Workshop on Behavioural Types (BEAT 2014)*, 2014.

[84] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In Josep Diaz, editor, *Automata, Languages and Programming*, pages 597–609, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg. ISBN 978-3-540-40038-7.

[85] Frank Pfenning. Lecture notes on bidirectional type checking, 2004.

[86] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 3–22. Springer, 2015.

[87] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Computer Society, 1977.

[88] Marc Pouzet. Lucid synchrone release, version 3.0 tutorial and reference manual, 2006.

[89] Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Technical report, Carnegie Mellon University, April 2018.

[90] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic Refinements for Relational Cost Analysis. *Proc. ACM Program. Lang.*, 2(POPL), 2017.

[91] Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, January 2009. URL http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf.

[92] Neda Saeedloei and Gopal Gupta. Timed π-calculus. In *8th International Symposium on Trustworthy Global Computing - Volume 8358*, TGC 2013, pages 119–135, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-319-05118-5. doi: 10.1007/978-3-319-05119-2_8. URL https://doi.org/10.1007/978-3-319-05119-2_8.

[93] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *CoRR*, abs/1801.00687, 2018. URL http://arxiv.org/abs/1801.00687.

[94] Miguel Silva, Mário Florido, and Frank Pfenning. Non-blocking concurrent imperative programming with session types. In *Fourth International Workshop on Linearity*, June 2016.

[95] Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*, 2012.

[96] Moritz Sinn, Florian Zuleger, and Helmut Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, 2014.

[97] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[98] Tachio Terauchi and Adam Megacz. Inferring channel buffer bounds via linear programming. In *ESOP'08*, 2008.

[99] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *22nd European Symposium on Programming (ESOP)*, pages 350–369. Springer, 2013.

[100] Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In M. Maffei and E. Tuosto, editors, *Proceedings of the 9th International Symposium on Trustworthy Global Computing (TGC 2014)*, pages 159–175, Rome, Italy, September 2014. Springer LNCS 8902.

[101] Pedro Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.

[102] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, August 2012. ISSN 0890-5401. doi: 10.1016/j.ic.2012.05.002. URL http://dx.doi.org/10.1016/j.ic.2012.05.002.

[103] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 546–566. North, 1990.

[104] Philip Wadler. Propositions as sessions. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012.

[105] Max Willsey, Rokhini Prabhu, and Frank Pfenning. Design and implementation of Concurrent C0. In *Fourth International Workshop on Linearity*, June 2016.

[106] Gavin Wood. Ethereum: A secure decentralized transaction ledger, 2014.

[107] Florian Zuleger, Moritz Sinn, Sumit Gulwani, and Helmut Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*, 2011.