DRAFT

CARNEGIE MELLON UNIVERSITY

Resource-Aware Session Types for Digital Contracts

Thesis

by

Ankush Das

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

April 2021

Abstract

Programming distributed systems is already very challenging due to the presence of data races and deadlocks; bugs are difficult to detect and reproduce when they only arise in certain thread interleavings. The rise of modern distributed systems have introduced unique domain-specific challenges further complicating software development. Although program analysis tools exist for distributed systems, the most popular and usable tools are still centered around traditional programming languages. With the pervasive usage of distributed systems in software design, there is an urgent need for formal tools to help with the design, verification, and quantitative analysis of distributed software.

In response, this thesis designs novel resource-aware session types that serve as a sound and practical foundation for distributed systems with strong type-theoretic guarantees. Session types statically prescribe bidirectional communication protocols for message-passing processes. Unfortunately, they cannot express quantitative properties of a distributed system, such as energy consumption, latency, response time, and throughput. This thesis addresses this limitation by proposing two extensions to express the *work* and *span* of parallel computation. To compute work, the key innovation was that messages and processes both carry an abstract notion of *potential* which is consumed to perform work. To compute span, the key innovation was to introduce operators from temporal logic to capture the timing of message exchanges. Resource-aware session types combine session types with work and span extensions allowing programmers to reason about both qualitative and quantitative aspects of distributed systems.

The thesis concludes with an application of resource-aware session types to the blockchain domain. Blockchains allow execution of complex protocols between mutually distrusting parties through *smart contracts*. Programming smart contracts comes with unique challenges such as enforcing transaction protocols, computing the execution cost of transactions, and ensuring that assets are not accidentally duplicated or discarded accidentally. This thesis presents *Nomos*: a language for smart contracts with resource-aware session types at its core. Session types statically express contract protocols. Resource-aware types automatically infer the execution cost of transactions leveraging ideas from automatic amortized resource analysis. The built-in linear type system of session types provides a natural representation for assets. The Nomos type checker statically enforces the above requirements: protocols are enforced at runtime, bounds inferred are sound and precise, and assets used are neither duplicated nor discarded. Nomos also significantly develops the theory of programming languages: integrating session types with functional programming, linear-time type checking to prevent denial-of-service attacks, and an acquire-release discipline to rule out re-entrancy attacks.

Contents

Chapter 1

Introduction

The design of safe, efficient, and secure software has always been a challenging task. With the proliferation of distributed systems, software development has become even more complex. In addition to the usual challenges, developers must also carefully

- ensure that no bugs manifest in any possible thread interleaving,
- avoid deadlocks and data races, and
- calculate the overall memory and time consumption of the system.

The rise of modern distributed systems such as server farms, cloud computing platforms, and blockchains have further complicated software design by introducing unique challenges. For instance, blockchain is a highly adversarial domain due to its transparency. The state of every decentralized application deployed on the blockchain can be publicly viewed and potentially exploited by malicious attackers. Security vulnerabilities in these applications have caused losses to the tune of several billions of dollars.

Advances in the design and principles of programming languages have significantly benefitted the design of software, improving its performance, safety and security. Such advances have enabled programmers analyze the *qualitative* aspects of software using verification techniques as well as the *quantitative* aspects using complexity analysis tools. Today, developers can utilize automatic verification tools such as Boogie [28] and Dafny [97], or program assistants such as Coq [29] and Isabelle [115] to express and verify that programs satisfy their specifications. They can exploit tools such as Fiat Crypto [61] and EverCrypt [121] to generate secure high-performance cryptographic functionalities. They can employ fully formally verified compilers such as CompCert [98] to compile these programs to efficient low-level machine code. Finally, they can take advantage of static analysis tools such as RaML [82],[83] and SPEED [76] to compute complexity bounds and Infer [41] to detect performance bugs.

Although analysis tools exist for distributed systems, most of the industrial strength support is still centered around traditional sequential programming languages. Programs implemented in sequential languages are easier to analyze and verify since all computation typically occurs in a single process. However, with the advent of multi-processor systems, most of the industrial software systems today are no longer sequential. Distributed algorithms lie at the heart of commitment protocols, distributed consensus, leader election and broadcast mechanisms with a wide array of applications ranging from communication networks and database management to industrial control systems, cloud computing and blockchains.

With the advent of such distributed algorithms and systems, there is an urgent need of formal tools to assist the design, verification and quantitative analysis of distributed software. However, analysis of concurrent and distributed systems poses additional challenges. First, while implementing concurrent programs, developers must consider all possible thread interleavings and ensure that no bug manifests in any such interleaving. Second, concurrent programs are notoriously prone to deadlocks where each process in a group is waiting on a resource held by another process, and data races where one thread is writing to a shared resource that is simulataneously being read by another thread. Finally, unlike sequential programs, creating a compositional reasoning for concurrent programs is especially challenging since their behavior ultimately depends on interactions within the system.

In response, this thesis has designed novel resource-aware session types that serve as a sound and practical foundation for distributed systems with strong type-theoretic guarantees [54, 55]. A common theme in a distributed system is communication between its components. And often, the type, value, and direction of communication depends on the state of the system. Session types leverage this property by capturing the system state in the type, providing a structured way of prescribing communication protocols. Session types can also guarantee freedom from deadlocks and data races. They also possess a confluence property stating that a communicating system converges to the same final state under all possible thread interleavings. All these properties greatly benefit programmers and automatically prevent a large class of bugs that can result from communication mismatches.

Unfortunately, simple session types cannot express quantitative properties of a distributed system, such as energy consumption, latency, response time, and throughput. Realizing this, this thesis proposes two extensions to express the *work* [54] and *span* [55] of parallel computation. Work is defined as the *total number of operations* executed as part of the computation. However, due to parallelism in the system, many of these operations execute simultaneously. Therefore, span is defined as the total *time* of computation taking the parallelism in the system into account.

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 [135]) 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.

Resource-aware session types combine session types with work and span extensions allowing programmers to reason about both qualitative and quantitative aspects of distributed systems.

1.1 Programming Digital Contracts using Resource-Aware Session Types

Digital contracts are computer protocols that describe and enforce the execution of a contract. With the rise of blockchains and cryptocurrencies such as Bitcoin [109], Ethereum [145], and Tezos [72], 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 [108], insurance agreements [88], supply chain management [96], and mortgage loans [107]. 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 [49], Rholang [7], and Liquidity [5]. 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 [105].

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 [108], 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.

Recently, I have designed the type-theoretic foundations of Nomos [57], 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 [69] 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 [25] 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 $\uparrow_{\mathsf{L}}^{\mathsf{S}}$ modality marking its start and $\downarrow_{\mathsf{L}}^{\mathsf{S}}$ modality marking its end. Programmatically, $\uparrow_{\mathsf{L}}^{\mathsf{S}}$ translates into an acquire of the contract, while $\downarrow_{\mathsf{L}}^{\mathsf{S}}$ into its release.

We complement the theory of Nomos with an efficient implementation. We have designed a lexer, parser, type checker, inference engine, and interpreter for Nomos. The type checker verifies if Nomos programs indeed satisfies their protocol specification, while generating linear constraints on the potential needed to execute the program. The inference engine solves the linear constraints minimizing the potential, thus generating tight bounds. Finally, the interpreter executes the transaction program.

In addition, we describe how Nomos is integrated with an account model blockchain like Ethereum [145]. We describe the challenges in this integration, and highlight the main limitation of the language. We also describe our efforts in simplifying programming in Nomos by introducing surface syntax and commonly used data structures. We conclude by evaluating Nomos on a variety of standard smart contracts, emphasizing the guarantees provided by Nomos and their advantages.

1.2 Overview

Thesis Statement *Resource-aware 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 descibes our novel extension to refinement session types. Such refinements are crucial to describing work and span bounds. We define what type equality means in this refined setting, and also present an algorithm for approximating equality.

Chapter 4 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 5 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 6 designs the Nomos language. 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 7 describes the implementation of Nomos.

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 [39]. This chapter focuses on the linear fragment of SILL [116] that internalizes session-based concurrency. Session types were introduced by Honda [87].

Linear logic [69] 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.

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

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 $c \leftrightarrow d$ *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* [45]. 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.

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

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 $(c.k \ ; \ c \leftrightarrow c')$ (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'.

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

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.

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

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.

$$\begin{array}{lll}(\&S) & \mathsf{proc}(d, c.k\ ;\ Q) \mapsto \mathsf{msg}(c', c.k\ ;\ c' \leftrightarrow c), \mathsf{proc}(d, [c'/c]Q) & (c' \text{ fresh}) \\ (\&C) & \mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\ ;\ c' \leftrightarrow c) \mapsto \mathsf{proc}(c', [c'/c]Q_k) & \end{array}$$

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 $\otimes L$.

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

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

$$\begin{array}{lll}(\otimes S) & \mathsf{proc}(c, \mathsf{send}\ c\ e\ ;\ P) \mapsto \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ e\ ;\ c \leftrightarrow c') & (c' \text{ fresh}) \\ (\otimes C) & \mathsf{msg}(c, \mathsf{send}\ c\ e\ ;\ c \leftrightarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \mapsto \mathsf{proc}(d, [c', e/c, x]Q) & \end{array}$$

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.

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

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

$$\begin{array}{lll}(\multimap S) & \mathsf{proc}(d, \mathsf{send}\ c\ e\ ;\ Q) \mapsto \mathsf{msg}(c', \mathsf{send}\ c\ e\ ;\ c' \leftrightarrow c), \mathsf{proc}(d, [c'/c]Q) & (c' \text{ fresh}) \\ (\multimap C) & \mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c), \mathsf{msg}(c', \mathsf{send}\ c\ e\ ;\ c' \leftrightarrow c) \mapsto \mathsf{proc}(c', [c', d/c, x]P) & \end{array}$$

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.

$$\begin{array}{ll} (\mathbf{1}S) & \mathsf{proc}(c, \mathsf{close}\ c) \mapsto \mathsf{msg}(c, \mathsf{close}\ c) \\ (\mathbf{1}C) & \mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \mapsto \mathsf{proc}(d, Q) \end{array}$$

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):

$$\begin{array}{l} \Delta \vdash f :: (x : A) \\ x \leftarrow f\ \Delta = P \end{array}$$

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

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

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 $x \leftarrow f\ \overline{y}$. This is a short-hand for $x' \leftarrow f\ \overline{y}\ ;\ x \leftrightarrow x'$ 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* [66]. 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

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

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

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

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

```
$y : \mathsf{bits} \vdash \textit{copy} :: (x : \mathsf{bits})$
$x \leftarrow \textit{copy}\ y =$
  $\mathsf{case}\ y\ (\mathsf{b0} \Rightarrow x.\mathsf{b0}\ ;\ x \leftarrow \textit{copy}\ y$      % received b0 on $y$, send b0 on $x$, recurse
           $\mid \mathsf{b1} \Rightarrow x.\mathsf{b1}\ ;\ x \leftarrow \textit{copy}\ y$      % 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 ($x \leftrightarrow y$), or turn b1 to b0 but then increment the remaining stream ($x \leftarrow \textit{plus1}\ y$) to capture the effect of the carry bit.

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

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

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

FIGURE 2.1: Typing rules for a configuration

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.

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

It states that Σ is well-formed and that the configuration $\mathcal{S}$ uses the channels in the context Δ_1 and provides the channels in the context Δ_2. The configuration typing judgment is defined using the rules presented in Figure 3.2. The rule empty defines that an empty configuration is well-typed. The rule compose composes two configurations $\mathcal{S}_1$ and $\mathcal{S}_2$; $\mathcal{S}_1$ provides service on the channels in Δ_1 while $\mathcal{S}_2$ 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* $\Sigma \; ; \; \Delta' \vDash \mathcal{S} :: \Delta$ *and* $\mathcal{S} \mapsto \mathcal{D}$*, 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* $\cdot \vDash \mathcal{S} :: \Delta$ *then either*

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

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

Proof. By induction on the right-to-left typing of $\mathcal{S}$ so that either $\mathcal{S}$ is empty (and therefore poised) or $\mathcal{S} = (\mathcal{D}\ \mathsf{proc}(c, P))$ or $\mathcal{S} = (\mathcal{D}\ \mathsf{msg}(c, M))$. By induction hypothesis, $\mathcal{D}$ can either take a step (and then so can $\mathcal{S}$), 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 $\mathcal{S}$ 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 [77], ranking functions [12, 18, 37, 134], recurrence relations [13, 14] and abstract interpretation [44, 150]. Automatic resource analysis for functional programs are based on sized types [140], term-rewriting [23] and amortized resource analysis [82, 84, 90, 133].

Automatic amortized resource (AARA) was introduced for a strict first-order functional language with built-in data types [84]. Since then, AARA techniques have been applied to univariate polynomial bounds [80], multivariate bounds [82], higher-order functional programs [90] and user-defined data types [83]. 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 [135] 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*.

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

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*.

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

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

Refinement Session Types

Traditional session types prescribe bidirectional communication protocols for concurrent computations, where well-typed programs are guaranteed to adhere to the protocols. However, simple session types cannot capture properties beyond the basic type of the exchanged messages. In response, we index session types with refinements from linear arithmetic, capturing intrinsic attributes of processes and data. These refinements then play a central role in describing sequential and parallel complexity bounds on session-typed programs (Chapters 4 and 5).

This chapter describes the metatheory of such indexed types. We show that, despite the decidability of Presburger arithmetic, type equality and therefore also type checking are now undecidable, which stands in contrast to analogous dependent refinement type systems from functional languages. We also present a practical incomplete algorithm for type equality and an algorithm for type checking which is complete relative to an oracle for type equality. Process expressions in this explicit language are rather verbose, so we also introduce an implicit form and a sound and complete algorithm for reconstructing explicit programs, borrowing ideas from the proof-theoretic technique of focusing. All the aforementioned ideas have been implemented in an open-source language named Rast [56]. We conclude by illustrating our systems and algorithms with a variety of examples that have been verified in the Rast implementation.

3.1 Introduction

Basic session types have limited expressivity. As a simple example, consider the session type offered by a queue data structure storing elements of type A.

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

This type describes a queue interface supporting insertion and deletion. The *external choice* operator $\&$ dictates that the process providing this data structure accepts either one of two messages: the labels **ins** or **del**. In the case of the label **ins**, it then receives an element of type A denoted by the $\multimap$ operator, and then the type recurses back to queue_A. On receiving a **del** request, the process can respond with one of two labels (**none** or **some**), indicated by the *internal choice* operator $\oplus$. It responds with **none** and then *terminates* (indicated by $\mathbf{1}$) if the queue is empty, or with **some** followed by the element of type A (expressed with the $\otimes$ operator) and recurses if the queue is nonempty. However, the simple session type does not express the conditions under which the **none** and **some** branches must be chosen, which requires tracking the length of the queue.

We propose extending session types with simple arithmetic refinements to express, for instance, the size of a queue. The more precise type

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

uses the index refinement n to indicate the size of the queue. In addition, we introduce a *type constraint* $?\{\phi\}.\,A$ which can be read as "*there exists a proof of* ϕ" and is analogous to the *assertion* of ϕ in imperative languages. Here, the process providing the queue must (conceptually) send a proof of $n = 0$ after it sends **none**, and a proof of $n > 0$ after it sends **some**. It is therefore constrained in its choice between the two branches based on the value of the index n. Because the the index domain from which the propositions ϕ are drawn is Presburger arithmetic and hence decidable, no proof of ϕ will actually be sent, but we can nevertheless verify the constraint statically (which is the subject of this chapter) or dynamically (see [70, 71]). Although not used in this example, we also add the dual $!\{\phi\}.\,A$ (*for all proofs of* ϕ, analogous to the *assumption* of ϕ), and explicit quantifiers $\exists n.\,A$ and $\forall n.\,A$ that send and receive natural numbers, respectively.

Of course, arithmetic type refinements are not new and have been explored extensively in functional languages, for example, by Zenger [148], in DML [146], or in the form of *Liquid Types* [127]. Variants have been adapted to session types as well [71, 75, 149], generally with the implicit assumption that index refinements are somehow "orthogonal" to session types. In this chapter we show that, upon closer examination, this is *not* the case. In particular, unlike in the functional setting, session type equality and therefore type checking become undecidable. Remarkably, this is the case whether we treat session types equirecursively [66] or isorecursively [101], and even in the quantifier-free fragment. In response, we develop a new algorithm for type equality which, though incomplete, easily handles the wide variety of example programs we have tried. Moreover, it is naturally extensible through the additional assertion of type invariants should the need arise.

With a practically effective type equality algorithm in hand, we then turn our attention to *type checking*. It turns out that assuming an oracle for type equality, type checking is decidable because it can be reduced to checking the validity of propositions in Presburger arithmetic.

We define *type checking* over a language where constructs related to arithmetic constraints ($\exists n.\, A$, $\forall n.\, A$, $?\{\phi\}.\, A$, and $!\{\phi\}.\, A$) have explicit communication counterparts. Despite the high theoretical complexity of deciding Presburger arithmetic, all our examples check very quickly using Cooper's decision procedure [48] with two optimizations.

Many programs in this explicit language are unnecessarily verbose and therefore tedious for the programmer to write, because the process constructs pertaining to the refinement layer contribute only to verifying its properties, but not its observable computational outcomes. As is common for refinement types, we therefore also designed an *implicit* language for processes where most constructs related to index refinements are omitted. The problem of *reconstruction* is then to map such an implicit program to an explicit one which is sound (the result type-checks) and complete (if there is a reconstruction, it can be found). Interestingly, the nature of Presburger arithmetic makes full reconstruction impossible. For example, the proposition $\forall n.\, \exists k.\, (n = 2k \lor n = 2k + 1)$ is true but the witness for k as a Skolem function of n (namely $\lfloor n/2 \rfloor$) cannot be expressed in Presburger arithmetic. Since witnesses are critical if we want to understand the work performed by a computation, we require that type quantifiers $\forall n.\, A$ and $\exists n.\, A$ have explicit witnesses in processes. We provide a sound and complete algorithm for the resulting reconstruction problem. This algorithm exploits proof-theoretic properties of the sequent calculus akin to focusing [20] to avoid backtracking and consequently provides precise error messages that we have found to be helpful.

We have implemented our language, named Rast, in SML, where a programmer can choose explicit or implicit syntax and the exact cost model for work analysis. The implementation consists of a lexer, parser, type checker, and reconstruction engine, with particular attention to providing precise error messages.

3.2 Arithmetic Refinements

Before we extend our language of types formally, we revisit the examples in order to motivate the specific constructs available. We write $V[\overline{e}]$ for a type indexed by a sequence of arithmetic expressions e. Since it has been appropriate for most of our examples, we restrict ourselves to natural numbers rather than arbitrary integers.

Example 3.1 (Queues, v2)**.** *The provider of a queue should be constrained to answer* **none** *exactly if the queue contains no elements and* **some** *if it is nonempty. The queue type from Section 3.1 does not express this. This means a client may need to have some redundant branches to account for responses that should be impossible. Instead we use the refined type* $\mathsf{queue}_A[n]$ *to stand for a queue with* n *elements.*

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

The first branch is easy to understand: if we add an element to a queue of length n, it subsequently contains $n + 1$ elements. In the second branch we constrain *the arithmetic variable n to be equal to 0 if the provider sends* **none** *and positive if the provider sends* **some**. *In the latter case, we subtract one from the length after an element has been dequeued.*

Conceptually, the type $?\{\phi\}.\,A$ means that the provider must send a proof of ϕ, so it corresponds to $\exists p : \phi.\,A$. A characteristic of *type refinement*, in contrast to fully dependent types, is that the computation of A can only depend on the *existence* of a proof p, but not on its form. Since our index domain is also decidable no actual proof needs to be sent (since one can be constructed from ϕ automatically, if needed), just a token *asserting* its existence. There is also a dual constructor $!\{\phi\}.\,A$ that licenses the *assumption* of ϕ, which, conceptually, corresponds to *receiving* a proof of ϕ.

Example 3.2 (Binary Numbers). *We would like the indexed type* $\mathsf{bin}[n]$ *to represent a binary number with value n. Because the least significant bit comes first, we expect, for example, that* $\mathsf{bin}[n] = \oplus\{\mathbf{b0} : ?\{2 \mid n\}.\,\mathsf{bin}[n/2], \ldots\}$. *However, while divisibility is available in Presburger arithmetic, division itself is not; instead, we can express the constraint and the index of the recursive occurrence using quantification.*

$$\begin{aligned}\mathsf{bin}[n] = \oplus\{&\mathbf{b0} : \exists k.\,?\{n = 2 * k\}.\,\mathsf{bin}[k],\\ &\mathbf{b1} : \exists k.\,?\{n = 2 * k + 1\}.\,\mathsf{bin}[k],\\ &\mathbf{e} : ?\{n = 0\}.\,\mathbf{1}\}\end{aligned}$$

As a further refinement, we could rule out leading zeros by adding the constraint $n > 0$ in the branch for **b0**.

The type $\exists n.\,A$ means that the provider must send a natural number i and proceed at type $A[i/n]$, corresponding to existential quantification in arithmetic. The dual universal quantifier $\forall n.\,A$ requires the provider to receive a number i and proceed at type $A[i/n]$.

We now extend our definitions to account for these new constructs. Below, i represents a constant, while n represents a natural number variable.

Types	A	$::=$	$\ldots$		
		$\mid$	$?\{\phi\}.\,A$	assert ϕ	continue at type A
		$\mid$	$!\{\phi\}.\,A$	assume ϕ	continue at type A
		$\mid$	$\exists n.\,A$	send number i	continue at type $A[i/n]$
		$\mid$	$\forall n.\,A$	receive number i	continue at type $A[i/n]$
		$\mid$	$V[\overline{e}]$	variable instantiation	
Arith. Expressions	e	$::=$	$i \mid e + e \mid e - e \mid i \times e \mid (i \mid e) \mid n$		
Arith. Propositions	ϕ	$::=$	$e = e \mid e > e \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \exists n.\,\phi \mid \forall n.\,\phi$		
Signature	Σ	$::=$	$\cdot \mid \Sigma, V[\overline{n} \mid \phi] = A$		

An indexed type definition $V[\overline{n} \mid \phi] = A$ requires every instance $\overline{e}$ of the sequence of variables $\overline{n}$ to satisfy $\phi[\overline{e}/\overline{n}]$. This is verified statically when a type signature is checked for validity, as defined below. We use $\mathcal{V}$ for a collection of arithmetic variables and $\mathcal{C}$ (to signify *constraints*) for an arithmetic proposition occurring among the antecedents of a judgment. We then have the following rules defining the validity of signatures ($\vdash \Sigma$ *signature*), declarations ($\vdash_\Sigma \Sigma'$ *valid*), and types ($\mathcal{V} \; ; \; \mathcal{C} \vdash_\Sigma A$ *valid*) where $\mathcal{V}$ is a collection of arithmetic variables including all free variables in constraint $\mathcal{C}$ and type A. We silently rename variables so that n does not already occur in $\mathcal{V}$ in the $\exists V$ and $\forall V$ rules. We also call upon the semantic entailment judgment $\mathcal{V} \; ; \; \mathcal{C} \vDash \phi$ which means that $\forall \mathcal{V}. \, \mathcal{C} \supset \phi$ holds in arithmetic and $\vDash \phi$ abbreviates $\cdot \; ; \; \top \vDash \phi$.

$$\frac{\vdash_\Sigma \Sigma \; \textit{valid}}{\vdash \Sigma \; \textit{signature}} \qquad \frac{}{\vdash_\Sigma (\cdot) \; \textit{valid}} \qquad \frac{\vdash_\Sigma \Sigma' \; \textit{valid} \quad \overline{n} \; ; \; \phi \vdash_\Sigma A \; \textit{valid} \quad A \neq V'[\overline{e}']}{\vdash_\Sigma \Sigma', V[\overline{n} \mid \phi] = A \; \textit{valid}}$$

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \wedge \phi \vdash_\Sigma A \; \textit{valid}}{\mathcal{V} \; ; \; \mathcal{C} \vdash_\Sigma ?\{\phi\}. \, A \; \textit{valid}} \; ?V \qquad \frac{\mathcal{V} \; ; \; \mathcal{C} \wedge \phi \vdash_\Sigma A \; \textit{valid}}{\mathcal{V} \; ; \; \mathcal{C} \vdash_\Sigma !\{\phi\}. \, A \; \textit{valid}} \; !V$$

$$\frac{\mathcal{V}, n \; ; \; \mathcal{C} \vdash_\Sigma A \; \textit{valid}}{\mathcal{V} \; ; \; \mathcal{C} \vdash_\Sigma \exists n. \, A \; \textit{valid}} \; \exists V^n \qquad \frac{\mathcal{V}, n \; ; \; \mathcal{C} \vdash_\Sigma A \; \textit{valid}}{\mathcal{V} \; ; \; \mathcal{C} \vdash_\Sigma \forall n. \, A \; \textit{valid}} \; \forall V^n$$

$$\frac{V[\overline{n} \mid \phi] = A \in \Sigma \quad \mathcal{V} \; ; \; \mathcal{C} \vDash \phi[\overline{e}/\overline{n}]}{\mathcal{V} \; ; \; \mathcal{C} \vdash_\Sigma V[\overline{e}] \; \textit{valid}} \; \mathsf{tdef}$$

We elide the compositional rules for all the other type constructors. Since we like to work over natural numbers rather than integers, it is convenient to assume that every definition $V[\overline{n}] = A$ abbreviates $V[\overline{n} \mid \overline{n} \geq 0] = A$. This means that in valid signatures every occurrence $V[\overline{e}]$ is such that $\overline{e} \geq 0$ follows from the known constraints.

Example 3.3. *The declaration*

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

is valid because $n \geq 0 \vDash n + 1 \geq 0$ *and* $n \geq 0 \wedge n > 0 \vDash n - 1 \geq 0$.

Unfolding a definition must substitute for the arithmetic variables we abstract over.

Definition 3.1. $\mathsf{unfold}_\Sigma(V[\overline{e}]) = A[\overline{e}/\overline{n}]$ if $V[\overline{n} \mid \phi] = A \in \Sigma$ and $\mathsf{unfold}_\Sigma(A) = A$ otherwise.

We say that a type is *closed* if it contains no free arithmetic variables n.

Definition 3.2. A relation $\mathcal{R}$ on types is a *type bisimulation* if $(A, B) \in \mathcal{R}$ implies that for $S = \mathsf{unfold}_\Sigma(A)$, $T = \mathsf{unfold}_\Sigma(B)$ we have

1. If $S = \oplus\{\ell : A_\ell\}_{\ell \in L}$ then $T = \oplus\{\ell : B_\ell\}_{\ell \in L}$ and $(A_\ell, B_\ell) \in \mathcal{R}$ for all $\ell \in L$.
2. If $S = \&\{\ell : A_\ell\}_{\ell \in L}$ then $T = \&\{\ell : B_\ell\}_{\ell \in L}$ and $(A_\ell, B_\ell) \in \mathcal{R}$ for all $\ell \in L$.

3. If $S = A_1 \otimes A_2$, then $T = B_1 \otimes B_2$ and $(A_1, B_1) \in \mathcal{R}$ and $(A_2, B_2) \in \mathcal{R}$.
4. If $S = A_1 \multimap A_2$, then $T = B_1 \multimap B_2$ and $(A_1, B_1) \in \mathcal{R}$ and $(A_2, B_2) \in \mathcal{R}$.
5. If $S = \mathbf{1}$ then $T = \mathbf{1}$.
6. If $S = ?\{\phi\}.\, A'$ then $T = ?\{\psi\}.\, B'$ and either (i) $\vDash \phi$, $\vDash \psi$, and $(A', B') \in \mathcal{R}$, or (ii) $\vDash \neg\phi$ and $\vDash \neg\psi$.
7. If $S = !\{\phi\}.\, A'$ then $T = !\{\psi\}.\, B'$ and either (i) $\vDash \phi$, $\vDash \psi$, and $(A', B') \in \mathcal{R}$, or (ii) $\vDash \neg\phi$ and $\vDash \neg\psi$
8. If $S = \exists m.\, A'$ then $T = \exists n.\, B'$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in \mathcal{R}$.
9. If $S = \forall m.\, A'$ then $T = \forall n.\, B'$ and for all $i \in \mathbb{N}$, $(A'[i/m], B'[i/n]) \in \mathcal{R}$.

Definition 3.3. We say that A is *equal* to B, written $A \equiv B$, if there is a type bisimulation $\mathcal{R}$ such that $(A, B) \in \mathcal{R}$.

An interesting point here is provided by the cases *(ii)* in the clauses (6) and (7). Because the type must be closed, we know that ϕ and ψ will be either true or false. If both are false, no messages can be sent along a channel of either type and therefore the continuation types A' and B' are irrelevant when considering type equality.

Fundamentally, due to the presence of arbitrary recursion and therefore non-termination, we always view a type as a restriction of what a process *might* send or receive along some channel, but it is neither *required* to send a message nor *guaranteed* to receive one. This is similar to functional programming with unrestricted recursion where an expression may not return a value. The definition based on observability of messages is then somewhat strict, as exemplified by the next example.

Example 3.4. *Consider*

$$\begin{aligned}
\mathsf{bin}[n] = \oplus\{&\mathbf{b0} : \exists k.\, ?\{n = 2 * k\}.\, \mathsf{bin}[k],\\
&\mathbf{b1} : \exists k.\, ?\{n = 2 * k + 1\}.\, \mathsf{bin}[k],\\
&\mathbf{e} : ?\{n = 0\}.\, \mathbf{1}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{zero} = \oplus\{&\mathbf{b0} : \exists k.\, ?\{k = 0\}.\, \mathsf{zero},\\
&\mathbf{e} : ?\{0 = 0\}.\, \mathbf{1}\}
\end{aligned}$$

We might expect $\mathsf{bin}[0] \equiv \mathsf{zero}$*, but this is not so. A process of type* $\mathsf{bin}[0]$ *could send the label* $\mathbf{b1}$ *and maybe even, say,* 0 *for* k *and then just loop forever (because there is no proof of* $0 = 1$*). The type* zero *can not exhibit this behavior so the types are not equivalent.*

In our implementation, missing branches for a choice in process definitions are reconstructed with a continuation that marks it as impossible, which is then verified by the type checker. We found this simple technique significantly limited the need for subtyping or explicit definition of types such as zero—instead, we just work with $\mathsf{bin}[0]$.

The following properties of type equality are straightforward.

Lemma 3.4 (Properties of Type Equality). *The relation $\equiv$ is reflexive, symmetric, transitive and a congruence on closed valid types.*

3.3 Undecidability of Type Equality

We prove the undecidability of type equality by exhibiting a reduction from an undecidable problem about two counter machines.

The type system allows us to simulate two counter machines [106]. Intuitively, arithmetic constraints allow us to model branching zero-tests available in the machine. This, coupled with recursion in the language of types, establishes undecidability. Remarkably, a small fragment of our language containing only type definitions, internal choice ($\oplus$) and assertions ($?\{\phi\}.\ A$) where ϕ just contains constraints $n = 0$ and $n > 0$ is sufficient to prove undecidability. Moreover, the proof still applies if we treat types isorecursively. In the remainder of this section we provide some details of the undecidability proof.

Definition 3.5 (Two Counter Machine). A two counter machine $\mathcal{M}$ is defined by a sequence of instructions $\iota_1, \iota_2, \ldots, \iota_m$ where each instruction is one of the following.

- "$\mathsf{inc}(c_j); \mathsf{goto}\ k$" (increment counter j by 1 and go to instruction k)
- "$\mathsf{zero}(c_j)?\ \mathsf{goto}\ k : \mathsf{dec}(c_j); \mathsf{goto}\ l$" (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 $\iota_i = \mathsf{halt}$ and $c_1, c_2 \in \mathbb{N}$.

The *halting problem* refers to determining whether the computation of a two counter machine $\mathcal{M}$ with given initial values $c_1, c_2 \in \mathbb{N}$ is finite. Both the halting problem and its dual, the *non-halting problem*, are undecidable.

Theorem 3.6. *Given a valid signature Σ and two types A and B such that $m, n\ ;\ \top \vdash_\Sigma A, B$ valid. Then it is undecidable whether for concrete $i, j \in \mathbb{N}$ we have $A[i/m, j/n] \equiv B[i/m, j/n]$.*

Proof. Given a two counter machine, we construct a signature Σ and two types A and B with free arithmetic variables m and n such that the computation of the machine starting with initial counter values i and j is infinite iff $A[i/m, j/n] \equiv B[i/m, j/n]$ in Σ.

We define types $T_{\text{inf}} = \oplus\{\ell : T_{\text{inf}}\}$ and $T'_{\text{inf}} = \oplus\{\ell' : T'_{\text{inf}}\}$ for *distinct* labels ℓ and ℓ'. Next, for every instruction ι_i, we define types T_i and T'_i based on the form of the instruction.

- Case ($\iota_i = \mathsf{inc}(c_1); \mathsf{goto}\ k$): We define

$$\begin{array}{rcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_1 : T_k[c_1 + 1, c_2]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_1 : T'_k[c_1 + 1, c_2]\} \end{array}$$

- Case ($\iota_i = \mathsf{inc}(c_2); \mathsf{goto}\ k$): We define

$$\begin{array}{rcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_2 : T_k[c_1, c_2 + 1]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{inc}_2 : T'_k[c_1, c_2 + 1]\} \end{array}$$

- Case ($\iota_i = \mathsf{zero}(c_1)?\ \mathsf{goto}\ k : \mathsf{dec}(c_1); \mathsf{goto}\ l$): We define

$$\begin{array}{rcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_1 : ?\{c_1 = 0\}.\, T_k[c_1, c_2], \mathsf{dec}_1 : ?\{c_1 > 0\}.\, T_l[c_1 - 1, c_2]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_1 : ?\{c_1 = 0\}.\, T'_k[c_1, c_2], \mathsf{dec}_1 : ?\{c_1 > 0\}.\, T'_l[c_1 - 1, c_2]\} \end{array}$$

- Case ($\iota_i = \mathsf{zero}(c_2)?\ \mathsf{goto}\ k : \mathsf{dec}(c_2); \mathsf{goto}\ l$): We define

$$\begin{array}{rcl} T_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_2 : ?\{c_2 = 0\}.\, T_k[c_1, c_2], \mathsf{dec}_2 : ?\{c_2 > 0\}.\, T_l[c_1, c_2 - 1]\} \\ T'_i[c_1, c_2] & = & \oplus\{\mathsf{zero}_2 : ?\{c_2 = 0\}.\, T'_k[c_1, c_2], \mathsf{dec}_2 : ?\{c_2 > 0\}.\, T'_l[c_1, c_2 - 1]\} \end{array}$$

- Case ($\iota_i = \mathsf{halt}$): We define

$$\begin{array}{rcl} T_i[c_1, c_2] & = & T_{\text{inf}} \\ T'_i[c_1, c_2] & = & T'_{\text{inf}} \end{array}$$

We set type $A = T_1[m, n]$ and $B = T'_1[m, n]$. Now suppose, the counter machine $\mathcal{M}$ is initialized in the state $(1, i, j)$. The type equality question we ask is whether $T_1[i, j] \equiv T'_1[i, j]$. The two types only differ at the halting instruction. If $\mathcal{M}$ does not halt, the two types capture exactly the same communication behavior, since the halting instruction is never reached and they agree on all other instructions. If $\mathcal{M}$ halts, the first type proceeds with label ℓ and the second with ℓ' and they are therefore not equal. Hence, the two types are equal iff $\mathcal{M}$ does not halt. $\square$

We can easily modify this reduction for an isorecursive interpretation of types, by wrapping $\oplus\{\mathbf{unfold} : \_\_\}$ around the right-hand side of each type definition to simulate the unfold message. We also see that a host of other problems are undecidable, such as determining whether two types with free arithmetic variables are equal for all instances. This is the problem that arises while type-checking parametric process definitions.

3.4 A Practical Algorithm for Type Equality

Despite its undecidability, we have designed a coinductive algorithm for soundly approximating type equality. Similar to Gay and Hole's algorithm [66], it proceeds by attempting to construct a bisimulation. Due to the undecidability of the problem, our algorithm can terminate in three different states: (1) we have succeeded in constructing a bisimulation, (2) we have found a counterexample to type equality by finding a place where the types may exhibit different behavior, or (3) we have terminated the search without a definitive answer. From the point of view of type-checking, both (2) and (3) are interpreted as a failure to type-check (but there is a recourse; see subsection 3.4.2). Our algorithm is expressed as a set of inference rules where the execution of the algorithm corresponds to the bottom-up construction of a deduction. The algorithm is deterministic (no backtracking) and the implementation is quite efficient in practice (see Section 3.8).

One of the basic operations in Gay and Hole's algorithm is *loop detection*, that is, we have to determine that we have already added an equation $A \equiv B$ to the bisimulation we are constructing. Since we must treat *open types*, that is, types with free arithmetic variables subject to some constraints, determining if we have considered an equation already becomes a difficult operation. To that purpose we make an initial pass over the given type and introduce fresh *internal names* abstracted over their free type variables and known constraints. In the resulting signature defined type variables and type constructor alternates and we can perform loop detection entirely on type definitions (whether internal or external).

Example 3.5 (Queues, v3)**.** *After creating internal names $\%i$ for the type of queue we obtain the following signature (here parametric in A).*

$$
\begin{array}{ll}
\mathsf{queue}_A[n] = \&\{\mathbf{ins} : \%0[n], \mathbf{del} : \%1[n]\} & \\
\%0[n] = A \multimap \mathsf{queue}_A[n+1] & \%3 = \mathbf{1} \\
\%1[n] = \oplus\{\mathbf{none} : \%2[n], \mathbf{some} : \%4[n]\} & \%4[n] = ?\{n > 0\}.\,\%5[n] \\
\%2[n] = ?\{n = 0\}.\,\%3 & \%5[n \mid n > 0] = A \otimes \mathsf{queue}_A[n-1]
\end{array}
$$

Based on the invariants established by internal names, the algorithm only needs to compare two type variables or two structural types. The rules are shown in Figure 3.1. The judgment has the form $\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Gamma \vdash A \equiv B$ where $\mathcal{V}$ contains the free arithmetic variables in the constraints $\mathcal{C}$ and the types A and B, and Γ is a collection of *closures* $\langle \mathcal{V}' \; ; \; \mathcal{C}' \; ; \; V_1'[\overline{e_1}'] \equiv V_2'[\overline{e_2}'] \rangle$. If a derivation can be constructed, all ground instances of all closures are included in the resulting bisimulation (see the proof of Theorem 3.10). A ground instance $V_1'[\overline{e_1}'[\sigma']] \equiv V_2'[\overline{e_2}'[\sigma']]$ is given by a substitution σ' over variables in $\mathcal{V}'$ such that $\vDash \mathcal{C}'[\sigma']$.

The rules for type constructors simply compare the components. If the type constructors (or the label sets in the $\oplus$ and $\&$ rules) do not match, then type equality fails (having constructed a counterexample to bisimulation) unless the $\bot$ rule applies. This rules handles the case where the constraints are contradictory and no communication is possible.

$$\frac{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_\ell \equiv B_\ell \quad (\forall \ell \in L)}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash \oplus\{\ell : A_\ell\}_{\ell \in L} \equiv \oplus\{\ell : B_\ell\}_{\ell \in L}} \oplus$$

$$\frac{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_\ell \equiv B_\ell \quad (\forall \ell \in L)}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash \&\{\ell : A_\ell\}_{\ell \in L} \equiv \&\{\ell : B_\ell\}_{\ell \in L}} \&$$

$$\frac{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_1 \equiv B_1 \qquad \mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_2 \equiv B_2}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_1 \otimes A_2 \equiv B_1 \otimes B_2} \otimes$$

$$\frac{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_1 \equiv B_1 \qquad \mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_2 \equiv B_2}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A_1 \multimap A_2 \equiv B_1 \multimap B_2} \multimap \qquad \frac{}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash \mathbf{1} \equiv \mathbf{1}} \mathbf{1}$$

$$\frac{\mathcal{V} \,;\, \mathcal{C} \vDash \phi \leftrightarrow \psi \qquad \mathcal{V} \,;\, \mathcal{C} \wedge \phi \,;\, \Gamma \vdash A \equiv B}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash ?\{\phi\}.\, A \equiv ?\{\psi\}.\, B} ?$$

$$\frac{\mathcal{V} \,;\, \mathcal{C} \vDash \phi \leftrightarrow \psi \qquad \mathcal{V} \,;\, \mathcal{C} \wedge \phi \,;\, \Gamma \vdash A \equiv B}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash !\{\phi\}.\, A \equiv !\{\psi\}.\, B} ! \qquad \frac{\mathcal{V}, k \,;\, \mathcal{C} \,;\, \Gamma \vdash A[k/m] \equiv B[k/n]}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash \exists m.\, A \equiv \exists n.\, B} \exists^k$$

$$\frac{\mathcal{V}, k \,;\, \mathcal{C} \,;\, \Gamma \vdash A[k/m] \equiv B[k/n]}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash \forall m.\, A \equiv \forall n.\, B} \forall^k \qquad \frac{\mathcal{V} \,;\, \mathcal{C} \vDash \bot}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A \equiv B} \bot$$

$$\frac{\mathcal{V} \,;\, \mathcal{C} \vDash e_1 = e_1' \wedge \ldots \wedge e_n = e_n'}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash V[\overline{e}] \equiv V[\overline{e'}]} \mathsf{refl} \qquad \frac{\begin{array}{c} V_1[\overline{v_1} \mid \phi_1] = A \in \Sigma \qquad V_2[\overline{v_2} \mid \phi_2] = B \in \Sigma \\ \gamma = \langle \mathcal{V} \,;\, \mathcal{C} \,;\, V_1[\overline{e_1}] \equiv V_2[\overline{e_2}] \rangle \\ \mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma, \gamma \vdash A[\overline{e_1}/\overline{v_1}] \equiv B[\overline{e_2}/\overline{v_2}] \end{array}}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]} \mathsf{expd}$$

$$\frac{\langle \mathcal{V}' \,;\, \mathcal{C}' \,;\, V_1[\overline{e_1}'] \equiv V_2[\overline{e_2}'] \rangle \in \Gamma \qquad \mathcal{V} \,;\, \mathcal{C} \vDash \exists \mathcal{V}'.\, \mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]} \mathsf{def}$$

FIGURE 3.1: Algorithmic Rules for Type Equality

The rule of reflexivity is needed explicitly here (but not in the version of Gay and Hole) because due to the incompleteness of the algorithm we may otherwise fail to recognize type variables with equal index expressions as equal.

Now we come to the key rules, expd and def. In the expd rule we expand the definitions of $V_1[\overline{e_1}]$ and $V_2[\overline{e_2}]$, and we also add the closure $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, V_1[\overline{e_1}] \equiv V_2[\overline{e_2}] \rangle$ to Γ. Since the equality of $V_1[\overline{e_1}]$ and $V_2[\overline{e_2}]$ must hold for all its ground instances, the extension of Γ with the corresponding closure remembers exactly that.

In the def rule we close off the derivation successfully if all instances of the equation $V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]$ are already instances of a closure in Γ. This is checked by the entailment in the second premise, $\mathcal{V} \,;\, \mathcal{C} \vDash \exists \mathcal{V}'.\mathcal{C}' \wedge \overline{E_1} = \overline{e_1} \wedge \overline{E_2} = \overline{e_2}$. This entailment is verified as a closed $\forall\exists$ arithmetic formula, even if the original constraints $\mathcal{C}$ and $\mathcal{C}'$ do not contain any quantifiers. While for Presburger arithmetic we can decide such a proposition using quantifier elimination, other constraint domains may not permit such a decision procedure.

The algorithm so far is sound, but potentially nonterminating because when encountering variable/variable equations, we can use the expd rule indefinitely. To ensure termination, we restrict the expd rule to the case where *no* formula with the same type variables V_1 and V_2 is already present in Γ. This also removes the overlap between these two rules. Note that if type variables have no parameters, our algorithm specializes to Gay and Hole's (with the small optimizations of reflexivity and internal naming), which means our algorithm is sound and complete on unindexed types.

Example 3.6 (Integer Counter). *An integer counter with increment (*$\mathbf{inc}$*), decrement (*$\mathbf{dec}$*) and sign-test (*$\mathbf{sgn}$*) operations provides type* $\mathsf{intctr}[x, y]$*, where the current value of the counter is* $x - y$ *for natural numbers* x *and* y*.*

$$\begin{aligned}\mathsf{intctr}[x, y] = \&\{&\mathbf{inc} : \mathsf{intctr}[x + 1, y],\\ &\mathbf{dec} : \mathsf{intctr}[x, y + 1],\\ &\mathbf{sgn} : \oplus\{\mathbf{neg} : ?\{x < y\}.\, \mathsf{intctr}[x, y],\\ &\qquad\quad \mathbf{zer} : ?\{x = y\}.\, \mathsf{intctr}[x, y],\\ &\qquad\quad \mathbf{pos} : ?\{x > y\}.\, \mathsf{intctr}[x, y]\}\}\end{aligned}$$

Under this definition our algorithm verifies, for example, that an increment followed by a decrement does not change the counter value. That is,

$$x, y \;;\; \top \;;\; \cdot \vdash \mathsf{intctr}[x, y] \equiv \mathsf{intctr}[x + 1, y + 1]$$

where we have elided the assumptions $x, y \geq 0$*. When applying* expd*, we assume that* $\gamma = \langle x', y' \;;\; \top \;;\; \mathsf{intctr}[x', y'] \equiv \mathsf{intctr}[x' + 1, y' + 1]\rangle$*. Then, for example, in the first branch (for* inc*) we conclude* $x, y \;;\; \top \;;\; \gamma \vdash \mathsf{intctr}[x + 1, y] \equiv \mathsf{intcr}[x + 2, y + 1]$ *using the* def *rule and the entailment* $x, y \;;\; \top \vDash \exists x'.\, \exists y'.\, x' = x + 1 \wedge y' = y \wedge x' + 1 = x + 2 \wedge y' + 1 = y + 1$*. The other branches are similar.*

3.4.1 Soundness of the Type Equality Algorithm

We prove that the type equality algorithm is sound with respect to the definition of type equality. The soundness is proved by constructing a type bisimulation from a derivation of the algorithmic type equality judgment. We sketch the key points of the proofs.

The first gap we have to bridge is that the type bisimulation is defined only for closed types, because observations can only arise from communication along channels which, at runtime, will be of closed type. So, if we can derive $\mathcal{V} \;;\; \mathcal{C} \;;\; \cdot \vdash A \equiv B$ then we should interpret this as stating that for all ground substitutions σ over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$ we have $A[\sigma] \equiv B[\sigma]$.

Definition 3.7. Given a relation $\mathcal{R}$ on valid ground types and two types A and B such that $\mathcal{V} \;;\; \mathcal{C} \vdash A, B$ *valid*, we write $\forall \mathcal{V}.\, \mathcal{C} \Rightarrow A \equiv_{\mathcal{R}} B$ if for all ground substitutions σ over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$ we have $(A[\sigma], B[\sigma]) \in \mathcal{R}$.

Furthermore, we write $\forall \mathcal{V}.\, \mathcal{C} \Rightarrow A \equiv B$ if there exists a type bisimulation $\mathcal{R}$ such that $\forall \mathcal{V}.\, \mathcal{C} \Rightarrow A \equiv_{\mathcal{R}} B$.

Note that if $\mathcal{V} \,;\, \mathcal{C} \vDash \bot$, then $\forall \mathcal{V}.\, \mathcal{C} \Rightarrow A \equiv B$ is vacuously true, since there does not exist a ground substitution σ such that $\vDash \mathcal{C}[\sigma]$. A key lemma is the following, which is needed to show the soundness of the def rule.

Lemma 3.8. *Suppose $\forall \mathcal{V}'.\mathcal{C}' \Rightarrow V_1[\overline{e_1}'] \equiv_{\mathcal{R}} V_2[\overline{e_2}']$ holds. Further assume that $\mathcal{V} \,;\, \mathcal{C} \vDash \exists \mathcal{V}'.\mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}$ for some $\mathcal{V}, \mathcal{C}, \overline{e_1}, \overline{e_2}$. Then, $\forall \mathcal{V}.\mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv_{\mathcal{R}} V_2[\overline{e_2}]$ holds.*

Proof. To prove $\forall \mathcal{V}.\, \mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv_{\mathcal{R}} V_2[\overline{e_2}]$, it is sufficient to show that $V_1[\overline{e_1}[\sigma]] \equiv_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$ for any substitution σ over $\mathcal{V}$ such that $\vDash \mathcal{C}[\sigma]$. Applying this substitution to $\mathcal{V} \,;\, \mathcal{C} \vDash \exists \mathcal{V}'.\, \mathcal{C}' \wedge \overline{e_1}' = \overline{e_1} \wedge \overline{e_2}' = \overline{e_2}$, we infer $\exists \mathcal{V}'.\, \mathcal{C}' \wedge \overline{e_1}' = \overline{e_1}[\sigma] \wedge \overline{e_2}' = \overline{e_2}[\sigma]$ since $\vDash \mathcal{C}[\sigma]$. Thus, there exists σ' over $\mathcal{V}'$ such that $\vDash \mathcal{C}'[\sigma']$ holds, and $\overline{e_1}'[\sigma'] = \overline{e_1}[\sigma]$ and $\overline{e_2}'[\sigma'] = \overline{e_2}[\sigma]$. And since $\forall \mathcal{V}'.\, \mathcal{C}' \Rightarrow V_1[\overline{e_1}'] \equiv_{\mathcal{R}} V_2[\overline{e_2}']$, we deduce that for any ground substitution (including the current one) σ' over $\mathcal{V}'$, $V_1[\overline{e_1}'[\sigma']] \equiv_{\mathcal{R}} V_2[\overline{e_2}'[\sigma']]$ holds. This implies that $V_1[\overline{e_1}[\sigma]] \equiv_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$ since $\overline{e_1}'[\sigma'] = \overline{e_1}[\sigma]$ and $\overline{e_2}'[\sigma'] = \overline{e_2}[\sigma]$. ☐

We construct the bisimulation from a derivation of $\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A \equiv B$ by (i) collecting the conclusions of all the sequents, excepting only the def rule, and (ii) forming all ground instances from them.

Definition 3.9. Given a derivation $\mathcal{D}$ of $\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma \vdash A \equiv B$, we define the set $\mathcal{S}(\mathcal{D})$ of closures. For each sequent $\mathcal{V}' \,;\, \mathcal{C}' \,;\, \Gamma' \vdash A' \equiv B'$ (except the conclusion of the def rule) we include the closure $\langle \mathcal{V}' \,;\, \mathcal{C}' \,;\, A' \equiv B' \rangle$ in $\mathcal{S}(\mathcal{D})$.

Theorem 3.10. *If $\mathcal{V} \,;\, \mathcal{C} \,;\, \cdot \vdash A \equiv B$, then $\forall \mathcal{V}.\, \mathcal{C} \Rightarrow A \equiv B$.*

Proof. We are given a derivation $\mathcal{D}_0$ of $\mathcal{V}_0 \,;\, \mathcal{C}_0 \,;\, \cdot \vdash A_0 \equiv B_0$. Construct $\mathcal{S}(\mathcal{D}_0)$ and define a relation $\mathcal{R}$ on closed valid types as follows:

$$\mathcal{R} = \{(A[\sigma], B[\sigma]) \mid \langle \mathcal{V} \,;\, \mathcal{C} \,;\, A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0) \text{ and } \sigma \text{ over } \mathcal{V} \text{ with } \vDash \mathcal{C}[\sigma]\}$$

We first prove that $\mathcal{R}$ is a type bisimulation. Then our theorem follows since the closure $\langle \mathcal{V}_0 \,;\, \mathcal{C}_0 \,;\, A_0 \equiv B_0 \rangle \in \mathcal{S}(\mathcal{D}_0)$.

Consider $(A[\sigma], B[\sigma]) \in \mathcal{R}$ where $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0)$ for some σ over $\mathcal{V}$ and $\vDash \mathcal{C}[\sigma]$.

First, consider the case where $\mathcal{V} \,;\, \mathcal{C} \vDash \bot$. Under such a constraint, $\mathcal{V} \,;\, \mathcal{C} \,;\, \cdot \vdash A \equiv B$ holds true due to the $\bot$ rule. Furthermore, $\forall \mathcal{V}.\, \mathcal{C} \Rightarrow A \equiv B$ holds vacuously, and the algorithm is sound. For the remaining cases, we case analyze on the structure of $A[\sigma]$ and assume that there exists a ground substitution σ such that $\vDash \mathcal{C}[\sigma]$.

Consider the case where $A = \oplus\{\ell : A_\ell\}_{\ell \in L}$. Since A and B are both structural, $B = \oplus\{\ell : B_\ell\}_{\ell \in L}$. Since $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A \equiv B \rangle \in \mathcal{S}(\mathcal{D}_0)$, by definition of $\mathcal{S}(\mathcal{D}_0)$, we get $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A_\ell \equiv$

$B_\ell\rangle \in \mathcal{S}(\mathcal{D}_0)$ for all $\ell \in L$. By the definition of $\mathcal{R}$, we get that $(A_\ell[\sigma], B_\ell[\sigma]) \in \mathcal{R}$. Also, $A[\sigma] = \oplus\{\ell : A_\ell[\sigma]\}_{\ell \in L}$ and similarly, $B[\sigma] = \oplus\{\ell : B_\ell[\sigma]\}_{\ell \in L}$. Hence, $\mathcal{R}$ satisfies the appropriate closure condition for a type bisimulation.

Next, consider the case where $A = ?\{\phi\}.\, A'$. Since A and B are both structural, $B = ?\{\psi\}.\, B'$. Since $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A \equiv B\rangle \in \mathcal{S}(\mathcal{D}_0)$, we obtain $\mathcal{V} \,;\, \mathcal{C} \vDash \phi \leftrightarrow \psi$ and $\langle \mathcal{V} \,;\, \mathcal{C} \wedge \phi \,;\, A' \equiv B'\rangle \in \mathcal{S}(\mathcal{D}_0)$. Thus, for any substitution σ such that $\vDash \mathcal{C}[\sigma] \wedge \phi[\sigma]$, we get that $(A'[\sigma], B'[\sigma]) \in \mathcal{R}$ with $A[\sigma] = ?\{\phi[\sigma]\}.\, A'[\sigma]$ and $B[\sigma] = ?\{\psi[\sigma]\}.\, B'[\sigma]$. Since $\vDash \phi[\sigma]$ and and $\mathcal{V} \,;\, \mathcal{C} \vDash \phi \leftrightarrow \psi$ we also obtain $\vDash \psi[\sigma]$ and the closure condition is satisfied.

Next, consider the case where $A = \exists m.\, A'$. Since A and B are both structural, $B = \exists n.\, B'$. Since $\langle \mathcal{V} \,;\, \mathcal{C} \,;\, A \equiv B\rangle \in \mathcal{S}(\mathcal{D}_0)$, we get that $\langle \mathcal{V}, k \,;\, \mathcal{C} \,;\, A'[k/m] \equiv B'[k/n]\rangle \in \mathcal{S}(\mathcal{D}_0)$. Since k was chosen fresh and does not occur in $\mathcal{C}$, we obtain that for any $i \in \mathbb{N}$ we have $\vDash \mathcal{C}[\sigma, i/k]$ and therefore $(A'[\sigma, i/k], B'[\sigma, i/k]) \in \mathcal{R}$ for all $i \in \mathbb{N}$ and the closure condition is satisfied.

The only case where a conclusion is not added to $\mathcal{S}(\mathcal{D}_0)$ is the def rule. In this case, adding $(\forall \mathcal{V}.\, \mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv V_2[\overline{e_2}])$ is redundant: Theorem 3.8 states that $V_1[\overline{e_1}[\sigma]] \equiv_{\mathcal{R}} V_2[\overline{e_2}[\sigma]]$ which implies $(V_1[\overline{e_1}[\sigma]], V_2[\overline{e_2}[\sigma]]) \in \mathcal{R}$. □

3.4.2 Type Equality Declarations

Even though the type equality algorithm in Section 3.4 is incomplete, we have yet to find a natural example where it fails after we added reflexivity as a general rule. But since we cannot see a simple reason why this should be so, we made our type equality algorithm extensible by the programmer via an additional form of declaration

$$\forall \mathcal{V}.\, \mathcal{C} \Rightarrow V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]$$

in signatures. Let Γ_Σ denote the set of all such declarations. Then we check

$$\mathcal{V} \,;\, \mathcal{C} \,;\, \Gamma_\Sigma \vdash V_1[\overline{e_1}] \equiv V_2[\overline{e_2}]$$

for each such declaration, seeding the construction of a bisimulation with all the given equations. Then, when type-checking has to decide the equality of two types, it starts not with the empty context Γ but with Γ_Σ. Our soundness proof can easily accommodate this more general algorithm.

3.5 Formal Description of the Rast Language

In this section, we give a formal description of the Rast language. We have already seen the rules for statics and dynamics for the basic session types in Chapter 2. Hence, this section will primarily present the rules for the refinement layer. We have already described the grammar for types in Section 3.2. We now present the grammar for process expressions in Rast.

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

TABLE 3.1: Basic session types with operational description

$$\begin{array}{llll}
\text{Procs} & P, Q & ::= & x.k \ ; \ P \mid \mathsf{case}\ x\ (l \Rightarrow P)_{l \in L} \\
& & \mid & \mathsf{send}\ x\ y \ ; \ P \mid y \leftarrow \mathsf{recv}\ x \ ; \ P \\
& & \mid & \mathsf{close}\ x \mid \mathsf{wait}\ x \ ; \ P \\
& & \mid & x \leftrightarrow y \mid x \leftarrow f\ y \ ; \ P \\
& & \mid & \mathsf{assert}\ x\ \{\phi\} \ ; \ P \mid \mathsf{assume}\ x\ \{\phi\} \ ; P \\
& & \mid & \mathsf{send}\ x\ \{e\} \ ; \ P \mid \{n\} \leftarrow \mathsf{recv}\ x \ ; \ P
\end{array}$$

The typing judgment has the form of a sequent

$$\mathcal{V} \ ; \ \mathcal{C} \ ; \ \Delta \vdash_\Sigma P :: (x : A)$$

where $\mathcal{V}$ are index variables n, $\mathcal{C}$ are constraints over these variables expressed as a single proposition, Δ are the linear antecedents $x_i : A_i$, P is a process expression, and $x : A$ is the linear succedent. We propose and maintain that the x_i's and x are all distinct, and that all free index variables in $\mathcal{C}$, Δ, P, and A are contained among $\mathcal{V}$. Finally, Σ is a fixed signature containing type and process definitions. Because it is fixed, we elide it from the presentation of the rules. In addition we write $\mathcal{V} \ ; \ \mathcal{C} \vDash \phi$ for semantic entailment (proving ϕ assuming $\mathcal{C}$) in the constraint domain where $\mathcal{V}$ contains all arithmetic variables in $\mathcal{C}$ and ϕ. Table 3.1 reviews the basic session types their associated process terms, their continuation (both in types and terms) and operational description.

We formalize the operational semantics as a system of *multiset rewriting rules* [45]. We introduce semantic objects $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$ which mean that process P or message M provide along channel c. A process configuration is a multiset of such objects, where any two channels provided are distinct (formally described in Section 3.4.1).

Process Definitions Process definitions have the form $\Delta \vdash f[\overline{n}] = P :: (x : A)$ where f is the name of the process and P its definition. In addition, $\overline{n}$ is a sequence of arithmetic variables that Δ, P and A can refer to. All definitions are collected in a fixed global signature Σ. For a *well-formed signature*, we require that $\overline{n}$; $\top$; $\Delta \vdash P :: (x : A)$ for every definition, thereby allowing definitions to be mutually recursive. A new instance of a defined process f can be spawned with the expression $x \leftarrow f[\overline{e}]\ \overline{y}\ ;\ Q$ where $\overline{y}$ is a sequence of channels matching the antecedents Δ and $[\overline{e}]$ is a sequence of arithmetic expression matching the variables $[\overline{n}]$. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q.

$$\frac{\begin{array}{c}\overline{y' : B} \vdash f[\overline{n}] = P_f :: (x' : A) \in \Sigma \\ \Delta' = \overline{(y : B)}[\overline{e}/\overline{n}] \qquad \mathcal{V} \,;\, \mathcal{C} \,;\, \Delta, (x : A[\overline{e}/\overline{n}]) \vdash Q :: (z : C)\end{array}}{\mathcal{V} \,;\, \mathcal{C} \,;\, \Delta, \Delta' \vdash (x \leftarrow f[\overline{e}]\ \overline{y}\ ;\ Q) :: (z : C)}\ \mathsf{def}$$

The declaration of f is looked up in the signature Σ (first premise), and $\overline{e}$ is substituted for $\overline{n}$ while matching the types in Δ' and $\overline{y}$ (second premise). Similarly, the freshly created channel x has type A from the signature with $\overline{e}$ substituted for $\overline{n}$. The corresponding semantics rule also performs a similar substitution (a fresh).

$$(\mathsf{def}C) : \mathsf{proc}(c, x \leftarrow f[\overline{e}]\ \overline{d}\ ;\ Q) \ \mapsto\ \mathsf{proc}(a, P_f[a/x, \overline{d}/\overline{y'}, \overline{e}/\overline{n}]),\ \mathsf{proc}(c, Q[a/x])$$

where $\overline{y' : B} \vdash f[\overline{n}] = P_f :: (x' : A) \in \Sigma$.

Sometimes a process invocation is a tail call, written without a continuation as $x \leftarrow f[\overline{e}]\ \overline{y}$. This is a short-hand for $x' \leftarrow f[\overline{e}]\ \overline{y}\ ;\ x \leftrightarrow x'$ for a fresh variable x', that is, we create a fresh channel and immediately identify it with x.

Type Definitions As our queue example already showed, session types can be defined recursively, departing from a strict Curry-Howard interpretation of linear logic, analogous to the way pure ML or Haskell depart from a pure interpretation of intuitionistic logic. For this purpose we allow (possibly mutually recursive) type definitions $V[\overline{n} \mid \phi] = A$ in the signature Σ. Here, $\overline{n}$ denotes a sequence of arithmetic variables. Again, for a well-formed signature, we require A to be *contractive* [66] meaning A should not itself be a type name. Our type definitions are *equirecursive* so we can silently replace type names $V[\overline{e}]$ indexed with arithmetic refinements by $A[\overline{e}/\overline{n}]$ during type checking, and no explicit rules for recursive types are needed.

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

3.5.1 The Refinement Layer

We describe quantifiers ($\exists n.\ A$, $\forall n.\ A$) and constraints ($?\{\phi\}.\ A$, $!\{\phi\}.\ A$) [52]. An overview of the types, process expressions, and their operational meaning can be found in Table 3.2.

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

TABLE 3.2: Refined session types with operational description

Quantification The provider of $(c : \exists n.\, A)$ should send a witness i along channel c and then continue as $A[i/n]$. The witness is specified by an arithmetic expression e which, since it must be closed at runtime, can be evaluated to a number i (following standard evaluation rules of arithmetic). From the typing perspective, we just need to check that the expression e denotes a natural number, using only the permitted variables in $\mathcal{V}$. This is represented with the auxiliary judgment $\mathcal{V}\ ;\ \mathcal{C} \vdash e : \mathsf{nat}$ (implicitly proving that $e \geq 0$ under constraint $\mathcal{C}$).

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

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

Statically, the client adds n to $\mathcal{V}$ to ensure that Q and A are closed w.r.t. $\mathcal{V}$. Operationally, the provider sends the arithmetic expression with the continuation channel as a message that the client receives and appropriately substitutes.

$$(\exists S) : \mathsf{proc}(c, \mathsf{send}\ c\ \{e\}\ ;\ P) \ \mapsto\ \mathsf{proc}(c', P[c'/c]),\ \mathsf{msg}(c, \mathsf{send}\ c\ \{e\}\ ;\ c \leftrightarrow c')$$
$$(\exists C) : \mathsf{msg}(c, \mathsf{send}\ c\ \{e\}\ ;\ c \leftrightarrow c'),\ \mathsf{proc}(d, \{n\} \leftarrow \mathsf{recv}\ c\ ;\ Q) \ \mapsto\ \mathsf{proc}(d, Q[e/n][c'/c])$$

The dual type $\forall n.\, A$ reverses the role of the provider and client. The client sends (the value of) an arithmetic expression e which the provider receives and binds to n.

$$\frac{\mathcal{V}, n\ ;\ \mathcal{C}\ ;\ \Delta \vdash P_n :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \{n\} \leftarrow \mathsf{recv}\ x\ ;\ P_n :: (x : \forall n.\, A)}\ \forall R$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vdash e : \mathsf{nat} \qquad \mathcal{V}\ ;\ \Delta, (x : A[e/n]) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : \forall n.\, A) \vdash \mathsf{send}\ x\ \{e\}\ ;\ Q :: (z : C)}\ \forall L$$

$$(\forall S) : \mathsf{proc}(d, \mathsf{send}\ c\ \{e\}\ ;\ P) \ \mapsto\ \mathsf{msg}(c', \mathsf{send}\ c\ \{e\}\ ;\ c' \leftrightarrow c),\ \mathsf{proc}(d, [c'/c]P)$$
$$(\forall C) : \mathsf{proc}(d, \{n\} \leftarrow \mathsf{recv}\ c\ ;\ Q), \mathsf{msg}(c', \mathsf{send}\ c\ \{e\}\ ;\ c' \leftrightarrow c) \ \mapsto\ \mathsf{proc}(d, [e/n][c'/c]Q)$$

Constraints Refined session types also allow constraints over index variables. As we have already seen in the examples, these critically govern permissible messages. From the message-passing perspective, the provider of $(c : ?\{\phi\}.\, A)$ should send a proof of ϕ along c and the client should receive such a proof. However, since the index domain is decidable and future computation cannot depend on the form of the proof (what is known in type theory as *proof irrelevance*) such messages are not actually exchanged. Instead, it is the provider's responsibility to ensure that ϕ holds, while the client is permitted to assume that ϕ is true. Therefore, and in an analogy with imperative languages, we write $\mathsf{assert}\ c\ \{\phi\}\ ;\ P$ for a process that *asserts* ϕ for channel c and continues with P, while $\mathsf{assume}\ c\ \{\phi\}\ ;\ Q$ *assumes* ϕ and continues with Q.

Thus, the typing rules for this new type constructor are

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

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

Notice how the provider must verify the truth of ϕ given the currently known constraints $\mathcal{C}$ (the premise $\mathcal{V}\ ;\ \mathcal{C} \vDash \phi$), while the client assumes ϕ by adding it to $\mathcal{C}$.

Operationally, the provider creates a message containing the constraint that is received by the client (c' fresh).

$$(?S) : \mathsf{proc}(c, \mathsf{assert}\ c\ \{\phi\}\ ;\ P)\ \mapsto\ \mathsf{proc}(c', [c'/c]P),\ \mathsf{msg}(c, \mathsf{assert}\ c\ \{\phi\}\ ;\ c \leftrightarrow c')$$
$$(?C) : \mathsf{msg}(c, \mathsf{assert}\ c\ \{\phi\}\ ;\ c \leftrightarrow c'),\ \mathsf{proc}(d, \mathsf{assume}\ c\ \{\phi'\}\ ;\ Q)\ \mapsto\ \mathsf{proc}(d, [c'/c]Q)$$

In well-typed configurations (which arise from executing well-typed processes) the constraint ϕ in these rules will always be closed and true so there is no need to check this explicitly.

The dual operator $!\{\phi\}.\, A$ reverses the role of provider and client. The provider of $x : !\{\phi\}.\, A$ may assume the truth of ϕ, while the client must verify it. The dual rules are

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \land \phi\ ;\ \Delta \vdash P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ P :: (x : !\{\phi\}.\, A)}\ !R$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \quad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : !\{\phi\}.\, A) \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ !L$$

The remaining issue is how to type-check a branch that is impossible due to unsatisfiable constraints. For example, if a client sends a **del** request to a provider along $c : \mathsf{queue}_A[0]$, the type then becomes

$$c : \oplus\{\mathbf{none} : ?\{0{=}0\}.\, \mathbf{1}, \mathbf{some} : ?\{0{>}0\}.\, A \otimes \mathsf{queue}_A[0{-}1]\}$$

The client would have to branch on the label received and then assume the constraint asserted by the provider

$$\begin{aligned}&\mathsf{case}\ c\ \ (\mathbf{none} \Rightarrow \mathsf{assume}\ c\ \{0 = 0\}\ ;\ P_1\\ &\qquad\quad \mid \mathbf{some} \Rightarrow \mathsf{assume}\ c\ \{0 > 0\}\ ;\ P_2)\end{aligned}$$

but what could we write for P_2 in the **some** branch? Intuitively, computation should never get there because the provider can not assert $0 > 0$. Formally, we use the process expression '$\mathsf{impossible}$' to indicate that computation can never reach this spot:

$$\begin{aligned}&\mathsf{case}\ c\ \ (\mathbf{none} \Rightarrow \mathsf{assume}\ c\ \{0 = 0\}\ ;\ P_1\\ &\qquad\quad \mid \mathbf{some} \Rightarrow \mathsf{assume}\ c\ \{0 > 0\}\ ;\ \mathsf{impossible})\end{aligned}$$

In implicit syntax (see Section 3.7) we could omit the **some** branch altogether and it would be reconstructed in the form shown above. Abstracting away from this example, the typing rule for impossibility simply checks that the constraints are indeed unsatisfiable

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

There is no operational rule for this scenario since in well-typed configurations the process expression '$\mathsf{impossible}$' is dead code and can never be reached.

3.6 Type Safety

The main theorems that establish the deep connection between our refined type system and operational semantics are the usual *type preservation* and *progress*, also referred as *session fidelity* and *deadlock freedom*. At runtime, a program is represented using a set of semantic objects, i.e. processes and messages together defined as a *configuration*.

$$\mathcal{S}\ ::=\ \cdot \mid \mathcal{S}, \mathcal{S}' \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, M)$$

We say that $\mathsf{proc}(c, P)$ (or $\mathsf{msg}(c, M)$) provide channel c. We stipulate that no two distinct semantic objects provide the same channel.

Type Preservation A key question then is how to *type configurations*? We define a well-typed configuration using the judgment $\Delta_1 \Vdash_\Sigma \mathcal{S} :: \Delta_2$ denoting that configuration $\mathcal{S}$ uses channels Δ_1 and provides channels Δ_2. The rules for typing a configuration are defined in Figure 3.2. A configuration is always typed w.r.t. a *well-formed signature* Σ, requiring that all *(i)* all type definitions are valid and contractive, and *(ii)* all process definitions are well-typed. Since the signature Σ is fixed, we elide it from the presentation.

$$\frac{}{\Delta \Vdash (\cdot) :: \Delta}\ \mathsf{emp} \qquad \frac{\Delta_1 \Vdash \mathcal{S}_1 :: \Delta_2 \quad \Delta_2 \Vdash \mathcal{S}_2 :: \Delta_3}{\Delta_1 \Vdash (\mathcal{S}_1, \mathcal{S}_2) :: \Delta_3}\ \mathsf{comp}$$

$$\frac{\cdot\ ;\ \top\ ;\ \Delta \vdash P :: (x : A)}{\Delta \Vdash \mathsf{proc}(x, P) :: (x : A)}\ \mathsf{proc} \qquad \frac{\cdot\ ;\ \top\ ;\ \Delta \vdash M :: (x : A)}{\Delta \Vdash \mathsf{msg}(x, M) :: (x : A)}\ \mathsf{msg}$$

FIGURE 3.2: Typing rules for a configuration

The rule emp defines that an empty configuration provides all the channels Δ that it uses. The comp rule composes two configurations $\mathcal{S}_1$ and $\mathcal{S}_2$; $\mathcal{S}_1$ provides channels Δ_2 while $\mathcal{S}_2$ uses channels Δ_2. The rule proc creates a configuration out of a single process. Configurations only exist at runtime where all arithmetic expressions in process terms are closed, i.e. they do not refer to any free variables. Hence, we use $\mathcal{V} = \cdot$ and $\mathcal{C} = \top$ when typing process P (premise in proc rule). Similar to proc, the rule msg creates a configuration out of a single message (where a message is also represented as a process).

Global Progress To state progress, we need the notion of a *poised process* [116]. A process $\mathsf{proc}(c, P)$ is poised if it is trying to receive a message on c. Dually, a message $\mathsf{msg}(c, M)$ is poised if it is sending along c. A configuration is poised if every message or process in the configuration is poised. Conceptually, this means that the configuration is trying to communicate *externally* along one of the channels it uses or provides.

Theorem 3.11 (Type Safety). *For a well-typed configuration $\Delta_1 \Vdash_\Sigma \mathcal{S} :: \Delta_2$:*

(i) (Preservation) If $\mathcal{S} \mapsto \mathcal{S}'$, then $\Delta_1 \Vdash_\Sigma \mathcal{S}' :: \Delta_2$

(ii) (Progress) Either $\mathcal{S}$ is poised, or $\mathcal{S} \mapsto \mathcal{S}'$.

Proof. The proof of preservation proceeds by case analysis on the rules of operational semantics, applying inversion to the given typing derivation of $\mathcal{S}$, and then assembling a new derivation of $\mathcal{S}'$. Progress is proved by induction on the right-to-left typing of $\mathcal{S}$ so that either $\mathcal{S}$ is empty (and therefore poised) or $\mathcal{S} = (\mathcal{D}, \mathsf{proc}(c, P))$ or $\mathcal{S} = (\mathcal{D}, \mathsf{msg}(c, M))$. By induction hypothesis, $\mathcal{D}$ can either take a step (and then so can $\mathcal{S}$), or $\mathcal{D}$ is poised. In the latter case, we analyze the cases for P and M, applying multiple steps of inversion to show that in each case either $\mathcal{S}$ can take a step or is poised. □

3.7 Constraint Reconstruction

The process expressions introduced so far in the language follow simple syntax-directed typing rules. This means they are immediately amenable to be interpreted as an algorithm for type-checking, calling upon a decision procedure where arithmetic entailments and type equalities

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \quad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta\ {}_i\!\vdash P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta\ {}_i\!\vdash P :: (x : ?\{\phi\}.\ A)}\ ?R \qquad \frac{\mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Delta, (x : A)\ {}_i\!\vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : ?\{\phi\}.\ A)\ {}_i\!\vdash Q :: (z : C)}\ ?L$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Delta\ {}_i\!\vdash P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta\ {}_i\!\vdash P :: (x : !\{\phi\}.\ A)}\ !R \qquad \frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \quad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : A)\ {}_i\!\vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : !\{\phi\}.\ A)\ {}_i\!\vdash Q :: (z : C)}\ !L$$

FIGURE 3.3: Implicit Typing Rules

need to be verified. However, this requires the programmer to write a significant number of explicit process constructs pertaining to the refinement layer in their code. Relatedly, this hinders reuse: we are unable to provide multiple types to the same program so that it can be used in different contexts.

This section introduces an *implicit type system* in which the source program never contains the assume and assert expressions, i.e. constructs corresponding to proof constraints. Moreover, impossible branches may be omitted from case expressions. The missing branches and other constructs are restored by a type-directed process of *reconstruction*.

Interestingly, the nature of Presburger arithmetic makes full reconstruction impossible. For example, the proposition $\forall n.\, \exists k.\, (n = 2k \vee n = 2k + 1)$ is true but the witness for k as a Skolem function of n (namely $\lfloor n/2 \rfloor$) cannot be expressed in Presburger arithmetic. Since witnesses are critical for establishing correctness of programs, we require that type quantifiers $\forall n.\, A$ and $\exists n.\, A$ have explicit witnesses in processes and we do not reconstruct them.

In the first phase, a case expression with a missing branch for label ℓ is transformed into a branch $\ell \Rightarrow$ impossible so that type checking later verifies that the omitted branch is indeed impossible. Then assumes and asserts are inserted according to a reconstruction algorithm described in this section.

Following branch reconstruction, the resulting process expression is checked with the implicit typing judgment $\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta\ {}_i\!\vdash P :: (x : A)$. The implicit system differs from the explicit system in only one way: for the implicit constructs related to constraints ($!R$, $!L$, $?R$, $?L$), the process expression does not change on application of these rules. Selected typing rules are described in Figure 3.3 and illustrate that expressions P and Q are unchanged in the premise and conclusion. For the remaining rules pertaining to base session types and quantifiers ($\exists R$, $\exists L$, $\forall R$, $\forall L$), no reconstruction is involved and the implicit rules exactly match the explicit rules.

The implicit rules are sound and complete with respect to the explicit system, since from an implicit typing derivation we can read off the corresponding explicit process expression and vice versa. The rules are also manifestly decidable since the types in the premise are smaller than the conclusion for all the rules presented.

However, the implicit type system is highly nondeterministic. Since the process expressions do not change on the application of implicit rules in Figure 3.3, they can be applied in many different orders. And *each valid order corresponds to a different explicit program*, intuitively changing the order in which constraints are sent and received. Thus, an implicit source program may correspond to many different explicit programs. The necessary backtracking would greatly complicate error messages and would also be exponential and severely inefficient.

To solve this problem, we introduce a novel *forcing calculus* which enforces an order among these implicit constructs. The core idea of this calculus is to follow the structure of each type, but within that *assume should be inserted as early as possible, and assert should be inserted as late as possible.* This reasoning is sound since the constraints obey a *monotonicity property*: if a constraint is true at a program point, it will always be true later in the program. Thus, eagerly assuming and lazily asserting constraints is sound: if a constraint can be proved now, it can be proved later. It is also complete under the mild assumption that the types can be polarized (explained below). Logically, the $!R$, $?L$ rules are invertible, and are applied eagerly while their dual rules are applied lazily.

This strategy is formally realized in the forcing calculus using the judgment $\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \; ; \; \Omega \vdash P :: (x : A)$. The context is split into two: the linear context Δ contains stable propositions on which the invertible left rules have been applied, while the ordered context Ω stores channels on which invertible rules can possibly still be applied to. First, we assign polarities to the type operators with implicit expressions, a notion borrowed from focusing [20] with a similar function here. Type definitions are unfolded in order to determine their polarity, which is always possible since type definitions are contractive. The types that involve communication are called *structural* and represented by S.

$$\begin{array}{lcl}
A^+ & ::= & S \mid ?\{\phi\}.\, A^+ \\
A^- & ::= & S \mid !\{\phi\}.\, A^- \\
A & ::= & A^+ \mid A^- \\
S & ::= & \oplus\{\ell : A\}_{\ell \in L} \mid \&\{\ell : A\}_{\ell \in L} \mid A \otimes A \mid \mathbf{1} \mid A \multimap A \mid \exists n.\, A \mid \forall n.\, A
\end{array}$$

Not all types can be polarized in this manner, particularly types containing alternating proof constraints e.g., $!\{\phi\}.\, ?\{\psi\}.\, A$. When checking the validity of types before performing reconstruction we reject such types with alternating polarities. We also require that all process declarations contain only structural types at the top-level. Both these restrictions turn out to be mild in practice and can be resolved by introducing additional communications.

Thus, the $?$ operator is positive, while $!$ is negative. The structural types, denoted by S are considered neutral. In the forcing calculus, the invertible rules are applied first.

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \wedge \phi \; ; \; \Delta^- \; ; \; \Omega \vdash P :: (x : A^-)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \Omega \vdash P :: (x : !\{\phi\}.\, A^-)} \; !R$$

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \wedge \phi \; ; \; \Delta^- \; ; \; \Omega \cdot (x : A^+) \vdash P :: (z : C^+)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \Omega \cdot (x : ?\{\phi\}.\, A^+) \vdash P :: (z : C^+)} \; ?L$$

If a negative type is encountered in the ordered context, it is considered stable (invertible rules applied) and moved to Δ^-.

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^-, (x : A^-) \; ; \; \Omega \vdash P :: (z : C^+)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \Omega \cdot (x : A^-) \vdash P :: (z : C^+)} \; \mathsf{move}$$

The ordered context Ω imposes an order on the channels on which these invertible rules are applied.

Once all the invertible rules are applied, we reach a stable sequent of the form $\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \cdot \vdash P :: (x : A^+)$, i.e., the ordered context is empty and the provided type A^+ is positive. A stable sequent implies that all constraints have been received. We send a constraint lazily, i.e., just before communicating on that channel. We realize this by *forcing* the channel just before communicating on it. As an example, while sending (or receiving) a label on channel x, we force it effectively sending any pending constraints.

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \cdot \vdash x.k \; ; \; P :: [x : A^+]}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \cdot \vdash x.k \; ; \; P :: (x : A^+)} \; \oplus F_R$$

$$\frac{\mathcal{V} \; ; \mathcal{C} \; ; \Delta, [x : A^-] \; ; \cdot \vdash \mathsf{case}\; x \; (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C^+)}{\mathcal{V} \; ; \mathcal{C} \; ; \Delta, (x : A^-) \; ; \cdot \vdash \mathsf{case}\; x \; (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C^+)} \; \oplus F_L$$

The square brackets $[\cdot]$ indicates that the channel is forced, indicating that a communication is about to happen on it. If there are assert constructs pending on the forced channel, they are applied now.

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash \phi \qquad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \cdot \vdash P :: [x : A^+]}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \cdot \vdash P :: [x : ?\{\phi\}.\, A^+]} \; ?R$$

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash \phi \qquad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^-, [x : A^-] \; ; \; \cdot \vdash P :: (z : C^+)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^-, [x : !\{\phi\}.\, A^-] \; ; \; \cdot \vdash P :: (z : C^+)} \; !L$$

Finally, if a forced channel has a structural type, we apply the corresponding structural rule and *lose the forcing*. Again, as an example, we consider the internal choice operator.

$$\frac{(k \in L) \quad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \cdot \vdash P :: (x : A_k)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta^- \; ; \; \cdot \vdash (x.k \; ; \; P) :: [x : \oplus\{\ell : A_\ell\}_{\ell \in L}]} \; \oplus R_k$$

$$\frac{(\forall \ell \in L) \quad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \; ; \; (x : A_\ell) \vdash Q_\ell :: (z : C^+)}{\mathcal{V} \; ; \mathcal{C} \; ; \Delta, [x{:} \oplus \{\ell : A_\ell\}] \; ; \cdot \vdash \mathsf{case}\; x \; (\ell \Rightarrow Q_\ell) {::} (z{:}C^+)} \; \oplus L$$

In either case, applying the structural rule creates a possibly unstable sequent, thereby restarting the inversion phase.

Remarkably, *the forcing calculus is sound and complete with respect to the implicit type system*, assuming types can be polarized. Since every rule in the forcing calculus is also present in the implicit system, it is trivially sound. Moreover, applying assume eagerly, and assert lazily also turns out to be complete due to the monotonicity property of constraints.

Theorem 3.12 (Soundness and Completeness)**.** *For (valid) polarized types A and contexts Δ we have:*

1. *If $\mathcal{V}$; $\mathcal{C}$; $\Delta \;{}_i\!\vdash P :: (x : A)$, then $\mathcal{V}$; $\mathcal{C}$; $\cdot$; $\Delta \vdash P :: (x : A)$.*
2. *If $\mathcal{V}$; $\mathcal{C}$; $\cdot$; $\Delta \vdash P :: (x : A)$, then $\mathcal{V}$; $\mathcal{C}$; $\Delta \;{}_i\!\vdash P :: (x : A)$.*

Proof. Part (1) of Theorem 3.12 corresponds to soundness. The proof of soundness follows by induction on the implicit typing judgment. Intuitively, soundness follows from the simple observation that every rule in the forcing calculus is also valid in the implicit typing judgment. Theorem 3.12 part (2) corresponds to completeness whose proof proceeds by induction on the forcing judgment. The proof relies on two *key lemmas*: *(i)* the rules $!R$ and $?L$ are invertible, and *(ii)* if $\mathcal{V}$; $\mathcal{C}$; Δ^- ; $\Omega \vdash (x : A^+)$ and $\mathcal{V}$; $\mathcal{C} \vDash \phi$, then $\mathcal{V}$; $\mathcal{C}$; Δ^- ; $\Omega \vdash (x : ?\{\phi\}.\, A^+)$, i.e. asserting a constraint ϕ on a channel can be done at any program point where ϕ holds assuming $\mathcal{C}$, and thus, can be delayed. □

If a process is well-typed in the implicit system, it is well-typed using the forcing calculus. Once the typing derivation, i.e., ordering of the typing rules is fixed by the forcing calculus, *a unique explicit program is constructed by applying the explicit typing rules to the derivation.* Thus, if a reconstruction is possible, the forcing calculus will find it! We use this calculus to reconstruct the explicit program, which is then typechecked using the explicit typing system.

3.8 Implementation

We have implemented a prototype for Rast in Standard ML (8100 lines of code). This implementation contains a lexer and parser (1200 lines), reconstruction engine (900 lines), an arithmetic solver (1200 lines), a type checker (2500 lines), pretty printer (400 lines), and an interpreter (200 lines). The source code is well-documented and available open-source [56].

Syntax Table 3.3 describes the syntax for Rast programs. Each row presents the abstract and concrete representation of a session type, and its corresponding providing expression. A program contains a series of mutually recursive type and process declarations and definitions.

```
  type v{n}... = A
  decl f{n}... : (x1 : A1) ... (xn : An) |- (x : A)
  proc x <- f {n}... x1 ... xn = P
```

Listing 3.1: Top-Level Declarations

Abstract Types	**Concrete Types**	**Abstract Syntax**	**Concrete Syntax**
$\oplus\{l : A, \ldots\}$	`+{l : A, ...}`	$x.k$	`x.k`
$\&\{l : A, \ldots\}$	`&{l : A, ...}`	case x $(\ell \Rightarrow P)_{\ell \in L}$	`case x (l => P | ...)`
$A \otimes B$	`A * B`	send x w	`send x w`
$A \multimap B$	`A -o B`	$y \leftarrow$ recv x	`y <- recv x`
$\mathbf{1}$	`1`	close x	`close x`
		wait x	`wait x`
$\exists n.\, A$	`?n. A`	send x $\{e\}$	`send x {e}`
$\forall n.\, A$	`!n. A`	$\{n\} \leftarrow$ recv x	`{n} <- recv x`
$?\{\phi\}.\, A$	`?{phi}. A`	assert x $\{\phi\}$	`assert x {phi}`
$!\{\phi\}.\, A$	`!{phi}. A`	assume x $\{\phi\}$	`assume x {phi}`
$V[\overline{e}]$	`V{e1}{e2}...`		
		$x \leftrightarrow y$	`x <-> y`
		$x \leftarrow f\ x_1 \ldots x_n$	`x <- f x1 ... xn`

Table 3.3: Abstract and Corresponding Concrete Syntax for Types and Expressions

The first line is a *type definition*, where v is the type name with index variables $\overline{n}$ and A is its definition. The second line is a *process declaration*, where f is the process name, $(x_1 : A_1) \ldots (x_n : A_n)$ are the used channels and corresponding types, while the provided channel is x of type A. Finally, the last line is a *process definition* for the same process f defined using the process expression P. In addition, f can be parameterized by index variables $\overline{n}$. We use a hand-written lexer and shift-reduce parser to read an input file and generate the corresponding abstract syntax tree of the program. The reason to use a hand-written parser instead of a parser generator is to anticipate the most common syntax errors that programmers make and respond with the best possible error messages.

Validity Checking Once the program is parsed and its abstract syntax tree is extracted, we perform a validity check on it. We check that all index refinements, potentials, and delay operators are non-negative. We also check that all index expressions are closed with respect to the the index variables in scope, and similarly for type expressions. To simplify and improve the efficiency of the type equality algorithm, we also assign internal names to type subexpressions [52, 58] parameterized over their free type and index variables. These internal names are not visible to the programmer.

Reconstruction and Type Checking The programmer can use a flag in the program file to indicate whether they are using *explicit* or *implicit* syntax. If the syntax is explicit, the reconstruction engine performs no program transformation. However, if the syntax is implicit, we use the implicit type system to approximately type-check the program. Once completed, we use the forcing calculus to insert assert and assume constructs.

The implementation takes some care to provide constructive and precise error messages, in particular as session types are likely to be unfamiliar. One technique is staging: first check approximate type correctness, ignoring refinements and only if that check passes perform reconstruction and strict type checking. Another particularly helpful technique has been *type compression*. Whenever the type checker expands a type $V[\overline{e}]$ with $V[\overline{n}] = B$ to $B[\overline{e}/\overline{n}]$, we record a reverse mapping back to $V[\overline{e}]$. When printing types for error messages this mapping is consulted, and complex types may be compressed to much simpler forms, greatly aiding readability of error messages. This is feasible in part because all intermediate subexpressions have an explicit (internal) definition, simplifying the lookup. Finally, our implementation uses a bi-directional [52, 58] type checking algorithm which reconstructs intermediate types for each channel. This helps localize the source of the error message as the program point where reconstruction fails. We designed the abstract syntax tree to also contain the relevant source code location information which is utilized while generating the error message.

Arithmetic Solver To determine the validity of arithmetic propositions that is used by our refinement layer, we use a straightforward implementation of Cooper's decision procedure [48] for Presburger arithmetic. We found a small number of optimizations were necessary, but the resulting algorithm has been quite efficient and robust.

1. We eliminate constraints of the form $x = e$ (where x does not occur in e) by substituting e for x in all other constraints to reduce the total number of variables.
2. We exploit that we are working over natural numbers so all solutions have a natural lower bound, i.e., 0.

We also extend our solver to handle non-linear constraints. Since non-linear arithmetic is undecidable, in general, we use a normalizer which collects coefficients of each term in the multinomial expression.

1. To check $e_1 = e_2$, we normalize $e_1 - e_2$ and check that each coefficient of the normal form is 0.
2. To check $e_1 \geq e_2$, we normalize $e_1 - e_2$ and check that each coefficient is non-negative.
3. If we know that $x \geq c$, we substitute $y + c$ for x in the constraint that we are checking with the knowledge that the fresh $y \geq 0$.
4. We try to find a quick counterexample to validity by plugging in 0 and 1 for the index variables.

If the constraint does not fall in the above two categories, we print the constraint and trust that it holds. A user can then view these constraints manually and confirm their validity. At present, all of our examples pass without having to trust unsolvable constraints with our set of heuristics beyond Presburger arithmetic.

Interpreter The current version of the interpreter pursues a sequential schedule following a prior proposal [123]. We only execute programs that have no free type or index variables and only one externally visible channel, namely the one provided. When the computation finishes, the messages that were asynchronously sent along this distinguished channel are shown, while running processes waiting for input are displayed simply as a dash '–'.

The interpreter is surprisingly fast. For example, using a linear prime sieve to compute the status (prime or composite) or all number in the range $[2, 257]$ takes 27.172 milliseconds using MLton during our experiments (see machine specifications below).

3.9 Further Examples

We present several different kinds of examples from varying domains illustrating different features of the type system and algorithms. Table 3.4 describes the results: iLOC describes the lines of source code in implicit syntax, eLOC describes the lines of code after reconstruction (which inserts implicit constructs), #Defs shows the number of process definitions, R (ms) and T (ms) show the reconstruction and type-checking time in milliseconds respectively. Note that reconstruction is faster than type-checking since reconstruction does not involve solving any arithmetic propositions. The experiments were run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory.

1. **arithmetic**: natural numbers in unary and binary representation indexed by their value and processes implementing standard arithmetic operations.
2. **integers**: an integer counter represented using two indices x and y with value $x - y$.
3. **linlam**: expressions in the linear λ-calculus indexed by their size.
4. **list**: lists indexed by their size, and some standard operations such as *append, reverse, map, fold*, etc. Also provides and implementation of stacks and queues using lists.
5. **primes**: the sieve of Eratosthenes to classify numbers as prime or composite.
6. **segments**: type $\mathsf{seg}[n] = \forall k.\mathsf{list}[k] \multimap \mathsf{list}[n+k]$ representing partial lists with a constant-work append operation.
7. **ternary**: natural numbers and integers represented in balanced ternary form with digits $0, 1, -1$, indexed by their value, and a few standard operations on them. This example is noteworthy since it is the only one stressing the arithmetic decision procedure.
8. **theorems**: processes representing valid circular [60] proofs of simple theorems such as $n(k+1) = nk + n, n + 0 = n, n * 0 = 0$, etc.
9. **tries**: a trie data structure to store multisets of binary numbers, with constant amortized work insertion and deletion verified with ergometric types.

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

Table 3.4: Case Studies

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

Linear λ-Calculus We implemented the linear λ-calculus with evaluation (weak head normalization) of terms. We use higher-order abstract syntax, representing linear abstraction in the object language by a process receiving a message corresponding to its argument.

```
type exp = +{ lam : exp -o exp,
              app : exp * exp }
```

We would like evaluation to return a value (a λ-abstraction), so we take advantage of the structural nature of types (allowing us to reuse the label **lam**) to define the value type.

```
type val = +{ lam : exp -o exp }
```

Rast can infer that val is a subtype of exp. We can derive constructors *apply* for expressions and *lambda* for values (we do not need the corresponding constructor for expressions).

```
decl apply : (e1 : exp) (e2 : exp) |- (e : exp)
proc e <- apply e1 e2 =
  e.app ; send e e1 ; e <-> e2

decl lambda : (f : exp -o exp) |- (v : val)
proc v <- lambda f = v.lam ; v <-> f
```

As a simple example, we follow the representation of a combinator that swaps the arguments to a function.

```
(* swap = \f. \x. \y. (f y) x *)
decl swap : . |- (e : exp)
proc e <- swap =
  e.lam ; f <- recv e ;
  e.lam ; x <- recv e ;
  e.lam ; y <- recv e ;
  fy <- apply f y ;
  e <- apply fy x
```

Evaluation is now the following very simple process.

```
decl eval : (e : exp) |- (v : val)
proc v <- eval e =
  case e ( lam => v <- lambda e
         | app => e1 <- recv e ;      % e = e2
                  v1 <- eval e1 ;
                  case v1 ( lam => send v1 e ;
                                   v <- eval v1 ) )
```

If e sends a **lam** label, we just rebuild the expression as a value. If e sends an **app** label then e represents a linear application $e_1\, e_2$ and the continuation has type $\mathsf{exp} \otimes \mathsf{exp}$. This means we *receive* a channel representing e_1 and the continuation (still called e) behaves like e_2. We note this with a comment in the source. We then evaluate e_1 which exposes a λ-expression along the channel v_1. We send e along v_1, carrying out the reduction via communication. The result of this (still called v_1) is evaluated to yield the final value v. This program is available in the repository at `examples/linlam.rast`.

We would now like to prove that the value of a linear λ-expression is smaller than or equal to the original expression. At the same time we would like to rule out a class of so-called *exotic terms* in the representation, which are possible due to the presence of recursion in the meta-language. We achieve this by indexing the types exp and val with their *size*. For an application, this is easy: the size is one more than the sum of the sizes of the subterms.

```
type exp{n} = +{ lam : ...
                 app : ?n1. ?n2. ?{n=n1+n2+1}. exp{n1} * exp{n2}}
```

The size $n_2 + 1$ of a λ-expression is one more than the size n_2 of its body, but what is that in our higher-order representation? The body is a linear function takes an expression of size n_1 and then behaves like an expression of size $n_1 + n_2$. Solving for n_2 then gives use the following type definitions and types for the constructor processes.

```
type exp{n} = +{lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1},
                app : ?n1. ?n2. ?{n=n1+n2+1}. exp{n1} * exp{n2}}

type val{n} = +{ lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1} }

decl apply{n1}{n2} :
  (e1 : exp{n1}) (e2 : exp{n2}) |- (e : exp{n1+n2+1})
decl lambda{n2} :
  (f : !n1. exp{n1} -o exp{n1+n2}) |- (v : val{n2+1})
```

The universal quantification over n_1 in the type of **lam** is important, because a linear λ-expression may be applied to an argument of any size. We also cannot predict the size of the result of evaluation, so we have to use existential quantification: The value of an expression of size n will have size k for some $k \leq n$.

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

Because witnesses for quantifiers are not reconstructed, the evaluation process has to send and receive suitable sizes.

```
proc v <- eval{n} e =
  case e ( lam => send v {n} ;
                  v <- lambda{n-1} e
         | app => {n1} <- recv e ;
                  {n2} <- recv e ;
                  e1 <- recv e ;
                  v1 <- eval{n1} e1 ;
                  {k2} <- recv v1 ;
                  case v1 ( lam => send v1 {n2} ;
                                   send v1 e ;
                                   v2 <- eval{n2+k2-1} v1 ;
                                   {l} <- recv v2 ;
                                   send v {l} ; v <-> v2))
```

Type-checking now verifies that if evaluation terminates, the resulting value is smaller than the expression (or of equal size if the expression is already a value). The repository contains the implementation in the file `examples/linlam-size.rast`.

Trie Data Structure We now implement multisets of natural numbers (in binary form). One of the key questions is how to maintain linearity in the design of the data structure and interface. For example, should we be able to delete an element from the trie, not knowing a priori if it is even *in* the trie? To avoid exceedingly complex types to account for these situations, the process maintaining a trie offers an interface with two operations: insert (label **ins**) and delete (label **del**). We index the type `trie{n}` with the number of elements in the trie, so inserting an element always increases n by 1. If the element is already present, we just add 1 to its multiplicity. Deleting an element actually removes all copies of it and returns its multiplicity m. If the element is *not* in the trie, we just return a multiplicity of $m = 0$. In either case, the trie contains $n - m$ elements afterwards.

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

This type requires universal quantification over k, (written `!k`) which is the value of the number inserted into or deleted from the trie on each interaction (which is arbitrary).

The basic idea of the implementation is that each bit in the number `x : bin{k}` addresses a subtrie: if it is **b0** we descend into the left subtrie, if it is **b1** we descent into the right subtrie. If it is **e** we have found (or constructed) the node corresponding to x and we either increase its multiplicity (for insert), or respond with its multiplicity and set the new multiplicity to zero (for delete). We have two forms of processes: a *leaf* with zero elements and an interior node with $n_0 + m + n_1$ elements (where n_0 and n_1 and the number of elements in the left and right subtries, and m is the multiplicity of the number corresponding to this node in the trie).

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

The code is somewhat repetitive, so we only show the code for inserting an element into an interior node.

```
proc t <- node{n0}{m}{n1} l c r =
  case t (
    ins => {k} <- recv t ;
           x <- recv t ;
           case x ( b0 => {k'} <- recv x ;
                          l.ins ; send l {k'} ; send l x ;
                          t <- node{n0+1}{m}{n1} l c r
                  | b1 => {k'} <- recv x ;
                          r.ins ; send r {k'} ; send r x ;
                          t <- node{n0}{m}{n1+1} l c r
                  | e => wait x ;
                         c.inc ;
                         t <- node{n0}{m+1}{n1} l c r )
  | del => ...)
```

What does type-checking verify in this case? It shows that the number of elements in the trie increases and decreases as expected for each insert and delete operation. On the other hand, it does not verify that the *correct* multiplicities are incremented or decremented, which is beyond the reach of the current type system. The source code is at `examples/trie-work.rast`.

3.10 Related Work

The literature on session types is by now vast, so we focus our review of related work on *binary session types* (rather than multiparty session types) with *implementations* (rather than theoretical foundations). Among them, we can distinguish those that offer a library or embedding to a pre-existing language, and those that may be considered stand-alone language designs.

Libraries There are a number of libraries for session types. Such libraries tend to have a very different flavor from Rast because they focus on practical usability in the context of a general-purpose language. As such, the challenge usually is how to encode session types so programs can be statically checked against them and how to achieve the expected dynamic behavior. Among them we find libraries for Haskell [100, 112], Scala [130], OCaml [113], and Rust [89]. Noteworthy is the embedding of session types in ATS [147] because, unlike the others, ATS supports arithmetic indexing similar to Rast. The most recent library for Rust [46] is perhaps the closest to Rast in that it extends the exact basic system of session types from Chapter 2 with shared types [25]. While some of these libraries permit limited polymorphism, none of them support ergometric or temporal types.

Languages Designing complete languages like Rast frees the researcher from the limitations and idiosyncrasies of the host language as they explore the design space. A relatively early effort was the object-oriented language MOOL [141] which distinguishes linear and nonlinear channels.

A different style of language is SePi [24, 65] based on the π-calculus. It supports *linear* refinements in terms of uninterpreted propositions (which may reference integers) in addition to assert and assume primitives on them. They are not intended to capture internal properties of data structures of processes; instead, they allow the programmer to express some security properties.

The CO$_2$ middleware language [30, 31] supports *binary timed session types*. The notion of time here is *external*. As such, it does not measure work or span based on a cost model like Rast, but specifies interaction time windows for processes that can be enforced dynamically via monitors.

Concurrent C0 [144] is an implementation of linear and shared session types as an extension of C0, a small type-safe and memory-safe subset of C. It integrates the basic session types from Chapter 2 with shared session type [25] in the context of an imperative language. Relatedly, the Nomos language [57] integrates linear and shared ergometric session types with a functional language to aid smart contract programming. Although Nomos does not support temporal types and polymorphism, it embeds a linear programming solver to automatically infer the exact potential annotations.

Links [64, 101, 102] is a language aimed at developing web applications. While based on a different foundations, it is related to SILL [73, 138] in that both integrate traditional functional types with linear session types. As such, they can express many (nonlinear) programs that Rast cannot, but they support neither arithmetic refinements nor ergometric or temporal types.

Context-free session types [19, 137] generalize ordinary session types with sequential composition as well as permitting some polymorphism. The linear sublanguage of context-free session types can be modeled in Rast with nested polymorphism [58].

3.11 Conclusion

This chapter describes the Rast programming language. In particular, we focused on the concrete syntax, type checking and equality, and the refinement layer [51, 52]. The refinements rely on an arithmetic solver based on Cooper's algorithm [48]. The interpreter uses the shared memory semantics introduced in recent work [123]. We concluded with several examples demonstrating the efficacy of the refined type system in expressing and verifying properties about data structure sizes and values. All our examples have been verified with our system, and are available in an open-source repository [56].

In the future, we plan to address some limitations of the Rast language. One goal of Rast was to explore the boundaries of purely linear programming with general recursion. Often, this imposes a certain programming discipline and can be inconvenient if we need to drop or duplicate channels. Recent work on adjoint logic [122] uniformly integrates different logical layers into a unified language by assigning modes to communication. We plan to utilize this adjoint formulation to support shared [25] and unrestricted channels. Prior work on SILL [73] has demonstrated such an integration is helpful in general-purpose programming. With respect to refinements, we intend to pursue richer constraint domains such as non-linear arithmetic, particularly SMT.

Chapter 4

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 [36] 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.

4.1 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 : 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

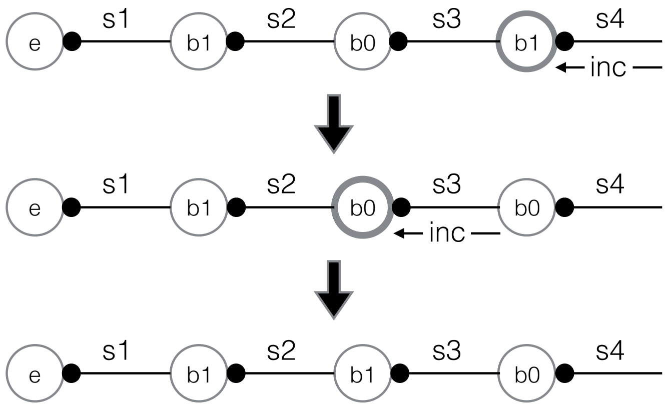

FIGURE 4.1: Binary counter system representing $5 = (101)_2$ with messages triggered when inc message is received on s_4.

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 4.1, the process $b0$ offers along channel s_3 (indicated by • between $b0$ and s_3) and uses channel s_2. This is formally written as

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

The *definitions* of e, $b0$, and $b1$ can be found in Figures 4.3 and 4.4. The only channel visible to an outside client (not shown) is s_4. Figure 4.1 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.

$$t : \mathsf{ctr} \;\vdash^0\; b0 :: (s : \mathsf{ctr}) \tag{4.1}$$
$$t : \mathsf{ctr} \;\vdash^1\; b1 :: (s : \mathsf{ctr}) \tag{4.2}$$
$$\cdot \;\vdash^0\; e :: (s : \mathsf{ctr}) \tag{4.3}$$

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 4.2 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, we need to define a more *refined* type.

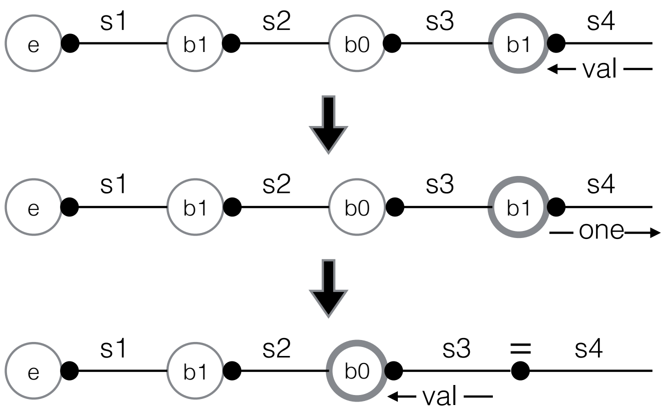

FIGURE 4.2: Binary counter system representing $5 = (101)_2$ with messages triggered when val message is received on s_4.

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. We have already set the stage for indexing types using refinements in Chapter 3. We now employ refinement type $\mathsf{ctr}[n]$ to denote a counter of value n. 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.

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

The type system verifies these types against the implementation of $b0$, $b1$, and e (see Figures 4.3 and 4.4, potential annotations marked in red). I will briefly explain the type derivation of $b0$, as shown in Figure 4.3 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 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

```
1:  $b0[n] :: (t : \mathsf{ctr}[n]) \vdash^{0} (s : \mathsf{ctr}[2n])$
2:    $s \leftarrow b0[n]\ t =$
3:      case $s$
4:        (inc $\Rightarrow$ get $s\ \{1\}$ ;                   % $(t : \mathsf{ctr}[n]) \vdash^{1} s : \mathsf{ctr}[2n+1]$
5:                $s \leftarrow b1[n]\ t$
6:        | val $\Rightarrow$ get $s\ \{2\lceil\log(2n+1)\rceil + 2\}$ ;  % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+1)\rceil+2} s : \mathsf{bits}$
7:                work $\{1\}$ ;                          % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+1)\rceil+2-1} s : \mathsf{bits}$
8:                $s$.zero ;                              % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+1)\rceil+1} s : \mathsf{bits}$
9:                work $\{1\}$ ;                          % $(t : \mathsf{ctr}[n]) \vdash^{2\lceil\log(2n+1)\rceil+1-1} s : \mathsf{bits}$
10:               $t$.val ;                               % $(t : \triangleleft^{2\lceil\log(n+1)\rceil+2}\mathsf{bits}) \vdash^{2\lceil\log(2n+1)\rceil} s : \mathsf{bits}$
11:               pay $t\ \{2\lceil\log(n+1)\rceil + 2\}$ ;   % $(t : \mathsf{bits}) \vdash^{2\lceil\log(2n+1)\rceil-2\lceil\log(n+1)\rceil-2} s : \mathsf{bits}$
12:               $s \leftrightarrow t)$                     % $(t : \mathsf{bits}) \vdash^{0} s : \mathsf{bits}$
```

FIGURE 4.3: Implementation for $b0$ process with its type derivation.

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 4.4.

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.

4.2 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 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.

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

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

FIGURE 4.4: Implementations for $b1$ and e processes with their type derivations.

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 4.5. 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 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.

$$
\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 \leftrightarrow d) \mapsto \mathsf{msg}(c, w + w', [c/d]M) & \\
(\mathsf{id}^-C) & \mathsf{proc}(c, w, c \leftrightarrow 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 \leftrightarrow c') & (c' \text{ fresh}) \\
(\oplus C) & \mathsf{msg}(c, w, c.k \; ; \; c \leftrightarrow 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' \leftrightarrow 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' \leftrightarrow 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 \leftrightarrow c') & (c' \text{ fresh}) \\
(\otimes C) & \mathsf{msg}(c, w, \mathsf{send}\ c\ e \; ; \; c \leftrightarrow 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' \leftrightarrow 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' \leftrightarrow 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 4.5: Cost semantics tracking total work for programs

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.

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

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.

4.3 Type System

The typing judgment has the form

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

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.

Σ defines the signature containing type and process definitions. It is defined as a finite set of type definitions $V = A$ and process definitions $\Delta \vdash^{q} f[\overline{n}] = P :: (x : A)$. The equation $V = A$ is used to define the type variable V as type expression A. We treat such definitions *equirecursively*. The process definition $\Delta \vdash^{q} f[\overline{n}] = P :: (x : A)$ defines a (possibly recursive) process named f parameterized by index variables $\overline{n}$ implemented by P providing along channel $x : A$ and using the channels Δ as a client, storing potential q. Because the signature is fixed, it is elided from the presentation of the rules.

Figure 4.6 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 $x \leftrightarrow y$ 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. Messages are typed exactly as processes.

In addition, the language has explicit rules for consuming and transfer of potential. Executing the work $\{w\}$ construct consumes w (non-negative) 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.

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

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

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash q = 0}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; y : A \vdash^{q} x \leftrightarrow y :: (x : A)} \; \mathsf{id}$$

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

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

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

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

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

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

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

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

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

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

FIGURE 4.6: Typing rules for resource-aware session types

equals what is gained by the recipient, thereby preserving the total potential of a configuration.

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

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

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

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash q \geq r \qquad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta, (x : A) \vdash^{q-r} Q :: (z : C)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta, (x : \triangleleft^{r} A) \vdash^{q} \mathsf{pay} \, x \, \{r\} \; ; \; Q :: (z : C)} \; \triangleleft L$$

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

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

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

FIGURE 4.7: Typing rules for a configuration

4.4 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 $V = A$ is contractive, and (b) every process definition $\Delta \vDash^{q} f[\overline{n}] = P :: (x : A)$ in Σ is well typed according to the process typing judgment, i.e. $\overline{n}\ ;\ \top\ ;\ \Delta \vDash^{q} P :: (x : A)$.

I use the following judgment to type a configuration.

$$\Delta_1 \overset{E}{\vDash}_{\Sigma} \mathcal{S} :: \Delta_2$$

It states that Σ is well-formed and that the configuration $\mathcal{S}$ uses the channels in the context Δ_1 and provides the channels in the context Δ_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 4.7. The rule empty defines that an empty configuration is well-typed with energy 0. The rule compose composes two configurations $\mathcal{S}$ and $\mathcal{S}'$; $\mathcal{S}$ provides service on the channels in Δ' while $\mathcal{S}'$ uses the channels in Δ'. The energy of the composed configuration $\mathcal{S}\ \mathcal{S}'$ 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 4.1 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 4.1 (Soundness). *Consider a well-typed configuration $\mathcal{S}$ w.r.t. a well-formed signature Σ such that $\Delta_1 \overset{E}{\vDash}_\Sigma \mathcal{S} :: \Delta_2$. If $\mathcal{S} \mapsto \mathcal{S}'$, then $\Delta_1 \overset{E}{\vDash}_\Sigma \mathcal{S}' :: \Delta_2$.*

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 $\mathcal{S}'$ 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 4.2 (Upper Bound). *If $\Delta_1 \overset{E}{\vDash}_\Sigma \mathcal{S} :: \Delta_2$, and $\mathcal{S} \mapsto^* \mathcal{S}'$, then $E \geq W'$, where W' is the total work performed by the configuration $\mathcal{S}'$, i.e. the sum of the work performed by each process and message in $\mathcal{S}'$. In particular, if the work done by the initial configuration $\mathcal{S}$ is 0, then the potential P of the initial configuration satisfies $P \geq W'$.*

Proof. Applying the Soundness theorem successively, we get that if $\mathcal{S} \mapsto^* \mathcal{S}'$ and $\Delta_1 \overset{E}{\vDash} \mathcal{S} :: \Delta_2$, then $\Delta_1 \overset{E}{\vDash} \mathcal{S}' :: \Delta_2$. Also, $E = P' + W'$, where P' is the total potential of $\mathcal{S}'$, while W' is the total work performed so far in $\mathcal{S}'$. 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 [138]. 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.

4.5 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 and size n).

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

The session type dictates that a process providing a service of type $\mathsf{store}_A[n]$ 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.

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

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 cost model), 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 $n > 0$).

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

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 with no potential 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

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{\, &\mathsf{ins} : \triangleleft^6 (A \multimap \mathsf{queue}_A[n+1]),\\ &\mathsf{del} : \triangleleft^2 \oplus \{\mathsf{none} : ?\{n = 0\}.\, \mathbf{1},\\ &\qquad\qquad \mathsf{some} : ?\{n > 0\}.\, A \otimes \mathsf{queue}_A[n-1]\}\}\end{aligned}$$

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.

4.6 Related Work

In the context of process calculi, capabilities [136] and static analyses [92] 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 [144]. 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., [47, 95]). 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 [84] and polynomial bounds [83] for sequential functional programs. It can also be integrated with program logics to derive bounds for imperative programs [21, 42]. Moreover, it has been used to derive bounds for term-rewrite systems [86] and object-oriented programs [85]. A recent work also considers bounds on the parallel evaluation cost (also called *span*) of functional programs [81]. 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. [68] 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. [15, 17] 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.

4.7 Future Directions

I briefly mention some important future directions with regard to work analysis.

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 [11] 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 5

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 [54]. 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).

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

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.

5.1 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:

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

5.1.1 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 5.4, 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

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

```
$y : \mathsf{bits} \vdash \mathit{copy} :: (x : \mathsf{bits})$          % No longer correct!
$x \leftarrow \mathit{copy} \leftarrow y =$
  case $y$  ( b0 ⇒ tick ; $x$.b0 ; $x \leftarrow \mathit{copy} \leftarrow y$
           | b1 ⇒ tick ; $x$.b1 ; $x \leftarrow \mathit{copy} \leftarrow y$
           | \$ ⇒ tick ; $x$.\$ ; wait $y$ ; tick ; 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

```
$y : \mathsf{bits} \vdash \mathit{copy} :: (x : \mathsf{bits})$
$x \leftarrow \mathit{copy} \leftarrow y =$
  case $y$  ( b0 ⇒                  % $y : \mathsf{bits} \vdash x : \mathsf{bits}$
                tick ; . . .)
```

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
```

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

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.

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

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 [55]).

5.1.2 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.

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

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

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.

5.2 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

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

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

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

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

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.

5.2.1 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 [118]. 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 [63] or Toninho et al. [139] 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 $(\mathsf{when?}\ c\ ;\ Q)$ 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.

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

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

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 $(\mathsf{when?}\ x\ ;\ Q)$ 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 $\mathsf{delayed}^{\Diamond}$ 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 5.2.

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.

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

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 ;                         % $y$ : bits $\vdash x$ : $\Diamond$sbits
                $x' \leftarrow \mathit{skip1s} \leftarrow y$ ;        % $x'$ : $\bigcirc\Diamond$sbits $\vdash x$ : $\Diamond$sbits
                $x \leftarrow \mathit{idle} \leftarrow x'$            % with $x'$ : $\bigcirc\Diamond$sbits $\vdash \mathit{idle}$ :: ($x$ : $\Diamond$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 $\quad y : \mathsf{bits} \vdash \mathit{skip1s} :: (x : \bigcirc\Diamond\mathsf{sbits})$

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.

5.2.2 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 [55]), 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.

5.2.3 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 $\Diamond A$.

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

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:

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

```
$c \leftarrow \mathit{bit0} \leftarrow d =$
   case $c$ ( inc $\Rightarrow c \leftarrow \mathit{bit1} \leftarrow d$                 % increment by continuing as $\mathit{bit1}$
            | val $\Rightarrow c.\mathsf{b0}\ ;\ d.\mathsf{val}\ ;\ c \leftarrow d$ )   % send b0 on $c$, send val on $d$, identify $c$ and $d$
$c \leftarrow \mathit{bit1} \leftarrow d =$
   case $c$ ( inc $\Rightarrow d.\mathsf{inc}\ ;\ c \leftarrow \mathit{bit0} \leftarrow d$   % send inc (carry) on $d$, continue as $\mathit{bit1}$
            | val $\Rightarrow c.\mathsf{b1}\ ;\ d.\mathsf{val}\ ;\ c \leftarrow d$ )   % send b1 on $c$, send val on $d$, identify $c$ and $d$
$c \leftarrow \mathit{empty} =$
   case $c$ ( inc $\Rightarrow e \leftarrow \mathit{empty}$ ;                   % spawn a new $\mathit{empty}$ process with channel $e$
                      $c \leftarrow \mathit{bit1} \leftarrow e$                % continue as $\mathit{bit1}$
            | val $\Rightarrow c.\$\ ;\ \mathsf{close}\ c$ )                   % 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.

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

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 5.2.

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

```
$c \leftarrow \mathit{bit0} \leftarrow d =$
   when? $c$ ;                              % $d : \bigcirc\mathsf{ctr} \vdash c : \&\{\ldots\}$
   case $c$ ( inc $\Rightarrow$ tick ;              % $d : \mathsf{ctr} \vdash c : \mathsf{ctr}$
                      $c \leftarrow \mathit{bit1} \leftarrow d$
            | val $\Rightarrow$ tick ;              % $d : \mathsf{ctr} \vdash c : \mathsf{bits}$
                      $c.\mathsf{b0}$ ;             % $d : \mathsf{ctr} \vdash c : \bigcirc\mathsf{bits}$
                      now! $d$ ; $d.\mathsf{val}$ ;  % $d : \bigcirc\mathsf{bits} \vdash c : \bigcirc\mathsf{bits}$
                      $c \leftarrow d$ )
```

```
$c \leftarrow \mathit{bit1} \leftarrow d =$
  when? $c$ ;                          % $d : \mathsf{ctr} \vdash c : \&\{\ldots\}$
  case $c$ ( inc $\Rightarrow$ tick ;           % $d : \mathsf{ctr} \vdash c : \mathsf{ctr}$        (see Section 5.2)
                 now! $d$ ; $d$.inc ;         % $d : \bigcirc\mathsf{ctr} \vdash c : \mathsf{ctr}$
                 $c \leftarrow \mathit{bit0} \leftarrow d$
           | val $\Rightarrow$ tick ;           % $d : \mathsf{ctr} \vdash c : \mathsf{bit}$        (see Section 5.2)
                 $c$.b1 ;                  % $d : \mathsf{ctr} \vdash c : \bigcirc\mathsf{bits}$
                 now! $d$ ; $d$.val ;         % $d : \bigcirc\mathsf{bits} \vdash c : \bigcirc\mathsf{bits}$
                 $c \leftarrow d$ )

$c \leftarrow \mathit{empty} =$
  when? $c$ ;                          % $\cdot \vdash c : \&\{\ldots\}$
  case $c$ ( inc $\Rightarrow$ tick ;           % $\cdot \vdash c : \mathsf{ctr}$
                 $e \leftarrow \mathit{empty}$ ;            % $e : \mathsf{ctr} \vdash c : \mathsf{ctr}$
                 $c \leftarrow \mathit{bit1} \leftarrow e$
           | val $\Rightarrow$ tick ; $c$.\$ ;      % $\cdot \vdash c : \bigcirc\mathbf{1}$
                 delay ; close $c$ )
```

5.2.4 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.

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
[\bigcirc A]_L^{-1} & = & A & [\bigcirc A]_R^{-1} & = & A & [x : A]_L^{-1} & = & x : [A]_L^{-1} \\
[\Box A]_L^{-1} & = & \Box A & [\Box A]_R^{-1} & = & \mathit{undefined} & [x : A]_R^{-1} & = & x : [A]_R^{-1} \\
[\Diamond A]_L^{-1} & = & \mathit{undefined} & [\Diamond A]_R^{-1} & = & \Diamond A & [\cdot]_L^{-1} & = & \cdot \\
[S]_L^{-1} & = & \mathit{undefined} & [S]_R^{-1} & = & \mathit{undefined} & [\Delta, \Delta']_L^{-1} & = & [\Delta]_L^{-1}, [\Delta']_L^{-1}
\end{array}
$$

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 5.1 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 $\mathsf{delayed}^{\Box}$ and A $\mathsf{delayed}^{\Diamond}$ are a shorthand for

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

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

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

FIGURE 5.1: Explicit Temporal Typing Rules

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

5.3 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.

5.3.1 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.

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

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 $A <: B$,

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

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.

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

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

5.3.2 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 5.1 (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 5.2 (Type Preservation). *If* $\Delta' \vDash \mathcal{C} :: \Delta$ *and* $\mathcal{C} \mapsto \mathcal{D}$ *then* $\Delta' \vDash \mathcal{D} :: \Delta$.

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

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

5.3.3 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 5.3 (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 5.4 (Global Progress). *If* $\cdot \vDash \mathcal{C} :: \Delta$ *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 5.3 to show that in each case either $\mathcal{C}$ can take a step or is poised. □

5.4 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 [68]) 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 [74] or explicitly polymorphic session types [40, 73]).

5.4.1 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 $`A$ 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,

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

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).

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

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

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 [55] omitting the tick constructs after each receive and send, and also the when? before each case that goes along with type $\Box A$.

```
$s \leftarrow elem \leftarrow x\ t =$
   case $s$  ( push $\Rightarrow$ $y \leftarrow$ recv $s$ ;
                    $s' \leftarrow elem \leftarrow x\ t$ ;        % previous top of stack, holding $x$
                    $s \leftarrow elem \leftarrow y\ s'$         % new top of stack, holding $y$
             | pop $\Rightarrow$  $s$.some ;
                    send $s\ x$ ;              % send channel $x$ along $s$
                    $s \leftarrow t$ )                % $s$ is now provided by $t$, via forwarding
$s \leftarrow empty =$
   case $s$  ( push $\Rightarrow$ $y \leftarrow$ recv $s$ ;
                    $e \leftarrow empty$ ;            % new bottom of stack
                    $s \leftarrow elem \leftarrow y\ e$
             | pop $\Rightarrow$  $s$.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 $\Box A$ has been received, there is a delay of an additional two units of time before the provider can accept the next request.

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

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

Time reconstruction will insert the additional delays in the *empty* process through subtyping, using $\Box\mathsf{queue}_A \leq \bigcirc\bigcirc\Box\mathsf{queue}_A$. 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 $s$ ;           % $y : \Box A \vdash s : \bigcirc\bigcirc\Box\mathsf{queue}_A$
                 $e \leftarrow \mathit{empty}$ ;                  % $y : \Box A, e : \Box\mathsf{queue}_A \vdash s : \bigcirc\bigcirc\Box\mathsf{queue}_A$
                 $s' \leftarrow \mathit{elem} \leftarrow y\ e$ ;     % $\Box\mathsf{queue}_A \leq \bigcirc\bigcirc\Box\mathsf{queue}_A$ (on $e$)
                 $s \leftarrow s'$                        % $\Box\mathsf{queue}_A \leq \bigcirc\bigcirc\Box\mathsf{queue}_A$ (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 [54] 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.

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

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.

5.4.2 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.

$\mathsf{bool} = \oplus\{\,\mathsf{b0} : {}^{\backprime}\mathbf{1}, \mathsf{b1} : {}^{\backprime}\mathbf{1}\,\}$

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

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.

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

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

```
$t \leftarrow \mathit{leaf} =$
  case $t$ ( parity $\Rightarrow$              % $\cdot \vdash t : \bigcirc^{5h+2}\,\mathsf{bool}$
                % delay$^{5h+2}$            % $\cdot \vdash t : \mathsf{bool}$
                $t$.b0 ;                    % $\cdot \vdash t : \mathbf{1}$
                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 $\Rightarrow$              % $l : \mathsf{tree}[h], r : \bigcirc^2\,\mathsf{tree}[h] \vdash t : \bigcirc^{5(h+1)+2}\,\mathsf{bool}$
                $l$.parity ;                % $l : \bigcirc^{5h+2}\,\mathsf{bool}, r : \bigcirc^1\mathsf{tree}[h] \vdash t : \bigcirc^{5(h+1)+1}\,\mathsf{bool}$
                % delay                     % $l : \bigcirc^{5h+1}\,\mathsf{bool}, r : \mathsf{tree}[h] \vdash t : \bigcirc^{5h+5}\,\mathsf{bool}$
                $r$.parity ;                % $l : \bigcirc^{5h}\,\mathsf{bool}, r : \bigcirc^{5h+2}\,\mathsf{bool} \vdash t : \bigcirc^{5h+4}\,\mathsf{bool}$
                % delay$^{5h}$              % $l : \mathsf{bool}, r : \bigcirc^2\,\mathsf{bool} \vdash t : \bigcirc^4\,\mathsf{bool}$
                $t \leftarrow \mathit{xor} \leftarrow l\ r$ )
```

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

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

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

5.5 Related Work

Most closely related is work on space and time complexity analysis of interaction nets by Gimenez and Moser [68], 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 [35] 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 [111], which I do not consider. Semantics counting communication costs for work and span in session-typed programs were given by Silva et al. [132], 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 [129]) or session-based communication primitives (see, for example, López et al. [103]) but generally these works do not develop a type system. Kobayashi [91] 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* [143] 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 [78], Esterel [33], or Lucid Synchrone [120] 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 [25, 39], 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 [110] temporal modality has been used to ensure productivity [93]. 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 [59]. 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. [62] 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 [81] 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 [34] use pipelining [114] to improve the complexity of parallel algorithms. However, they use futures [79], 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. [16] 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 [67] 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.

5.6 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 6

Session Types for 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 [109], Ethereum [145], and Tezos [72], 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 [108], insurance agreements [88], supply chain management [96], and mortgage loans [107]. 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 [22], which can have direct financial consequences. A well-known example, is the attack on The DAO [108], 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 [38]), or OCaml (Tezos' Liquidity [5]). 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 [105].

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* [54]. 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 [142] into a functional language. Linear type systems use the ideas of Girard's linear logic [69] to ensure that certain data is neither duplicated nor discarded by a program. Programming languages such as Rust [8] 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 [25, 39, 116, 138, 143].

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

1. Linear session types that support controlled sharing [25, 26] have been integrated into a conventional functional type system. To leave the logical foundation intact, the integration is achieved by a *contextual monad* [138] (Section 6.3) that gives process expressions first-class status in the functional language. Moreover, shared session types [25] are recast to accommodate the explicit notions of *contracts* and clients (Section 6.2).

2. I prove the type soundness of Nomos with respect to a novel asynchronous cost semantics using progress and preservation (Section 6.5).

6.1 Nomos by Example

This section provides an overview of the main features of Nomos based on a simple auction contract.

Explicit Protocols of Interaction Digital contracts, like traditional contracts, follow a *predefined protocol*. For instance, an auction contract distinguishes a bidding phase, where bidders submit their bids, possibly multiple times, from a subsequent collection phase, where the highest bidder receives the lot while all other bidders receive their bids back. In Solidity [1], the bidding phase of an auction is typically implemented as the bid function below. This function receives a bid (msg.value) from a bidder (msg.sender) and adds it to the bidder's total previous bids (bidValue).

```
function  bid() public payable {
  require (status == running);
  bidder = msg.sender; bid = msg.value;
  bidValue[bidder] = bidValue[bidder] + bid; }
```

To guarantee that a bid can only be placed in the bidding phase, the contract uses the state variable status to track the different phases of a contract. The require statement tests whether the auction is still running and thus accepts bids. It is checked at run-time and aborts the execution if the condition is not met. It is the responsibility of the programmer to define state variables, update them, and introduce corresponding guards.

Rather than burying the contract's interaction protocol in implementation code by means of state variables and run-time checks, Nomos allows the explicit expression and static enforcement of protocols with *session types*. The auction's protocol amounts to the below session type:

$$
\begin{array}{ll}
\mathsf{auction} = \uparrow_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{22} \oplus \{\textbf{running} : \&\{\textbf{bid} : \mathsf{id} \rightarrow \mathsf{money} \multimap \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}, & \text{\% recv bid from client} \\
\qquad\qquad\qquad\qquad \textbf{cancel} : \triangleright^{21} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}, & \text{\% client canceled} \\
\qquad\quad \textbf{ended} : \&\{\textbf{collect} : \mathsf{id} \rightarrow \oplus\{\textbf{won} : \mathsf{lot} \otimes \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}, & \text{\% client won} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{lost} : \mathsf{money} \otimes \triangleright^{7} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}, & \text{\% client lost} \\
\qquad\qquad\qquad \textbf{cancel} : \triangleright^{21} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}\} & \text{\% client canceled}
\end{array}
$$

We first focus on how the session type defines the main interactions of a contract with a bidder and ignore the operators $\uparrow_{\mathsf{L}}^{\mathsf{S}}$, $\downarrow_{\mathsf{L}}^{\mathsf{S}}$, $\triangleleft$, and $\triangleright$ for now. To distinguish the two main phases an auction can be in, the session type uses an internal choice ($\oplus$), leading the contract to either send the label **running** or **ended**, depending on whether the auction still accepts bids or not, respectively. Dual to an internal choice is an external choice ($\&$), which leaves the choice to the client (i.e., bidder) rather than the provider (i.e., contract). For example, in case the auction is running, the client can choose between placing a bid (label **bid**) or backing out (**cancel**). In the former case, the client indicates their identifier (type id), followed by a payment (type money). Nomos session types allow transfer of both non-linear (e.g., id) and linear assets (e.g., money), using the operators arrow ($\rightarrow$) and ($\multimap$), respectively. Should the auction have ended, the client can choose to check their outcome (label collect) or back out (cancel). In the case of collect, the auction will answer with either **won** or **lost**. In the former case, the auction will send the lot, in the latter case, it will return the client's bid. The linear product ($\otimes$) is dual to

$\multimap$ and denotes the transfer of a linear value from the contract to the client. The auction type guarantees that a client cannot collect during the running phase, while they cannot bid during the ended phase.

Nomos uses *shared* session types [25] to guarantee that bidders interact with the auction in mutual exclusion from each other and that the sequences of actions are executed *atomically*. To demarcate the parts of the protocol that become a *critical section*, the above session type uses the $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ and $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ modalities. The $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ modality denotes the beginning of a critical section, the $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ modality denotes its end. Programmatically, $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ translates into an *acquire* of the auction session and $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ into its *release*, which is only sound if the protocol behaves like an auction afterwards (*equi-synchronizing* type).

Contracts are implemented by *processes*, revealing the concurrent, message-passing nature of session-typed languages. The process *run* below implements the auction's running phase. Line 2 gives the process' signature, indicating that it offers a shared session of type auction along the channel sa and uses a linear hash map b : $\mathsf{hashmap}_{\mathsf{id,bid}}$ of bids indexed by id and a linear lot l. The bid session type (line 1) can be queried for the stored identifier and bid value, and is offered by a process (not shown) that internally stores this identifier and money. Line 4 onward list the process body. Line 1 defines session types bid and bids, respectively.

```
 1:  stype bid = $\&\{\mathbf{addr} : \mathsf{id} \times \mathsf{bid}, \mathbf{val} : \mathsf{money}\}$,   stype bids = $\mathsf{hashmap}_{\mathsf{id,bid}}$
 2:  $(b : \mathsf{bids}), (l : \mathsf{lot}) \vdash run :: (sa : \mathsf{auction})$        % syntax for process declaration
 3:    $sa \leftarrow run\ b\ l =$                                 % syntax for process definition
 4:      $la \leftarrow$ accept $sa$ ;                            % accept a client acquire request
 5:      $la$.running ;                                  % auction is running
 6:      case $la$ ( bid $\Rightarrow r \leftarrow$ recv $la$ ;             % receive identifier $r$ : id
 7:                      $m \leftarrow$ recv $la$ ;             % receive bid $m$ : money
 8:                      $sa \leftarrow$ detach $la$ ;          % detach from client
 9:                      $b' \leftarrow addbid\ r\ b\ m$ ;      % store bid internally
10:                      $sa \leftarrow check\ b'\ l$           % check if threshold reached
11:               | cancel $\Rightarrow sa \leftarrow$ detach $la$ ;     % detach from client
12:                         $sa \leftarrow run\ b\ l$)          % recurse
```

The contract process first *accepts* an acquire request by a bidder (line 4) and then sends the message running (line 5), indicating the auction status and waiting for the bidder's choice. Should the bidder choose to make a bid, the process waits to receive the bidder's identifier (line 6) followed by money equivalent to the bidder's bid (line 7). After this linear exchange, the process leaves the critical section by issuing a *detach* (line 8), matching the bidder's release request. Internally, the process stores the pair of the bidder's identifier and bid in the data structure bids (line 9). The ended protocol of the contract is governed by a different process (not shown), responsible for distributing the bids back to the clients. The contract transitions to the ended state when the number of bidders reaches a threshold (stored in auction). This is achieved by the *check* process (line 10) which checks if the threshold has been reached and makes this transition, or calls *run* otherwise. Should the bidder choose to cancel, the contract simply detaches and recurses (lines 11,12).

Re-Entrancy Vulnerabilities A contract function is re-entrant if, once called by a user, it can potentially be called again before the previous call has completed. As an illustration, consider the following collect function of the auction contract in Solidity where the funds are transferred to the bidder before the hash map is updated to reflect this change.

```
function  collect() public payable {
  require (status == ended);
  bidder = msg.sender; bid = bidValue[bidder];
  bidder.send(bid); bidValue[bidder] = 0; }

function () payable {
  // 'auction' variable stores the
  // address to auction contract
  auction.collect(); }
```

A bidder can now cause re-entrancy by creating a dummy contract with an unnamed *fallback* function (on the bottom) that calls the auction's collect function. This call is triggered when collect calls send (last line in collect), leading to an infinite recursive call to collect, depleting all funds from the auction. 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. To attempt re-entrancy, a bidder will need to acquire the auction contract twice without releasing it.

Linear Assets Nomos integrates a linear type system that tracks the assets stored in a process. The type system enforces that assets are never duplicated, but only exchanged between processes. Moreover, the type system prevents a process from terminating while it holds linear assets. For example, the auction contract treats money and lot as linear assets, which is witnessed by the use of the linear operators $\multimap$ and $\otimes$ for their exchange. In contrast, no provisions to handle assets linearly exist in Solidity, allowing such assets to be created out of thin air, duplicated, or discarded. In the above bid function, for instance, the language does not prevent the programmer from writing $\mathsf{bidValue}[\mathsf{bidder}] = \mathsf{bid}$ instead, losing the bidder's previous bid.

Resource Cost Another important aspect of digital contracts is their *resource usage*. On a blockchain, executing a contract function, or *transaction*, requires new blocks to be added to the blockchain. In existing blockchains like Ethereum, this is done by *miners* who charge a fee based on the *gas* usage of the transaction, indicating the cost of its execution. Precisely computing this cost statically is important because the sender of a transaction must pay this fee to the miners along with sending the transaction. If the sender does not pay a sufficient amount, the transaction will be rejected by the miners and the sender's fee is lost!

Nomos uses resource-aware session types [54] to statically analyze the resource cost of a transaction. 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. If the cost model assigns a cost to each operation as equivalent to their gas cost during execution, the potential consumed during a transaction reflects upper bound on the gas usage.

Resource-aware session types express the potential as part of the session type using the operators $\triangleleft$ and $\triangleright$. The $\triangleleft$ operator prescribes that the client must send potential to the contract, with the amount of potential indicated as a superscript. Dually, $\triangleright$ prescribes that the contract must send potential to the client. In case of the auction contract, we 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 an auction session is 22 (taking the max number of operations in all branches). Thus, we require the client to send 22 units of potential at the start of a session using $\triangleleft^{22}$. In the lost branch of the auction type, on the other hand, the contract returns 7 units of potential to the client using $\triangleright^{7}$. This simulates gas usage in smart contracts, where the sender initiates a transaction with some initial gas, and the leftover gas at the end of the transaction is returned to the sender. In contrast to existing smart contract languages like Solidity, which provide no support for analyzing the cost of a transaction, Nomos' type checker has automatically inferred these potential annotations and guarantees that well-typed transactions cannot run out of gas.

Bringing It All Together Combining all these features soundly in one language is challenging. In Nomos, we achieve this by using different *typing judgments* and *modes*, identifying the role of the process offered along that channel. The mode R denotes *purely linear processes* for linear assets or private data structures, such as b and l in the auction. The modes S and L denote *sharable processes*, i.e., contracts, that are either in their shared or linear phase such as sa and la, respectively. The mode T denotes a *transaction process* that can refer to shared and linear processes and is issued by a user, such as bidder in the auction. The mode assignment carries over into the process typing judgments imposing invariants (Definition 6.1) that are key to type safety. The mode annotations are automatically inferred by the type checker relieving programmers from this burden.

Typing Judgment For typing Nomos processes, we use the judgment $\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{q} P :: (x : A)$. As we introduce each concept, the role of each symbol will become clear. Henceforth, we indicate the current concepts from the judgment in black while the concepts that will be introduced in later sections are marked in blue.

6.2 Sharing Contracts

Multi-user support is fundamental to digital contract development. Linear session types, as defined in Chapter 3, 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, we base Nomos on *shared* session types [25]. 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* [32, 116, 126], session types are stratified into a linear and shared layer with two *adjoint modalities* going back and forth between them:

$$\begin{array}{lcll} A_{\mathsf{S}} & ::= & \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}} & \text{shared session type} \\ A_{\mathsf{L}} & ::= & \ldots \mid \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}} & \text{linear session types} \end{array}$$

The $\uparrow_{\mathsf{L}}^{\mathsf{S}}$ type modality translates into an *acquire*, while the dual $\downarrow_{\mathsf{L}}^{\mathsf{S}}$ 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. This is also the key behind eliminating *re-entrancy vulnerabilities* since it prevents a user from interrupting an ongoing session in the middle and initiating a new one.

Recall the process typing judgment in Nomos $\Psi \ ; \ \Gamma \ ; \ \Delta \vdash^{q} P :: (x_m : A)$ denoting a process P offering service of type A along channel x at mode m. The contexts Γ and Δ store the shared and linear channels that P can refer to, respectively (Ψ and q are explained later and thus marked in blue in Figure 6.2). The stratification of channels into layers arises from a difference in structural properties that exist for types at a mode. Shared propositions exhibit weakening, contraction and exchange, thus can be discarded or duplicated, while linear propositions only exhibit exchange.

Allowing Contracts to Rely on Linear Assets 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 assets (e.g., money or lot for the auction) regardless of its mode. Unfortunately, current shared session types [25] do not allow a shared process to rely on any linear channels, requiring any linear assets to be consumed before becoming shared. This precaution was logically motivated [124] and also crucial for type preservation.

$$\begin{array}{rcl}
A_{\mathsf{R}} & ::= & \oplus\{\ell : A_{\mathsf{R}}\}_{\ell \in L} \mid \&\{\ell : A_{\mathsf{R}}\}_{\ell \in L} \mid \mathbf{1} \mid A_m \multimap_m A_{\mathsf{R}} \mid A_m \otimes_m A_{\mathsf{R}} \\
& \mid & \tau \to A_{\mathsf{R}} \mid \tau \times A_{\mathsf{R}} \\
A_{\mathsf{L}} & ::= & \oplus\{\ell : A_{\mathsf{L}}\}_{\ell \in L} \mid \&\{\ell : A_{\mathsf{L}}\}_{\ell \in L} \mid \mathbf{1} \mid A_m \multimap_m A_{\mathsf{L}} \mid A_m \otimes_m A_{\mathsf{L}} \\
& \mid & \tau \to A_{\mathsf{L}} \mid \tau \times A_{\mathsf{L}} \mid \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}} \\
A_{\mathsf{S}} & ::= & \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}} \\
A_{\mathsf{T}} & ::= & A_{\mathsf{R}}
\end{array}$$

FIGURE 6.1: Grammar for shared session types

A key novelty of our work is to lift this restriction while *maintaining type preservation.* The main concern regarding preservation is to prevent a process from acquiring its client, which would result in a cycle in the linear process tree. To this end, we factorize the process typing judgment according to the *three roles* that arise in digital contract programs: *contracts*, *transactions*, and *linear assets*. Since contracts are shared and thus can oscillate between shared and linear, we get 4 sub-judgments for typing processes, each characterized by the mode of the channel being offered.

Definition 6.1 (Process Typing)**.** The judgment $\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{q} P :: (x_m : A)$ is categorized according to mode m. This factorization imposes certain invariants on the judgment outlined below. $\mathbf{L}(A)$ denotes the language generated by the grammar of A.

1. If $m = \mathsf{R}$, then (i) Γ is empty, (ii) for all $d_k \in \Delta \implies k = \mathsf{R}$, and (iii) $A \in \mathbf{L}(A_{\mathsf{R}})$.
2. If $m = \mathsf{S}$, then (i) for all $d_k \in \Delta \implies k = \mathsf{R}$, and (ii) $A \in \mathbf{L}(A_{\mathsf{S}})$.
3. If $m = \mathsf{L}$, then (i) for all $d_k \in \Delta \implies k = \mathsf{R} \vee k = \mathsf{L}$, and (ii) $A \in \mathbf{L}(A_{\mathsf{L}})$.
4. If $m = \mathsf{T}$, then $A \in \mathbf{L}(A_{\mathsf{T}})$.

Figure 6.1 shows the session type grammar in Nomos. The first sub-judgment in Definition 6.1 is for typing linear assets. These type a purely linear process P using a purely linear context Δ (types belonging to grammar A_{R} in Figure 6.1) and offering a purely linear type A along channel x_{R}. The mode R of the channel indicates that a purely linear session is offered. The second and third sub-judgments are for typing contracts. The second sub-judgment shows the type of a contract process P using a shared context Γ and a purely linear channel context Δ (judgment Δ purelin) and offering shared type A on the shared channel x_{S}. Once this shared channel is acquired by a user, the shared process transitions to its linear phase, whose typing is governed by the third sub-judgment. The offered channel transitions to linear mode L, while the linear context may now contain channels at modes R or L. Finally, the fourth typing judgment types a linear process, corresponding to a *transaction* holding access to shared channels Γ and linear channels Δ, and offering at mode T.

This novel factorization upholds preservation while allowing shared contract processes to rely on linear resources. The modes impose the ordering $\mathsf{R} < \mathsf{S} < \mathsf{L} < \mathsf{T}$ among the linear channels in the configuration. A process (offering a channel) at a certain mode is allowed to rely only on processes at the same or lower mode. These are exactly the conditions imposed by Definition 6.1. This introduces an implicit ordering among the linear processes depending on their mode, thus eliminating cycles in the process tree. Relatedly, shared processes can only refer to shared channels (at mode S) or purely linear channels (at mode R) as exemplified by the judgment Δ $\mathsf{purelin}$ in Figure 6.2. Formally, Δ $\mathsf{purelin}$ denotes that for all $d_k \in \Delta \implies k = \mathsf{R}$. This ensures that a shared contract must release all processes it has acquired before itself being released. This further enforces an ordering in which the channels are acquired and released, thus *allowing contracts to interact with other contracts without compromising type safety.*

Shared session types introduce new typing rules into our system, concerning the *acquire-release* constructs (see Figure 6.2). In rule $\uparrow_{\mathsf{L}}^{\mathsf{S}} L$, an acquire is applied to the shared channel $x_{\mathsf{S}} : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}}$ in Γ and yields a linear channel x_{L} added to Δ 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} (rule $\uparrow_{\mathsf{L}}^{\mathsf{S}} R$).

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

$$
\begin{aligned}
(\uparrow_{\mathsf{L}}^{\mathsf{S}} C) : {} & \mathsf{proc}(a_{\mathsf{S}}, w', x_{\mathsf{L}} \leftarrow \mathsf{accept}\ a_{\mathsf{S}}\ ;\ P_{x_{\mathsf{L}}}), \mathsf{proc}(c_m, w, x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ a_{\mathsf{S}}\ ;\ Q_{x_{\mathsf{L}}}) \mapsto \\
& \mathsf{proc}(a_{\mathsf{L}}, w', P_{a_{\mathsf{L}}}), \mathsf{proc}(c_m, w, Q_{a_{\mathsf{L}}})
\end{aligned}
$$

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 6.2. 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. 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}. Both right rules $\uparrow_{\mathsf{L}}^{\mathsf{S}} R$ and $\downarrow_{\mathsf{L}}^{\mathsf{S}} R$ require Δ $\mathsf{purelin}$ ensuring that a shared process releases all shared channels before themselves being released. Operationally, the release-detach rule is inverse to the acquire-accept rule.

$$
\begin{aligned}
(\downarrow_{\mathsf{L}}^{\mathsf{S}} C) : {} & \mathsf{proc}(a_{\mathsf{L}}, w', x_{\mathsf{S}} \leftarrow \mathsf{detach}\ a_{\mathsf{L}}\ ;\ P_{x_{\mathsf{S}}}), \mathsf{proc}(c_m, w, x_{\mathsf{S}} \leftarrow \mathsf{release}\ a_{\mathsf{L}}\ ;\ Q_{x_{\mathsf{S}}}) \mapsto \\
& \mathsf{proc}(a_{\mathsf{S}}, w', P_{a_{\mathsf{S}}}), \quad \mathsf{proc}(c_m, w, Q_{a_{\mathsf{S}}})
\end{aligned}
$$

6.3 Adding a Functional Layer

To support general-purpose programming patterns, Nomos combines linear channels with conventional data structures, such as integers, lists, or dictionaries. To reflect and track different

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

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta, (x_{\mathsf{L}} : A_{\mathsf{L}}) \vdash^q Q :: (z_m : C)}{\Psi \,;\, \Gamma, (x_{\mathsf{S}} : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}}) \,;\, \Delta \vdash^q x_{\mathsf{L}} \leftarrow \mathsf{acquire}\; x_{\mathsf{S}} \,;\, Q :: (z_m : C)} \uparrow_{\mathsf{L}}^{\mathsf{S}} L$$

$$\frac{\Delta\ \mathsf{purelin} \qquad \Psi \,;\, \Gamma \,;\, \Delta \vdash^q P :: (x_{\mathsf{L}} : A_{\mathsf{L}})}{\Psi \,;\, \Gamma \,;\, \Delta \vdash^q x_{\mathsf{L}} \leftarrow \mathsf{accept}\; x_{\mathsf{S}} \,;\, P :: (x_{\mathsf{S}} : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}})} \uparrow_{\mathsf{L}}^{\mathsf{S}} R$$

$$\frac{\Psi \,;\, \Gamma, (x_{\mathsf{S}} : A_{\mathsf{S}}) \,;\, \Delta \vdash^q Q :: (z_m : C)}{\Psi \,;\, \Gamma \,;\, \Delta, (x_{\mathsf{L}} : \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}}) \vdash^q x_{\mathsf{S}} \leftarrow \mathsf{release}\; x_{\mathsf{L}} \,;\, Q :: (z_m : C)} \downarrow_{\mathsf{L}}^{\mathsf{S}} L$$

$$\frac{\Delta\ \mathsf{purelin} \qquad \Psi \,;\, \Gamma \,;\, \Delta \vdash^q P :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Psi \,;\, \Gamma \,;\, \Delta \vdash^q x_{\mathsf{S}} \leftarrow \mathsf{detach}\; x_{\mathsf{L}} \,;\, P :: (x_{\mathsf{L}} : \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}})} \downarrow_{\mathsf{L}}^{\mathsf{S}} R$$

FIGURE 6.2: Typing rules corresponding to the shared layer.

classes of data in the type system, we take inspiration from prior work [116, 138] and incorporate processes into a functional core via a *linear contextual monad* that isolates session-based concurrency. To this end, we 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.

$$\begin{array}{rcl} \tau & ::= & \tau \to \tau \mid \tau + \tau \mid \tau \times \tau \mid \mathsf{int} \mid \mathsf{bool} \mid L^q(\tau) \\ & \mid & \{A_{\mathsf{R}} \leftarrow \overline{A_{\mathsf{R}}}\}_{\mathsf{R}} \mid \{A_{\mathsf{S}} \leftarrow \overline{A_{\mathsf{S}}} \,;\, \overline{A_{\mathsf{R}}}\}_{\mathsf{S}} \mid \{A_{\mathsf{T}} \leftarrow \overline{A_{\mathsf{S}}} \,;\, \overline{A}\}_{\mathsf{T}} \end{array}$$

The types are standard, except for the potential annotation $q \in \mathbb{N}$ in list type $L^q(\tau)$, which we explain in Section 6.4, and the contextual monadic types in the last line, which are the topic of this section. The expressivity of the types and terms in the functional layer are not important for the development in this paper. Thus, we do not formally define functional terms M but assume that they have the expected term formers such as function abstraction and application, type constructors, and pattern matching. We also define a standard type judgment for the functional part of the language.

$\Psi \Vdash^p M : \tau$ term M has type τ in functional context Ψ (potential p discussed later)

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 $\{A_{\mathsf{R}} \leftarrow \overline{A_{\mathsf{R}}}\}_{\mathsf{R}}$ denotes a process offering a *purely linear* session type A_{R} and using the purely linear vector of types $\overline{A_{\mathsf{R}}}$. The corresponding introduction form in the functional language is the monadic

$$\boxed{\Psi \; ; \; \Gamma \; ; \; \Delta \vDash^{q} P :: (x_m : A)} \quad \text{Process } P \text{ uses functional values in } \Psi.$$

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

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

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

FIGURE 6.3: Typing rules corresponding to the functional layer.

value constructor $\{c_{\mathsf{R}} \leftarrow P \leftarrow \overline{d_{\mathsf{R}}}\}$, denoting a runnable process offering along channel c_{R} that uses channels $\overline{d_{\mathsf{R}}}$, all at mode R. The corresponding typing rule for the monad is (ignore the blue portions)

$$\frac{\Delta = \overline{d_{\mathsf{R}} : D} \qquad \Psi \; ; \; \cdot \; ; \; \Delta \vDash^{q} P :: (x_{\mathsf{R}} : A)}{\Psi \Vdash^{q} \{x_{\mathsf{R}} \leftarrow P \leftarrow \overline{d_{\mathsf{R}}}\} : \{A \leftarrow \overline{D}\}_{\mathsf{R}}} \; \{\}I_{\mathsf{R}}$$

The monadic *bind* operation implements process composition and acts as the elimination form for values of type $\{A_{\mathsf{R}} \leftarrow \overline{A_{\mathsf{R}}}\}_{\mathsf{R}}$. The bind operation, written as $c_{\mathsf{R}} \leftarrow M \; \overline{d_{\mathsf{R}}} \; ; \; Q_c$, composes the process underlying the monadic term M, which offers along channel c_{R} and uses channels $\overline{d_{\mathsf{R}}}$, with Q_c, which uses c_{R}. The typing rule for the monadic bind is rule $\{\}E_{\mathsf{RR}}$ in Figure 6.3. 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 6.4. The shared context Γ is empty in accordance with the invariants established in Definition 6.1 *(i)*, since the mode of offered channel x is R. 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 rule (named $\mathsf{spawn}_{\mathsf{RR}}$) is given as follows:

$$\mathsf{proc}(d_{\mathsf{R}}, w, x_{\mathsf{R}} \leftarrow \{x'_{\mathsf{R}} \leftarrow P_{x'_{\mathsf{R}}, \overline{y}} \leftarrow \overline{y}\} \; \overline{a} \; ; \; Q) \mapsto \mathsf{proc}(c_{\mathsf{R}}, 0, P_{c_{\mathsf{R}}, \overline{a}}), \mathsf{proc}(d_{\mathsf{R}}, w, [c_{\mathsf{R}}/x_{\mathsf{R}}]Q)$$

The above rule spawns the process P offering along a globally fresh channel c_{R}, and using channels $\overline{a}$. The continuation process Q acts as a client for this fresh channel c_{R}. The other two monadic types correspond to spawning a shared process $\{A_{\mathsf{S}} \leftarrow \overline{A_{\mathsf{S}}} \; ; \; \overline{A_{\mathsf{R}}}\}_{\mathsf{S}}$ and a transaction process $\{A_{\mathsf{T}} \leftarrow \overline{A_{\mathsf{S}}} \; ; \; \overline{A}\}_{\mathsf{T}}$ at mode S and T, respectively. Their rules are analogous to $\{\}I_{\mathsf{R}}$ and $\{\}E_{\mathsf{RR}}$.

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

$$A ::= \ldots \mid \tau \to A \mid \tau \times A$$

The type $\tau \to A$ prescribes receiving a value of type τ with continuation type A, while its dual $\tau \times A$ prescribes sending a value of type τ with continuation A. The corresponding typing rules for arrow $(\to R, \to L)$ are given in Figure 6.3 (rules for $\times$ are inverse). Receiving a value adds it to the functional context Ψ, while sending it requires proving that the value has type τ. Semantically, sending a value $M : \tau$ creates a message predicate along a fresh channel c_m^+ containing the value:

$$\begin{aligned}(\to S) : \mathsf{proc}(d_k, w, \mathsf{send}\ c_m\ M\ ;\ P) \mapsto\ & \mathsf{msg}(c_m^+, 0, \mathsf{send}\ c_m\ M\ ;\ c_m^+ \leftrightarrow c_m), \\ & \mathsf{proc}(d_k, w, [c_m^+/c_m]P)\end{aligned}$$

The recipient process substitutes M for x, and continues to offer along the fresh continuation channel received by the message. This ensures that messages are received in the order they are sent. The rule is formalized below.

$$\begin{aligned}(\to C) : & \ \mathsf{proc}(c_m, w', x \leftarrow \mathsf{recv}\ c_m\ ;\ Q), \mathsf{msg}(c_m^+, w, \mathsf{send}\ c_m\ M\ ;\ c_m^+ \leftrightarrow c_m) \mapsto \\ & \quad \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m][M/x]Q)\end{aligned}$$

Tracking Linear Assets As an illustration, consider the type money introduced in the auction example (Section 6.1). The type is an abstraction over funds stored in a process and is described as

$$\begin{array}{lll}
\mathsf{money} = & & \\
\quad \&\{\mathbf{value} : \mathsf{int} \times \mathsf{money}, & & \%\ \text{send value} \\
\qquad \mathbf{add} : \mathsf{money} \multimap_{\mathsf{R}} \mathsf{money}, & & \%\ \text{receive money and add it} \\
\qquad \mathbf{subtract} : \mathsf{int} \to \oplus\{\mathbf{sufficient} : \mathsf{money} \otimes_{\mathsf{R}} \mathsf{money}, & & \%\ \text{receive int, send money} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{insufficient} : \mathsf{money}\} & & \%\ \text{funds insufficient to subtract} \\
\qquad \mathbf{coins} : \mathsf{list}_{\mathsf{coin}}\} & & \%\ \text{send list of coins}
\end{array}$$

The type supports querying for value, and addition and subtraction. The type also expresses insufficiency of funds in the case of subtraction. The provider process only supplies money to the client if the requested amount is less than the current balance, as depicted in the **sufficient** label. The type is implemented by a *wallet* process that internally stores a linear list of coins and an integer representing its value. Since linearity is only enforced on the list of coins in the linear context, we trust the programmer updates the integer in the functional context correctly during transactions. The process is typed and implemented as (modes of channels l and m is R, skipped in the definition for brevity)

```
1:  $(n : \mathsf{int})\ ;\ (l_\mathsf{R} : \mathsf{list}_\mathsf{coin}) \vdash \mathit{wallet} :: (m_\mathsf{R} : \mathsf{money})$
2:    $m \leftarrow \mathit{wallet}\ n\ l =$
3:      $\mathsf{case}\ m$                                   % case analyze on label received on $m$
4:        $(\mathbf{value} \Rightarrow \mathsf{send}\ m\ n\ ;$            % receive value, send $n$
5:                  $m \leftarrow \mathit{wallet}\ n\ l$
6:        $\mid \mathbf{add} \Rightarrow m' \leftarrow \mathsf{recv}\ m\ ;$         % receive $m'$ : money to add
7:                $m'.\mathbf{value}\ ;$                       % query value of $m'$
8:                $v \leftarrow \mathsf{recv}\ m'\ ;$
9:                $m'.\mathbf{coins}\ ;$                       % extract list of coins stored in $m'$
10:               $k \leftarrow \mathit{append}\ l\ m'\ ;$                % append list received to internal list
11:               $m \leftarrow \mathit{wallet}\ (n + v)\ k$
12:       $\mid \mathbf{subtract} \Rightarrow n' \leftarrow \mathsf{recv}\ m\ ;$     % receive int to subtract
13:                   $\mathsf{if}\ (n' > n)\ \mathsf{then}$
14:                     $m.\mathbf{insufficient}\ ;$             % funds insufficient
15:                     $m \leftarrow \mathit{wallet}\ n\ l$
16:                   $\mathsf{else}$
17:                     $m.\mathbf{sufficient}\ ;$               % funds sufficient
18:                     $l' \leftarrow \mathit{remove}\ n'\ l\ ;$             % remove $n'$ coins from $l$
19:                     $k \leftarrow \mathsf{recv}\ l'\ ;$                % and create its own list
20:                     $m' \leftarrow \mathit{wallet}\ n'\ k\ ;$           % new wallet process for subtracted funds
21:                     $\mathsf{send}\ m\ m'\ ;$                  % send new money channel to client
22:                     $m \leftarrow \mathit{wallet}\ (n - n')\ l'$
23:       $\mid \mathbf{coins} \Rightarrow m \leftrightarrow l)$
```

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 money (line 6), it queries the value of the received money m' (line 7), stores it in v (line 8), extracts the coins stored in m' (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 compares the requested amount against the stored funds. If the balance is insufficient, it sends the corresponding label, and recurses (lines 14 and 15). Otherwise, it removes n' coins using the *remove* process (line 18), creates a money abstraction using the *wallet* process (line 20), sends it (line 21) and recurses. Finally, if the *wallet* receives the message coins, it simply forwards its internal list along the offered channel.

6.4 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 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 it 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 bear 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. We discuss our approach of tracking resource usage, both at the functional and process layer.

Functional Layer Numerous techniques have been proposed to statically derive resource bounds for functional programs [23, 43, 50, 94, 125]. In Nomos, we adapt the work on automatic amortized resource analysis (AARA) [82, 84] that has been implemented in Resource Aware ML (RaML) [83]. RaML can automatically derive worst-case resource bounds for higher-order 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 *applyInterest* that iterates over a list of balances and applies interest on each element, multiplying them by a constant c. We use *tick* annotations to define the resource usage of an expression in this article. We have annotated the code to count the number of multiplications. The resource usage of an evaluation of *applyInterest* b is $|b|$.

```
let applyInterest balances =
  match balances with
  | [] -> []
  | hd::tl -> tick(1); (c*hd)::(applyInterest tl)
                (* consume unit potential for tick *)
```

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 example, we can derive the following resource-annotated type.

$$\textit{applyInterest} : L^1(\mathsf{int}) \xrightarrow{0/0} L^0(\mathsf{int})$$

The type $L^1(\mathsf{int})$ denotes a list of integers assigning a unit potential to each element in the list. The return value, on the other hand, has no potential. The annotation on the function arrow indicates that we do not need any 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 *applyInterest* again on the result of a call to the function. In this case, we would need to assign the type $L^1(\mathsf{int})$ to the resulting list and require $L^2(\mathsf{int})$ 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 [82, 83].

In Nomos, we simply adopt 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 $q \geq \mu$.

More details about AARA can be found in the literature [83, 84] and the Nomos supplementary material.

Process Layer To bound the resource usage of a process, Nomos features resource-aware session types [54] for work analysis. Resource-aware session types describe resource contracts for inter-process communication. The type system supports amortized analysis by assigning potential to both messages and processes. The derived resource bounds are functions of interactions between processes. As an illustration, consider the following resource-aware list interface from prior work [54].

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

The type prescribes that the provider of a list must send one unit of potential with every cons message that it sends. Dually, a client of this list will receive a unit potential with every cons message. All other type constructors are marked with potential 0, and exchanging the corresponding messages does not lead to transfer of potential.

While resource-aware session types in Nomos are equivalent to the existing formulation [54], our version is simpler and more streamlined. Instead of requiring every message to carry a potential (and potentially tagging several messages with 0 potential), we introduce two new type constructors for exchanging potential.

$$A ::= \ldots \mid \triangleright^r A \mid \triangleleft^r A$$

The type $\triangleright^r A$ requires the provider to pay r units of potential which are transferred to the client. Dually, the type $\triangleleft^r A$ requires the client to pay r units of potential that are received by the provider. Thus, the reformulated list type becomes

$$\mathsf{list}_A = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : \triangleright^1 (A \otimes \mathsf{list}_A)\}$$

The reformulation is more compact since we need to account for potential in only the typing rules corresponding to $\triangleright^r A$ and $\triangleleft^r A$.

With all aspects introduced, the process typing judgment

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

denotes a process P accessing functional variables in Ψ, shared channels in Γ, linear channels in Δ, offers service of type A along channel x at mode m and stores a non-negative constant potential q. Similarly, the expressing typing judgment

$$\Psi \Vdash^p M : \tau$$

denotes that expression M has type τ in the presence of functional context Ψ and potential p.

Figure 6.4 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 $x_m : \triangleleft^r A$ 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 $x_m : \triangleleft^r A$ in Δ using the *pay* construct, and the continuation remains with p units of potential. The typing rules for the dual constructor $\triangleright^r A$ 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.

The tick construct is used to simulate a cost model in Nomos. If an operation (e.g., sending a message, calling a function, etc.) has a cost of r, this cost is simulated by inserting tick (r)

$$\boxed{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{q} P :: (x_m : A)} \quad \text{Process } P \text{ has potential } q.$$

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

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

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

FIGURE 6.4: Selected typing rules corresponding to potential.

just before the operation. Then, the tick operations are the only ones that cost potential, thus simplifying the type system. These tick operations are automatically inserted by the Nomos type checker, using a predefined cost model that assigns a constant cost to each operation. In addition, our implementation provides some standard cost models, for instance, that assign a unit cost to each internal operation and sending a message.

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 $\{\}E_{\mathsf{RR}}$ from Figure 6.3. A process storing potential $r = p + q$ can spawn a process corresponding to the monadic expression 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 [83] describes that the base types in Ψ are copied to both Ψ_1 and Ψ_2, but the potential is split up. For instance, $L^{q_1+q_2}(\tau) \curlyvee (L^{q_1}(\tau), L^{q_2}(\tau))$. The rule $\rightarrow L$ in Figure 6.3 follows a similar pattern. A process Q storing $r = p + q$ potential sends a monadic expression M needing p units of potential to evaluate, and the continuation remains with q units of potential to execute. The p units of potential are consumed to evaluate M to a value before sending since only values are exchanged at runtime. Thus, the combination of the two type systems is smooth, assigning a uniform meaning to potential, both for the functional and process layer. Remarkably, this technical device of exchanging functional values can be used to exchange non-constant potential with messages. For instance, exchanging a list $l : L^{q}(\tau)$ will exchange $q \cdot n$ units of potential where n is the size of the list l.

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

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

This rule describes that if an expression M evaluates to V with cost μ, then the process $P[M]$ depending on monadic expression M 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 (or message) 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.

6.5 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, Definition 6.1 asserts certain invariants on process typing judgment depending on the mode of the channel offered by a process. This mode, remains invariant, as the process evolves. This is ensured by the process typing rules, which remarkably preserve these invariants despite being parametric in the mode.

Lemma 6.2 (Invariants)**.** *The typing rules on the judgment $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} (x_m : A)$ preserve the invariants outlined in Definition 6.1, i.e., if the conclusion satisfies the invariant, so do all the premises.*

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).

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

A key question is how to type these configurations because a configuration both uses and provides a number of channels. The solution is to have the typing impose a partial order among the processes and messages, requiring the provider of a channel to appear before its client. We stipulate that no two distinct processes or messages in a well-formed configuration provide the same channel c.

The typing judgment for configurations has the form $\Sigma\ ;\ \Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta)$ 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 their linear counterparts offered by a contract process, switching back and forth between them when the channel is acquired or released respectively. This mapping, along with the type of the shared channels, is stored in Γ_0. E is a natural number and stores the sum of the total potential and work as recorded in each process and message. We call E the energy of the configuration. The supplement details the configuration typing rules.

Finally, Σ denotes a signature storing the type and function definitions. A signature is well-formed if *(i)* every type definition $V = A_V$ is *contractive* [66] and *(ii)* 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; every process is encapsulated inside a function using the contextual monad.

Theorem 6.3 (Type Preservation)**.**

- *If a closed well-typed expression* $\cdot \Vdash^{q} M : \tau$ *evaluates to a value, i.e.,* $M \Downarrow V \mid \mu$*, then* $q \geq \mu$ *and* $\cdot \Vdash^{q-\mu} V : \tau$*.*
- *Consider a closed well-formed and well-typed configuration* Ω *such that* $\Sigma \ ; \ \Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \ ; \ \Delta)$*. If the configuration takes a step, i.e.* $\Omega \mapsto \Omega'$*, then there exist* Γ_0', Γ' *such that* $\Sigma \ ; \ \Gamma_0' \overset{E}{\vDash} \Omega' :: (\Gamma' \ ; \ \Delta)$*, i.e., the resulting configuration is well-typed. Additionally,* $\Gamma_0 \subseteq \Gamma_0'$ *and* $\Gamma \subseteq \Gamma'$*.*

The preservation theorem is standard for expressions [83]. For processes, we proceed by induction on the operational cost semantics and inversion on the configuration and process typing judgment.

To state progress, we need the notion of a *poised* process [116]. A process $\mathsf{proc}(c_m, w, P)$ is poised if it is trying to receive a message on c_m. Dually, a message $\mathsf{msg}(c_m, w, N)$ 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 process can be *blocked* [25] if it is trying to acquire a contract process that has already been acquired by some process. This can lead to the possibility of deadlocks.

Theorem 6.4 (Progress)**.** *Consider a closed well-formed and well-typed configuration* Ω *such that* $\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \ ; \ \Delta)$*. 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 linear session types, where progress guarantees deadlock freedom due to absence of shared channels.

6.6 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 [9] 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 [3], 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 [7] is formally modeled by the ρ-calculus, a reflective higher-order extension of the π-calculus. Michelson [6] is a purely functional stack-based language that has no side effects. Liquidity [5] is a high-level language that complies with the security restrictions of Michelson. Scilla [131] 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 [138]. However, this integration does not account for resource usage, nor sharing. Similarly, shared session types [25] have previously not been integrated with a functional layer or tracked for resource usage. Moreover, existing shared session types [25] 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 [83] and the process layer [54], but the two have never been integrated together.

6.7 Future Directions

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 7

Implementation of Nomos

Chapter 6 described the theory of Nomos, along with its static and dynamic semantics and type safety theorems. This chapter complements the previous chapter by describing the implementation of Nomos along with specific features making it easier to write smart contracts.

7.1 Overview of Nomos with an Auction Implementation

We highlight the main features of the Nomos language using the implementation of an auction contract in concrete syntax. An auction operates in two phases: a *running* phase where bidders bid into the auction followed by an *ended* phase where bidders collect their earnings. If a bidder wins the auction, they receive the *lot*, otherwise they receive their bids back.

The first key idea behind Nomos is to express and enforce the contract protocols like the auction via *session types*. The auction session type is defined as

```
type auction =
  /\ <{20} +{running : money -o |{3}> \/ auction,
             ended : +{won : lot * |{5}> \/ auction,
                       lost : money * |{0}> \/ auction}}
```

We first ignore the operators `<{q}|` and `|{q}>` for natural numbers `q` (described later) and describe the remaining type.

The type initiates with `/\` ($\uparrow_{\mathsf{L}}^{\mathsf{S}}$ in abstract syntax) indicating that auction is a *shared* session type [25] that must be acquired by a bidder to interact with the contract. Shared session types guarantee that bidders interact with the auction in *mutual exclusion* and their interaction with the auction executes atomically. Once the action contract is acquired, it replies either with **running** or **ended** indicating the phase of the auction. In the former case, the auction receives money using the `-o` constructor ($\multimap$ in abstract syntax) followed by the bidder releasing the contract matching the `\/` ($\downarrow_{\mathsf{L}}^{\mathsf{S}}$ in abstract syntax, dual to $\uparrow_{\mathsf{L}}^{\mathsf{S}}$) constructor in the type. In the latter case, the auction determines if the bidder won or lost the election. If the bidder wins, the auction sends the **won** label followed by sending the lot using the `*` constructor ($\otimes$ in abstract syntax, dual to $\multimap$). If the bidder loses, the auction sends the **lost** label followed by returning the bidder's money back. In either case, the type recurses back to auction after a `\/` indicating that the bidder must release the auction.

The second key feature of Nomos is that it statically enforces that assets are never duplicated nor discarded, but only transferred between processes. Nomos' type system relies on session types [39] that are rooted in linear logic [69]. This linear type system tracks the assets stored in a process. For instance, the auction contract treats money and lot as linear assets, which is witnessed by the use of the linear logic operators $\multimap$ and $\otimes$ for their exchange.

Finally, an important aspect of smart contracts is their *execution cost*. Blockchains such as Ethereum [145] charge users a fee proportional to the execution (aka gas) cost of their transaction. A unique feature of Nomos is that it uses resource-aware session types [54] to statically analyze the execution cost of a transaction. 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 using the operators `<{q}|` and `|{q}>` ($\triangleleft^q$ and $\triangleright^q$ in abstract syntax). The `<{q}|` operator prescribes that the client must send q potential to the contract, with the amount of potential indicated as a superscript. Dually, `|{q}>` prescribes that the contract must send q potential to the client. In case of the auction contract, we 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 an auction session is 20 (taking the maximum execution cost all branches). Thus, we require the client to send 20 units of potential at the start of a session using `<{20}|`. In the **won** branch of the auction type, on the other hand, the contract returns 5 units of potential to the client using `|{q}>`. This mirrors gas usage in smart contracts, where the sender initiates a transaction with some initial gas, and the leftover gas at the end of the transaction is returned to the sender. All the above potential annotations have been automatically inferred by the Nomos type checker that internally relies on an LP solver to compute gas bounds.

Process Implementations As a final set of illustrations, we describe the main parts of the auction contract program. Since the auction operates in two phases, we have two main processes: `running_auction` for the running phase, and `ended_auction` for the ended phase. The type and definition of `running_auction` process is presented below.

```
contract running_auction :
($bm : Map<address, money>), ($l : lot) |- (#a : auction) =
    $la <- accept #a ;                    % accept acquire request
    get $la {20} ;                        % get 20 potential units
    $la.running ;                          % send 'running' label
    $m <- recv $la ;                       % receive money from bidder
    pay $la {3} ;                          % pay leftover potential
    #a <- detach $la ;                     % detach from bidder
    let addr = Nomos.GetTxnSender() ;      % get bidder's address
    $bm.insert(addr, $m) ;                 % insert bid into bidmap
    #a <- run_or_end $bm $l                % call 'run_or_end' process
```

The process uses two linear channels: `$bm` represents the mapping from `address` to `money`, and `$l` represents the `lot`. On the other hand, it offers the shared channel `#a` that connects the auction contract to the bidder process. To syntactically distinguish session-typed channels from functional variables, Nomos prefixes linear channels with $ (e.g. `$bm`) and shared channels with # (e.g. `#a`). The process closely follows the session-typed protocol described by the `auction`. It first accepts the acquire request from the bidder followed by receiving 20 potential units. Since this process represents the running phase of auction, it sends the **running** label, receives the money from the bidder in `$m`, pays the leftover 3 potential units, and detaches from the bidder. Internally, the process then computes the sender's address using the built-in `GetTxnSender` function, and inserts key-value pair (`addr`, `$m`) into the `$bm` map. Maps have a built-in session type and can be used as such, but we simplify programming by introducing syntactic sugar construct `$bm.insert(addr, $m)`. Finally, the process calls the `run_or_end` process which decides whether to call `running_auction` or `ended_auction` (maybe if the number of bidders reaches a certain threshold).

The bidder process lies on the other end of the auction channel and is responsible for bidding in the auction. We outline the process definition.

```
transaction bid_proc : ($m : money), (#a : auction) |- ($d : 1) =
  $la <- acquire #a ;                 % acquire auction contract
  pay $la {20} ;                      % pay 20 potential units
  case $la (                          % branch on label received
    running => send $la $m ;          % send money to contract
               get $la {3} ;          % get leftover potential
               #a <- release $la ;    % release the auction contract
               close $d               % terminate the transaction
  | ended => abort )                  % abort if auction has ended
```

The process uses linear channel `$m` representing the bid, and the shared channel `#a` that connects to the auction contract. It offers on the channel `$d` of type **1**. We mandate all transaction processes to offer type **1** for simplicity (more details in Section 7.3). The process initiates with acquiring the auction contract, pays 20 potential units, and case analyzes on the response. If the response is **running** (indicating that the auction is running), the process sends the money, gets the leftover potential, releases the contract, and terminates the transaction. If the response is **ended**, we simply abort the transaction. Note how the `running_auction` and `bid_proc` processes perform matching dual actions on the auction channel, as governed by the `auction` session type.

7.2 Mode Inference

Combining all these features in a single language is challenging. To achieve this integration without compromising type safety, Nomos introduces *channel modes*: R for linear asset channels; S for shared channels; L for shared channels in linear phase when acquired; and T for transaction channels. Nomos assigns a mode to *every* channel. For instance, in the `bid_proc` process, `$m` has mode R and `#a` has mode S. Once acquired, `$la` has mode L, and `$d` has mode T.

Practically however, annotating every channel with a mode can be a burden for the programmer. To address this, Nomos automatically infers the mode of every channel automatically. Intuitively, we require the programmer to annotate the process with a *role*. We employ three roles: *asset* for linear assets, *contract* (e.g. `running_auction`), and *transaction* (e.g. `bid_proc`). Nomos then uses these process roles, written before the process name, to assign modes to all channels in that process.

First, based on the mode m of the channel offered by a process, Nomos asserts mode invariants on the shared and linear channels that the process uses. Definition 6.1 details those invariants.

Definition 7.1 (Process Typing)**.** Given judgment $\Psi \ ; \ \Delta \vdash^q P :: (x_m : A)$, and an arbitrary channel $y_k \in \Delta$,

1. If $m = \mathsf{R}$, then $k = \mathsf{R}$.
2. If $m = \mathsf{S}$, then $k = \mathsf{R} \vee k = \mathsf{S}$.
3. If $m = \mathsf{L}$, then $k = \mathsf{R} \vee k = \mathsf{S} \vee k = \mathsf{L}$.
4. If $m = \mathsf{T}$, then $k = \mathsf{R} \vee k = \mathsf{S} \vee k = \mathsf{L} \vee k = \mathsf{T}$.

Intuitively, the above invariants impose a modal hierarchy $\mathsf{R} < \mathsf{S} < \mathsf{L} < \mathsf{T}$ and enforce that a process at mode m only uses channels at mode m' if $m' \leq m$. This hierarchy prevents cycles in the process dependency tree at runtime and is crucial to proving type safety [57].

Relevant to the tool implementation, the above invariants are central to inferring the channel modes automatically. First, Nomos uses the process roles to infer the mode m of the offered channel. We use the following rule:

- For role *asset*: $m = \mathsf{R}$.
- For role *contract*: $m = \mathsf{S}$.
- For role *contract*: $m = \mathsf{T}$.
- (we do not allow defining processes at mode L)

Once we know the offered mode, we use the invariants from Definition 6.1 to generate constraints for mode k of each channel used by a process. In addition, we have three additional constraints.

- The process expression `$lc <- acquire #sc` imposes that the mode of linear channel `$lc` must be L, and the mode of shared channel `#sc` must be S.
- Dually, the process expression `#sc <- release $lc` imposes that mode of `$lc` must be L, and the mode of shared channel `#sc` must be S.
- Finally, if a channel c has a shared type, we conclude that the mode of c must be S.

The Nomos implementation generates and collects these constraints, and ships them to the LP solver. The solver, in turn, solves the constraints, computes the mode of each channel, and substitutes them back into the program. Section 7.3 provides more details on the LP solver.

7.3 Implementation

We have developed an open-source Nomos implementation [10] in OCaml (8469 lines of code). The implementation contains a lexer and parser (594 lines), a type checker (3486 lines), a pretty printer (531 lines), a cost inference engine with an LP solver interface (969 lines of code), and an interpreter (1942 lines). The program is first implemented in the concrete Nomos syntax and then parsed into an abstract syntax tree. The program is then instrumented with work constructs to realize the cost model. Finally, the program is type checked to verify whether it implements the interface described by its session type. In this type checking phase, we also generate LP constraints on the potential annotations, and ship these constraints to the LP solver. The solver then minimizes the total potential while solving these constraints, computes a satisfying assignment, which is substituted back into the program, thus computing the gas cost of the program. Finally, the interpreter runs the program against the current blockchain state to obtain the new blockchain state. We follow a brief description of each aspect of the implementation.

Lexing and Parsing The Nomos lexer and parser have been implemented in Menhir [119], an LR(1) parser generator for OCaml. A Nomos program is a list of mutually recursive type and process definitions. The syntax for definitions is

```
type v = A
<role> f : (x1 : T), (#c2 : A2), ... |{q}- ($c : A) = M
```

The first line describes a type definition: `A` is the type expression that stands for the definition of type name `v` (e.g. `auction` type in Section 7.1). Since Nomos treats types equi-recursively,

Abstract Types	**Concrete Types**	**Abstract Syntax**	**Concrete Syntax**
$\oplus\{l : A, \ldots\}$	`+{l : A, ...}`	$\$x.k$	`$x.k`
$\&\{l : A, \ldots\}$	`&{l : A, ...}`	$\mathsf{case}\ \$x\ (\ell \Rightarrow P)_{\ell \in L}$	`case $x (l => P | ...)`
$A \otimes B$	`A * B`	$\mathsf{send}\ \$x\ \$w$	`send $x $w`
$A \multimap B$	`A -o B`	$\$y \leftarrow \mathsf{recv}\ \$x$	`$y <- recv $x`
$\mathbf{1}$	`1`	$\mathsf{close}\ \$x$	`close $x`
		$\mathsf{wait}\ \$x$	`wait $x`
$\uparrow_{\mathsf{L}}^{\mathsf{S}} A$	`/\ A`	$\$y \leftarrow \mathsf{accept}\ x$	`$y <- accept #x`
		$\$y \leftarrow \mathsf{acquire}\ x$	`$y <- acquire #x`
$\downarrow_{\mathsf{L}}^{\mathsf{S}} A$	`\/ A`	$\#y \leftarrow \mathsf{detach}\ \$x$	`#y <- detach $x`
		$\#y \leftarrow \mathsf{release}\ \$x$	`#y <- release $x`
$t \times A$	`t ^ A`	$\mathsf{send}\ \$x\ M$	`send $x M`
$t \rightarrow A$	`t -> A`	$v \leftarrow \mathsf{recv}\ \$x$	`v = recv $x`
$\triangleright^r A$	`|{r}> A`	$\mathsf{pay}\ \$x\ \{r\}$	`pay $x {r}`
$\triangleleft^r A$	`<{r}| A`	$\mathsf{get}\ \$x\ \{r\}$	`get $x {r}`

TABLE 7.1: Abstract and Corresponding Concrete Syntax for Nomos Types and Expressions

we can silently replace type name `v` with its definition `A`. The second line describes a process declaration and definition. We write the process role, followed by its name, its context and offered channel and type. The process role assigns a mode to the offered channel: *asset*, *contract* or *transaction*, assigning respective modes R, S and T to the offered channel. The modes for all other channels are inferred automatically (explained in Section 7.2). The context contains both functional variables and session-typed channel variables: `x1 : T` defines a functional variable `x1` of type `T`; `#c2 : A2` defines a channel `#c2` with type `A2`; the process offers channel `$c` with type `A`. The expression `M` stands for the process definition. To visually separate out functional variables from session-typed channels, we require that shared channels are prefixed by $\#$, while linear channels are prefixed by $\$$. This avoids confusion between the two, both for the programmer and the parser. Finally, the potential `{q}` of a process is marked on the turnstile in the declaration.

As a reference, Table 7.1 provides the abstract and concrete syntax of the session types and their corresponding process constructs in Nomos.

Cost Instrumentation Once a program has been parsed and converted into an abstract syntax tree, we instrument it with work constructs based on the cost model. The cost model intuitively defines the execution cost of each construct. The instrumentation engine takes the program and the cost model as input and produces a program with work constructs inserted at appropriate places. We use the following rule ($[\![P]\!]$ denotes the work-instrumented version of process P):

$$[\![S\ ;\ P]\!] ::= \mathsf{work}\ \{C_S\}\ ;\ S\ ;\ [\![P]\!]$$

Here, S is a process construct with P as its continuation. and C_S denotes the cost of construct S according to the given cost model. We use this rule to add work annotations throughout the program, thus realizing the execution cost of the program. This instrumentation simplifies the cost analysis which can simply assign a cost of c to work $\{c\}$ and cost 0 to all other process expressions.

Type Checking The Nomos type checker is based on bi-directional type checking [117]. Intuitively, the programmer provides the initial type of each variable and channel in the declaration and the definition is checked against it, while reconstructing the intermediate types. This helps localize the source of a type error as the point where type reconstruction fails. As an illustration, recall the implementation of the running phase of the auction.

```
type auction =
  /\ <{20} +{running : money -o |{3}> \/ auction,
             ended : +{won : lot * |{5}> \/ auction,
                       lost : money * |{0}> \/ auction}}

contract running_auction :
($bm : Map<address, money>), ($l : lot) |{0}- (#a : auction) =
       % ($bm : Map<address, money>), ($l : lot)
           |- (#a : auction)
     $la <- accept #a ;
       % ($bm : Map<address, money>), ($l : lot)
           |- ($la : <{20}| +{running : ..., ended : ...})
     get $la {20} ;
       % ($bm : Map<address, money>), ($l : lot)
           |- ($la : +{running : ..., ended : ...})
     $la.running ;
       % ($bm : Map<address, money>), ($l : lot)
           |- ($la : money -o |{3}> \/ auction)
     $m <- recv $la ;
       % ($bm : Map<address, money>), ($l : lot), ($m : money)
           |- ($la : |{3}> \/ auction)
     pay $la {3} ;
       % ($bm : Map<address, money>), ($l : lot), ($m : money)
           |- ($la : \/ auction)
     #a <- detach $la ;
       % ($bm : Map<address, money>), ($l : lot), ($m : money)
           |- (#a : auction)
     let addr = Nomos.GetTxnSender() ;
       % (addr : address), ($bm : Map<address, money>), ($l : lot)
           ($m : money) |- (#a : auction)
     $bm.insert(addr, $m) ;
       % (addr : address), ($bm : Map<address, money>), ($l : lot)
           |- (#a : auction)
     #a <- run_or_end $bm $l
```

If the programmer forgets to write `$bm.insert(addr, $m)`, reconstruction would fail. There would be an extra `($m : money)` left in the process context, and since Nomos employs a linear type system, the type checker would report an error. In effect, the type checker *forces the programmer to consume* the channel `$m` to ensure linearity.

Type equality is restricted to reflexivity (constant time), although we have also implemented the standard co-inductive algorithm [66] which is quadratic in the size of type definitions. For all our examples, the reflexive notion of equality was sufficient. If type equality is restricted to constant-time reflexive notion, *type checking is linear time in the size of the program.* This property is quite relevant in the blockchain setting, since type checking can be part of the attack surface. If type checking is too slow, malicious users can issue transaction programs that take too long to type check, effectively forcing a denial-of-service attack.

LP Solver for Potential and Mode Inference The potential and mode annotations are the most interesting aspects of the Nomos type system. Since modes are associated with each channel, they are tedious to write. Similarly, the exact potential annotations depend on the cost assigned to each operation and is difficult to predict statically. Thus, we implemented an automatic inference algorithm for both these annotations by relying on an off-the-shelf LP solver.

Using ideas from existing techniques for type inference for AARA [83, 84], we reduce the reconstruction of potential annotations to linear optimization. To this end, Nomos' inference engine uses the Coin-Or LP solver. In a Nomos program, the programmer can indicate unknown potential using $*$. Thus, resource-aware session types can be marked with $\triangleright^*$ and $\triangleleft^*$, list types can be marked as $L^*(\tau)$ and process definitions can be marked with $|\{*\}-$ on the turnstile. The mode of all the channels is marked as 'unknown' while parsing.

As an example, consider the auction session type. The programmer writes the following type:

```
type auction =
  /\ <{*} +{running : money -o |{*}> \/ auction,
             ended : +{won : lot * |{*}> \/ auction,
                         lost : money * |{*}> \/ auction}}
```

Using $*$ annotations minimizes the programmer burden who does not need to compute exact potential annotations and execution cost.

The inference engine then iterates over the program and substitutes the star annotations with potential variables and 'unknown' with mode variables. For the auction type, this reduces to

```
type auction =
  /\ <{v0} +{running : money -o |{v1}> \/ auction,
              ended : +{won : lot * |{v2}> \/ auction,
                          lost : money * |{v3}> \/ auction}}
```

In the first phase of type checking, we ignore the potential and mode annotations and approximately type check the remaining program. This phase rules out type errors resulting from structural session types, i.e., protocol or linearity violation. In the second phase, we apply the rules for potential constructors (see Figures 6.3, 6.4) to generate linear constraints on the potential variables. For the above example, we generate the constraints

$$v_0 - v_1 \geq 17 \qquad v_0 - v_2 \geq 15 \qquad v_0 - v_3 \geq 20$$
$$v_0 \geq 0 \qquad v_1 \geq 0 \qquad v_2 \geq 0 \qquad v_3 \geq 0$$
$$\mathsf{min}(v_0 + v_1 + v_2 + v_3)$$

Finally, these constraints are shipped to the LP solver, which minimizes the value of the potential annotations to achieve tight bounds. The LP solver either returns that the constraints are infeasible, or returns a satisfying assignment, which is then substituted into the program. For the above example, we obtain the solution: $v_0 = 20, v_1 = 3, v_2 = 5, v_3 = 0$. The final program is pretty printed for the programmer to view and verify the potential and mode annotations.

7.4 Blockchain Integration

To integrate Nomos with a blockchain, we need a mechanism to *(i)* represent contracts and their addresses in the current blockchain state, *(ii)* create and execute transactions, and *(iii)* construct the global distributed ledger. This section addresses these challenges and also highlights the main limitation of the language: deadlocks in the transaction programs.

Nomos on a Blockchain We assume working in a blockchain setting similar to Ethereum with a standard account model. At any given time, the blockchain contains a set of Nomos contracts: $C_1, \ldots, C_n$ with their type information: $\cdot \; ; \; \Gamma^i \; ; \; \Delta^i_{\mathsf{R}} \vdash^{q_i} C_i :: (x^i_{\mathsf{S}} : A^i_{\mathsf{S}})$. The shared context Γ^i types the shared contracts that C_i refers to, and the linear context Δ^i_{R} types the contract's linear assets. The channel name x^i_{S} of a contract is its address and has to be globally unique. Our implementation contains a deterministic mechanism to generate unique fresh names. We allow contracts to carry potential given by the annotation q_i, which can be used to share and amortize gas cost across transactions. It is also straightforward to modify the blockchain setup to suppress the potential stored in the contracts.

These contracts together form a stuck configuration (a valid virtual blockchain state) typed as

$$\Gamma \overset{E}{\models} \mathsf{proc}(x^1_{\mathsf{S}}, w_1, C_1) \ldots \mathsf{proc}(x^n_{\mathsf{S}}, w_n, C_n) :: (\Gamma \; ; \; \cdot)$$

where $\Gamma = (x^1_{\mathsf{S}} : A^1_{\mathsf{S}}), \ldots, (x^n_{\mathsf{S}} : A^n_{\mathsf{S}})$ and $E = \Sigma^n_{i=1} q_i + w_i$ is the total energy of the configuration, that is, the sum of the stored potential and previously performed work. The actual implementation stores some additional metadata such as mapping from linear channels to their shared counterpart, linear channels to their continuation, and a mapping from shared channels to their types. The mapping from linear channels to their counterpart is necessary for the

$\downarrow_{\mathsf{L}}^{\mathsf{S}} C$ rule where a linear channel is released. To release a linear channel a_{L} to its corresponding shared channel a_{S}, we utilize this mapping that stores $a_{\mathsf{L}} \mapsto a_{\mathsf{S}}$. But since fresh channels are created with every communication, if a shared channel a_{S} is acquired to create a_{L}, then the linear channel to be released to a_{S} might be different from a_{L}, particularly if some communication occurred on a_{L}. The need for mapping shared channels to their types is explained in the paragraph on the Nomos interpreter.

To perform a transaction with a contract, a user submits a transaction script Q (a process) that is well-typed with respect to the existing contracts:

$$\cdot \; ; \; \Gamma \; ; \; \cdot \vdash^{q} Q :: (x_{\mathsf{T}} : \mathbf{1})$$

We mandate that the transaction offers along a channel of type $\mathbf{1}$ and terminates by sending a close message on its offered channel. Intuitively, this enforces that the transaction, at termination, leaves the blockchain in a well-formed state. This transaction process is added to the set of contracts and the new (closed) configuration is typed as

$$\Gamma \overset{E+q}{\vDash} \mathsf{proc}(x_{\mathsf{S}}^{1}, w_1, C_1) \ldots \mathsf{proc}(x_{\mathsf{S}}^{n}, w_n, C_n) \; \mathsf{proc}(x_{\mathsf{T}}, 0, Q) :: (\Gamma \; ; \; (x_{\mathsf{T}} : \mathbf{1}))$$

This configuration then steps according to the Nomos semantics. A transaction can either create new contracts, or update the state of existing contracts. In the former case, new contracts are added to the blockchain state, making them visible in the type of the configuration for subsequent transactions to access. The type safety of Nomos ensures that transaction execution will be successful terminating in the following configuration

$$\Gamma \overset{E'}{\vDash} \mathsf{proc}(x_{\mathsf{S}}^{1}, w_1', C_1') \ldots \mathsf{proc}(x_{\mathsf{S}}^{m}, w_m', C_m') \; \mathsf{msg}(x_{\mathsf{T}}, 0, \mathsf{close}\; x_{\mathsf{T}}) :: (\Gamma \; ; \; \cdot)$$

where m may be greater than n since the transaction can create additional contracts. At this point, we remove the close message from the configuration, resulting in the stuck configuration

$$\Gamma \overset{E'}{\vDash} \mathsf{proc}(x_{\mathsf{S}}^{1}, w_1', C_1') \ldots \mathsf{proc}(x_{\mathsf{S}}^{m}, w_m', C_m') :: (\Gamma \; ; \; \cdot)$$

This stuck configuration represents a valid blockchain state guaranteeing that the transaction execution has successfully terminated, and we may initiate a new transaction on this new blockchain state.

Gas Accounts With this blockchain setup, one may reasonably wonder about the origin of potential. In our integration, we allow users to create *gas accounts* that store potential. To simplify matters, users can only store gas in these accounts; to store other private data and assets, they must create contracts where this additional information is collected. Gas accounts only contain a username and gas balance. We allow users to create new accounts and deposit gas into their accounts. When a user submits a transaction, the gas cost of the transaction is inferred by the LP solver, and this gas is automatically deducted from the account balance of that user. If the account balance is insufficient to pay for the transaction cost, the transaction is simply aborted.

Subsynchronizing Session Types A transaction program usually proceeds by *acquiring* existing contracts, exchanges data and assets with them, and subsequently *releasing* them. In Nomos, we require session types to be *equi-synchronizing* [25], i.e., contracts must be released at the same type that they are acquired. This requirement is crucial to type safety due to the shared nature of contracts. Since multiple users interact with a shared contract, it is critical that they observe a common shared type for the contract. If one client alters this type after interaction, other clients are not notified of this change, which will break type safety.

This equi-synchronizing constraint imposes a strong restriction: contracts must maintain a common type despite the phase they operate in. For instance, the auction contract operates over two phases: an *open* phase where bidders bid into the auction; and a *closed* phase when bidders withdraw their bids from the auction. However, these phases are not reflected in the type. Recall the auction type.

```
type auction =
  /\ <{20} +{running : money -o |{3}> \/ auction,
             ended : +{won : lot * |{5}> \/ auction,
                         lost : money * |{0}> \/ auction}}
```

As we observe, the auction is acquired and released at the common type: auction.

In Nomos, we can relax this restriction by allowing *sub-synchronizing* types. Here, a contract can be released at a subtype of the type it was acquired at. For instance, we can introduce two mutually recursive auction types.

```
type open_auction =
  /\ <{20} +{running : money -o |{3}> \/ open_auction,
             ended : +{won : lot * |{5}> \/ closed_auction,
                         lost : money * |{0}> \/ closed_auction}}

type closed_auction =
  /\ <{20} +{ended : +{won : lot * |{5}> \/ closed_auction,
                         lost : money * |{0}> \/ closed_auction}}
```

The open_auction type describes the open auction phase: it can either stay open (if it sends the running message), or transition to closed (if it sends the ended) message. As exemplified by the type, the phase of the auction is now visible in its type.

This relaxation, however, comes at a cost. Type checking in this relaxed setting needs to reason about subtyping, instead of type equality. Subtyping, unlike type equality, is more general than reflexivity, hence cannot be decided in constant time. Thus, in the worst case, type checking would no longer be linear in this relaxed environment.

Deterministic Execution Since blockchains rely on consensus among the miners, it is important to ensure deterministic execution of transactions. However, Nomos semantics has one source of non-determinism: the *acquire-accept* rule where an accepting contract latches on to any acquiring transaction. The Nomos implementation resolves this non-determinism by mandating that the contract must interact with the transaction with the lowest channel number. This simple heuristic, although not the cleanest, is easy to implement and sufficient for our purposes. Another promising approach is *record-and-replay* [99, 128]. The miner who mines the transaction records the order in which they resolved the acquire-accept non-determinism. All other miners validating the blockchain state must replay the same order, thus obtaining the same blockchain state after execution.

Interpreter The Nomos implementation provides two key functionalities: *inference* and *execution*. The inference engine takes a transaction program as input, infers the potential and mode annotations and outputs a well-typed program which is verified by the type checker. Next, the execution engine takes the well-typed transaction program and a valid blockchain state as input, executes the transaction against the state and outputs a valid blockchain state.

The Nomos interpreter uses OCaml S-expressions to represent blockchain states. The interpreter has read/write functionality which converts blockchain state to S-expression and vice-versa. This helps persist the blockchain state across transactions. The interpreter takes an input file, reads the S-expression from it, converts it to a blockchain state, executes the transaction against the state, and writes the output blockchain state to an output file. Internally, the interpreter is based on the semantics rules presented in Chapter 6.

Deadlocks The only language specific reason a transaction can fail is a deadlock in the transaction code. Our progress theorem accounts for this possibility of deadlocks. We currently employ dynamic deadlock detection techniques internally in the implementation. Intuitively, since a valid blockchain state represents a stuck configuration of a particular form (only shared contracts in the configuration), we verify that the execution terminates with the configuration in this form. If not, we conclude that a deadlock occurred during the execution, and we simply abort the whole transaction. We maintain snapshots of the configuration after every transaction execution, so we simply revert to the previous valid blockchain state. It is the user's responsibility to issue a new transaction that does not deadlock. In the future, we plan to employ deadlock prevention techniques [27] to statically rule out deadlocks.

7.4.1 Blockchain-specific Features

To further simplify programming, we enhance Nomos with blockchain-specific features. This section provides a brief overview of these features.

Map Data Structure The most widely used data structure in smart contracts is maps. It is often used in contracts to store a mapping from users to their balance. The auction example uses it to map users to their bids. We provide surface syntax to make it easier for programmers to interact with maps.

The first step is to dinstinguish between linear and non-linear maps. Although the two maps differ in their statics and semantics, we want to provide a unified syntax for ease of programming. We first describe the session type for a non-linear map with key type `kt` and value type `vt`.

```
type Map<kt, vt> = &{insert : kt -> vt -> Map<kt, vt>,
                     delete : kt -> vt option ^ Map<kt, vt>,
                     size : int ^ Map<kt, vt>,
                     close : 1}
```

The map is implemented with a recursive session type initiating with an external choice. It accepts one of three messages: `insert`, `delete`, `size` or `close`. In the case of `insert`, the map receives a key and value and inserts the pair in the dictionary. If the key already exists, the existing value is overwritten. In the case of `delete`, the map receives a key and returns an optional value depending on whether the key exists in the map or not. In the case of `size`, the map returns an integer corresponding to its size. In each of these three cases, the type then recurses back to `Map<kt, vt>`. Finally, in the case of `close`, the map simply terminates with a `close` message.

The case of a linear map is only slightly different. The type is as follows:

```
type Map<kt, vt> = &{insert : kt -> vt -o Map<kt, vt>,
                     delete : kt -> vt option * Map<kt, vt>,
                     size : int ^ Map<kt, vt>,
                     close : +{empty : 1,
                               nonempty: Map<kt, vt>}}
```

The type differs from a non-linear map in a few key ways. First, `vt` is a linear type, hence we use `-o` and `*` constructors to exchange them. Second, closing a linear map is only allowed when it is empty. Hence, on receiving a `close` message, a linear map will respond with either the `empty` message followed by termination. Or with the `nonempty` message followed by recursing back to its original type.

Finally, we provide surface syntax to ease programming with maps.

- `$m <- new Map<kt, vt>`: for creating a new map `$m` of key type `kt` and value type `vt`
- `$m.insert(k, v)`: for inserting key `k` and value 'v' into map `$m`. If the map is linear, the value is replaced by a channel `$v`.
- `v = $m.delete(k)`: for deleting key `k` from map `$m`. If the map is linear, the expression changes to `$v <- $m.delete(k)`.

- `n = $m.size`: for obtaining the size of map `$m` and storing it in variable `n`.
- `$m.close`: for closing the map.

Blockchain-specific Expressions To keep track of blockchain users, we introduce a built-in type called `address`. We use the expression `Nomos.GetTxnSender()` as an introduction form that returns a value of type `address`. There are no elimination forms for this type.

We also introduce the expression `Nomos.GetTxnNum()` to track the current transaction number. Our implementation uses a sequential mode of execution: every new transaction is automatically assigned the number $n + 1$ where n is the number of the previous transaction. The first transaction is assigned the number 0.

Exact Gas Computation So far, we have only discussed upper bound gas computation by the Nomos inference engine. However, a central limitation of upper gas bounds is that we still need to monitor gas cost at runtime [53] to return the leftover gas back to the user at the end of execution. And this dynamic gas monitoring can create significant runtime overheads, which can increase transaction fees and reduce the total throughput of the system.

Forunately, a minor tweak to the language can lead to completely eliminating dynamic gas monitoring. Upper gas bounds are primarily caused by different gas costs along different program branches. To mitigate this issue, we introduce a novel expression `Nomos.deposit {r}` which deposits r units of gas in the sender's account. The less costly branch is automatically augmented with the expression `Nomos.deposit {*}`. The program is then shipped to the LP solver which computes the value of all such $*$ annotations. And at runtime, these expressions safely return the leftover gas in the sender's account. There is no need to dynamically monitor gas cost at runtime!

Custom Coins Finally, Nomos provides a built-in abstract `coin` type. There are no introduction or elimination forms of this type. Thus, there are no ways to delete or duplicate a channel of this type; they can only be transferred across different contracts. This type can be used to truly enforce linearity of assets on the blockchain.

These coins can further be used to encode fancier tokens. For example, we can create a type `coin2` to represent 2 coins.

```
type coin2 = coin * coin
```

We can even store more information inside the coin, e.g. the owner's address or the history of all owners in the past. For instance, consider the UTxO coin.

```
type utxo = &{history : address list ^ utxo,
              delete : coin}
```

This type, when queried, can return the address list of all its owners. But we can also delete its history, and turn it into a simple coin. The Nomos language is thus, very general, and we are not tied to a particular system-specific coin.

7.5 Evaluation

We evaluate the design of Nomos by implementing several smart contract applications and discussing the typical issues that arise. All the contracts are implemented and type checked in the prototype implementation and the potential and mode annotations are derived automatically by the inference engine. The cost model used for these examples assigns 1 unit of cost to every atomic internal computation and sending of a message. We show the contract types from the implementation with the following ASCII format: i) `/\` for $\uparrow_{\mathsf{L}}^{\mathsf{S}}$, ii) `\/` for $\downarrow_{\mathsf{L}}^{\mathsf{S}}$, iii) `<{q}|` for $\triangleleft^{q}$, iv) `|{q}>` for $\triangleright^{q}$, v) `^` for $\times$, vi) `*[m]` for $\otimes_m$, vii) `-o[m]` for $\multimap_m$.

ERC-20 Token Standard ERC-20 [4] is a technical standard for smart contracts on the Ethereum blockchain that defines a common list of standard functions that a token contract has to implement. The majority of tokens on the Ethereum blockchain are ERC-20 compliant.

The ERC-20 token contract implements the following session type in Nomos:

```
stype erc20token = /\ <{11}| &{
  totalSupply : int ^ |{9}> \/ erc20token,
  balanceOf : id -> int ^ |{8}> \/ erc20token,
  transfer : id -> id -> int -> |{0}> \/ erc20token,
  approve : id -> id -> int -> |{6}> \/ erc20token,
  allowance : id -> id -> int ^ |{6}> \/ erc20token }
```

The type ensures that the token implements the protocol underlying the ERC-20 standard. To query the total number of tokens in supply, a client sends the totalSupply label, and the contract sends back an integer. If the contract receives the balanceOf label followed by the owner's identifier, it sends back an integer corresponding to the owner's balance. A balance transfer can be initiated by sending the transfer label to the contract followed by sender's and receiver's identifier, and the amount to be transferred. If the contract receives approve, it receives the two identifiers and the value, and updates the allowance internally. Finally, this allowance can be checked by issuing the allowance label, and sending the owner's and spender's identifier.

The design of Nomos is orthogonal to the concrete representation of money or currency in the language. The Nomos implementation provides a simple built-in abstract coin type of a unit value. Our implementation of the erc20token session type relies on these abstract coins used exclusively for exchanges among the private accounts. Coins are treated linearly as no operations are allowed on primitive types. As a result, coins cannot be created or discarded.

It is straightforward to add features by using more sophisticated abstract coin types or by providing built-in operations that are executed by the runtime system. For example, we can add coins with unique identifiers or coins of different denominations by changing the underlying session type of coins. Similarly, we can add operations for minting (creating) or burning (discarding) coins if users have the respective privileges. Such operations could be, for instance, implemented in an abstract contract that is an interface to the runtime system. Finally, there can be operations for exchanging coins and gas at rates that are fixed when type-checking transactions.

It is also possible to allow programmers to define their own abstract types with their individual introduction and elimination forms to use them in an implementation of a session type like erc20token.

Hacker Gold (HKG) Token The HKG token is one particular implementation of the ERC-20 token specification. Recently, a vulnerability was discovered in the HKG token smart contract based on a typographical error leading to a re-issuance of the entire token [2]. When updating the receiver's balance during a transfer, instead of writing `balance+=value`, the programmer mistakenly wrote `balance=+value` (semantically meaning `balance=value`). Nomos' type system would have caught the linearity violation in the latter statement that drops the existing balance in the recipient's account.

Puzzle Contract This contract, taken from prior work [104] rewards users who solve a computational puzzle and submit the solution. The contract allows two functions, one that allows the owner to update the reward, and the other that allows a user to submit their solution and collect the reward.

In Nomos, this contract is implemented to offer the type

```
stype puzzle = /\ <{14}| &{
  update : id -> money -o[R] |{0}> \/ puzzle,
  submit : int ^ &{
    success : int -> money *[R] |{5}> \/ puzzle,
    failure : |{9}> \/ puzzle } }
```

The contract still supports the two transactions. To update the reward, it receives the update label and an identifier, verifies that the sender is the owner, receives money from the sender, and acts like a puzzle again. The transaction to submit a solution has a *guard* associated with it. First, the contract sends an integer corresponding to the reward amount, the user then verifies that the reward matches the expected reward (the guard condition). If this check succeeds, the user sends the success label, followed by the solution, receives the winnings, and the session terminates. If the guard fails, the user issues the failure label and immediately terminates the session. Thus, the contract implementation guarantees that the user submitting the solution receives their expected winnings.

Voting The voting contract provides a ballot type.

```
stype ballot = /\ <{16}| +{
  open : id -> +{ vote : id -> |{0}> \/ ballot,
                    novote : |{9}> \/ ballot },
  closed : id ^ |{13}> \/ ballot }
```

This contract allows voting when the election is **open** by sending the candidate's *id*, and prevents double voting by checking if the voter has already voted (the **novote** label). Once the election closes, the contract can be acquired to check the winner. We use two implementations for the contract: the first stores a counter for each candidate that is updated after each vote is cast (voting in Table 7.2); the second does not use a counter but stores potential inside the vote list that is consumed for counting the votes at the end (voting-aa in Table 7.2). This stored potential is provided by the voter to amortize the cost of counting. The type above shows the potential annotations corresponding to the latter.

Insurance Nomos has been carefully designed to allow inter-contract communication without compromising type safety. We illustrate this feature using an insurance contract that processes flight delay insurance claims after verifying them with a trusted third party. The insurer and third party verifier are implemented as separate contracts providing the following session types.

```
stype insurer = /\ <{6}| &{
  submit : claim -> +{
    success : money *[R] |{0}> \/ insurer,
    failure : |{1}> \/ insurer } }

stype verifier = /\ <{3}| &{
  verify : claim -> +{
    valid : |{0}> \/ verifier,
    invalid : |{0}> \/ verifier } }
```

The insurer type provides the option to **submit** a claim by receiving it and responds with **success** or **failure** depending upon verification of the claim. If the claim is successful, the insurer sends over the reimbursement in the form of money. The verifier type provides the option to **verify** a claim by receiving it and responding with **valid** or **invalid** depending on the validity of the claim.

The insurer, upon receiving a claim, acquires the verifier and sends it the claim details. If the claim is valid, then it responds with **success**, sends the money and detaches from its client. If the claim is invalid, it responds with **failure** and immediately detaches from its client.

Contract	LOC	T (ms)	Vars	Cons	I (ms)	Gap
auction	176	0.558	229	730	5.225	3
ERC 20	136	0.579	161	561	4.317	6
puzzle	108	0.410	126	389	8.994	8
voting	101	0.324	109	351	3.664	0
voting-aa	101	0.346	140	457	3.926	0
escrow	85	0.404	95	321	3.816	3
insurance	56	0.299	76	224	8.289	0
bank	147	0.663	173	561	4.549	0
wallet	30	0.231	32	102	3.224	0

Table 7.2: Evaluation of Nomos with Case Studies. LOC = lines of code; T (ms) = the type checking time in ms; Vars = #variables generated during type inference; Cons = #constraints generated during type inference; I (ms) = type inference time in ms; Gap = maximal gas bound gap.

Experimental Evaluation We describe the 8 case studies we implemented in Nomos. We have already discussed auction (Section 7.1), ERC 20, puzzle, voting, and insurance. The other case studies are:

- A bank account that allows users to register, make deposits and withdrawals and check the balance.
- An escrow to exchange bonds between two parties.
- A wallet allowing users to store money on the blockchain.

Table 7.2 contains a compilation of our experiments with the case studies and the prototype implementation. The experiments were run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory. It presents the contract name, its lines of code (LOC), the type checking time (T (ms)), number of potential and mode variables introduced (Vars), number of potential and mode constraints that were generated while type checking (Cons) and the time the LP solver took to infer their values (I (ms)). The last column describes the maximal gap between the static gas bound inferred and the actual runtime gas cost. It accounts for the difference in the gas cost in different program paths. However, this waste is clearly marked in the program by explicit *tick* instructions so the programmer is aware of this runtime gap, based on the program path executed.

The evaluation shows that the type-checking overhead is less than a millisecond for case studies. This indicates that Nomos is applicable to settings like distributed blockchains in which type checking could add significant overhead and could be part of the attack surface. Type inference is also efficient but an order of magnitude slower than type checking. This is acceptable since inference is only performed once during deployment and can be carried out off-chain. Gas bounds are tight in most cases. Loose gas bounds are caused by conditional branches with different gas cost. In practice, this is not a major concern since the Nomos semantics tracks the exact gas cost, and a user will not be overcharged for their transaction. However, Nomos' type system can be easily modified to only allow contracts with tight bounds.

Our implementation experience revealed that describing the session type of a contract crystallizes the important aspects of its protocol. Often, subtle aspects of a contract are revealed while defining the protocol as a session type. Once the type is defined, the implementation simply *follows* the type protocol. The error messages from the type checker were helpful in ensuring linearity of assets and adherence to the protocol. Using $*$ for potential annotations meant we could remain unaware of the exact gas cost of operations. The syntactic sugar constructs reduced the programming overhead and the size of the contract implementations.

Chapter 8

Conclusion

This chapter concludes my thesis with a summary of contributions, a brief look at possible future directions, and some final thoughts.

8.1 Summary of Contributions

This thesis makes two major contributions: design and type-thoeretic foundation of resource-aware session types, and application of resource-aware session types for a safe smart contract programming language called Nomos.

Chapter 3 begins with introducing novel arithmetically refined session types. The key innovation here was to index types with natural numbers that represent data structure sizes and values. We also allow session-typed processes to exchange linear arithmetic constraints on these indices to constrain process behavior. Our first, rather surprising, theoretical result was that type equality, and therefore, type checking is undecidable for refinement session types, even though Presburger arithmetic itself is decidable. But as a recourse, we devised an algorithm for approximating type equality which, despite its incompleteness, works very well in practice. Since refinement constructs introduce verbosity overhead to programs, we also devised a novel forcing algorithm to insert refinement constructs automatically. This assists with code reuse by greatly reducing programmer burden. Finally, the language including all the above algorithms are implemented in a type-safe system called Rast which is evaluated on standard session-typed benchmarks.

Chapter 4 crucially employs refinements to express work bounds for session-typed processes. To compute work, our key innovation was to introduce an abstract notion of potential. This potential can be stored inside processes and exchanged using special messages but, most importantly, it must be consumed to take an execution step. Intuitively, potential gets converted into work at runtime. This implies that the initial potential stands as an upper bound on the work bound of a concurrent computation. This is exactly what is formalized by the soundness theorem. As an application, we compare the efficiency of standard stack and queue implementations.

Chapter 5 employs refinements to express time bounds for programs. The key innovation here was to augment the timing information of message exchanges in the session type. In particular, we introduce the $\bigcirc^r$ operator that stands for a delay of r units of time. For instance, a channel of type $\bigcirc^r A$ behaves as A after exactly r time units. To capture unknown timings, we also introduce $\Box$ and $\Diamond$ operators which describe an eventual message exchange. We combine these three operators to flexibly express time bounds on standard concurrent data structures.

Session types, together with work and span extensions, are dubbed as *resource-aware session types*.

Chapter 6 is dedicated to the type-theoretic foundation of the Nomos language that provides 3 domain-specific features. First, the communication protocols that are statically prescribed by session types are instrumental in expressing and enforcing transaction behavior. Second, the linear characteristic of session types enables programmers to track assets preventing their accidental duplication or deletion. Finally, resource-aware session types automatically infer the execution cost of transactions relieving programmers of this burden too. In addition, the chapter also describes how session types are integrated into a functional language via a linear contextual monad. The chapter concludes with the usual progress and preservation theorems.

Chapter 7 complements the previous chapter by describing the implementation of the Nomos language. The aim of this chapter is to demonstrate the practicality and wide applicability of Nomos. More concretely, the chapter describes our efforts in reducing programmer burden. First, verbose channel modes are automatically inferred using an LP solver. Second, the implementation provides several blockchain-specific features such as easy use of map data structures, transaction-related expressions, and gas accounts to send transactions. Third, the chapter also describes how Nomos integrates into an account model blockchain. Finally, Nomos is evaluated on a variety of standard smart contracts with a detailed description of the guarantees it can provide.

8.2 Future Directions

I conclude my thesis with a few broad future directions that closely align with resource-aware session types and the Nomos language. I have classified these directions in 3 broad projects.

(i) design of optimal scheduling policies based on execution cost

(ii) automatic synthesis of distributed programs from their specification

(iii) implementation and analysis of cryptographic systems

Cost Analysis for Optimal Scheduling Cost analysis can assist developers in designing optimal scheduling policies for their applications. The sequential complexity bounds from Rast can be used to determine whether a new computation needs to be executed in the current or a freshly spawned thread. The parallel complexity bounds from Rast implicitly determine data dependency between threads and can be used to decide the order of thread execution. We can implement complexity-driven scheduling policies in Rast and evaluate their impact on performance.

Synthesis of Distributed Programs One effective way of assisting developers is by writing programs for them! Refinement session types can naturally express program specifications. Programs can then be synthesized from their specifications by applying data-driven deep learning techniques. Refinements can also be utilized for semi-automatic synthesis of smart contracts. I believe refinement session types can propel program synthesis to distributed systems.

Rast for Cryptographic Protocols In a recent collaboration, I observed that refinement session types can neatly represent security protocols. With the recent growth in the complexity of such protocols, developers can greatly benefit from language support while building secure systems. We can also employ type-based techniques to model ill-behaved adversaries and formally verify cryptographic protocols. Furthermore, resource-aware types can provide computational security by modeling adversaries who are capable of only polynomial-time computation. We can also use programming language techniques to demonstrate *universal composability*, which entails that security properties of cryptographic protocols are preserved even when arbitrarily composed with other protocols.

More broadly, session types carry the safety guarantees that type systems can provide to the distributed domain. Resource-aware session types augment session types with cost information to provide computational guarantees along with safety. Together, they can inspire the design of next-generation concurrent programming languages. On the practical side, blockchains hold the potential of providing financial infrastructure access to underprivileged communities. Safe smart contract languages are the first step towards increase our trust in financial systems and improving their broader applicability. Nomos can inspire the design of future smart contract languages. In conclusion, programming language tools and techniques hold the power of improving software design and development and making technologies safer, faster, and more reliable!

Bibliography

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

[2] Ether.camp's hkg token has a bug and needs to be reissued. https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued, January 2017. Accessed: 2019-02-25.

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

[4] Erc20 token standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard, december 2018. Accessed: 2018-02-027.

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

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

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

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

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

[10] Nomos implementation. https://github.com/ankushdas/Nomos, 2019. Accessed: 2019-11-11.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Deciding the bisimilarity of context-free session types. In A. Biere and D. Parker, editors, *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, pages 39–56, Dublin, Ireland, April 2020. Springer LNCS 12079.

[20] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992.

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

[22] 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.

[23] 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.

[24] Pedro Baltazar, Dimitris Mostrous, and Vasco T. Vasconcelos. Linearly refined session types. In S. Alves and I. Mackie, editors, *International Workshop on Linearity (LINEARITY 2012)*, pages 38–49, Tallinn, Estonia, April 2012. EPTCS 101. doi: 10.4204/eptcs.101.4.

[25] 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.

[26] 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.

[27] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In Luís Caires, editor, *Programming Languages and Systems*, pages 611–639, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17184-1.

[28] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-36750-5.

[29] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. URL https://hal.inria.fr/inria-00069968. Projet COQ.

[30] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sabastian Podda, and Livio Pompianu. A contract-oriented middleware. In C. Braga and P. Ölveczky, editors, *Formal Aspects of Component Software (FACS 2015)*, pages 86–104. Springer LNCS 9539, 2015.

[31] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia Allessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In S. Graf and M. Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2015)*, pages 161–177, Grenoble, France, June 2015. Springer LNCS 9039.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

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

[40] 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.

[41] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 459–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-20398-5.

[42] 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.

[43] 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.

[44] 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.

[45] 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).

[46] Ruofei Chen and Stephanie Balzer. Ferrite: A judgmental embedding of session types in Rust. *CoRR*, abs/2009.13619, 2020. URL `http://arxiv.org/abs/2009.13619`.

[47] 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.

[48] David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7(91-99):300, 1972.

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

[50] 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`.

[51] Ankush Das and Frank Pfenning. Session types with arithmetic refinements. In I. Konnov and L. Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171, pages 13:1–13:18. LIPIcs 171, 2020. doi: 10.4230/LIPIcs.CONCUR.2020.13. URL `https://drops.dagstuhl.de/opus/volltexte/2020/12825`.

[52] Ankush Das and Frank Pfenning. Verified linear session-typed concurrent programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming (PPDP 2020)*, pages 7:1–7:15, Bologna, Italy, September 2020. ACM.

[53] Ankush Das and Shaz Qadeer. Exact and linear-time gas-cost analysis. In David Pichardie and Mihaela Sighireanu, editors, *Static Analysis*, pages 333–356, Cham, 2020. Springer International Publishing. ISBN 978-3-030-65474-0.

[54] 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.

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

[56] Ankush Das, Farzaneh Derakhshan, and Frank Pfenning. Rast Implementation. `https://bitbucket.org/fpfenning/rast/src/master/`, 2019. Accessed: 2019-11-11.

[57] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-Aware Session Types for Digital Contracts. In *34th IEEE Computer Security Foundations Symposium, CSF*, 2021. To appear.

[58] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Nested Session Types. In *30th European Symposium on Programming, ESOP*, 2021. To appear.

[59] 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.

[60] Farzaneh Derakhshan and Frank Pfenning. Circular proofs as session-typed processes: A local validity condition. *CoRR*, abs/1908.01909, August 2019. URL http://arxiv.org/abs/1908.01909.

[61] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019.

[62] 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.

[63] 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.

[64] 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.

[65] Juliana Franco and Vasco T. Vasconcelos. A concurrent programming language with refined session types. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods (SEFM 2013)*, pages 15–28, Madrid, Spain, September 2013. Springer LNCS 8368.

[66] 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.

[67] 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.

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

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

[70] Hannah Gommerstadt. *Session-Typed Concurrent Contracts*. PhD thesis, Carnegie Mellon University, September 2019. Available as Technical Report CMU-CS-19-119.

[71] 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.

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

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

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

[75] Dennis Griffith and Elsa L. Gunter. LiquidPi: Inferrable dependent session types. In *Proceedings of the NASA Formal Methods Symposium*, pages 186–197. Springer LNCS 7871, 2013.

[76] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 51–62, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02658-4.

[77] 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.

[78] 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.

[79] 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.

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

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

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

[83] 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.

[84] 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.

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

[86] 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.

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

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

[89] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In P. Bahr and S. Erdweg, editors, *Proceedings of the 11th Workshop on Generic Programming (WGP 2015)*, pages 13–22, Vancouver, Canada, August 2015. ACM.

[90] 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.

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

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

[93] 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.

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

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

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

[97] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

[98] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL `https://doi.org/10.1145/1538788.1538814`.

[99] Christopher Lidbury and Alastair F. Donaldson. Sparse record and replay with controlled scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 576–593, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314635. URL http://doi.acm.org/10.1145/3314221.3314635.

[100] Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In G. Mainland, editor, *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*, pages 133–145, Nara, Japan, September 2016. ACM.

[101] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 434–447. ACM, 2016. ISBN 9781450342193. doi: 10.1145/2951913.2951921. URL https://doi.org/10.1145/2951913.2951921.

[102] Sam Lindley and J. Garrett Morris. Lightweight functional session types. In S. Gay and A. Ravara, editors, *Behavioural Types: from Theory to Tools*, chapter 12, pages 265–286. River Publishers, June 2017. doi: https://doi.org/10.13052/rp-9788793519817.

[103] 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.

[104] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978309. URL http://doi.acm.org/10.1145/2976749.2978309.

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

[106] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA, 1967. ISBN 0131655639.

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

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

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

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

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

[112] Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In R. Bodik and R. Majumdar, editors, *Proceedings of the 43rd Annual Symposium on Principles of Programming Languages (POPL 2016)*, pages 568–581, St. Petersburg, Florida, January 2016. ACM.

[113] Luca Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27(e4), 2017.

[114] 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.

[115] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[116] 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.

[117] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345100. URL http://doi.acm.org/10.1145/345099.345100.

[118] 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.

[119] Francois Pottier and Yann Régis-Gianas. *Menhir Reference Manual*, 2019.

[120] Marc Pouzet. Lucid synchrone release, version 3.0 tutorial and reference manual, 2006.

[121] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Beguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 983–1002, Los Alamitos, CA, USA, may 2020. IEEE Computer Society. doi: 10.1109/SP40000.2020.00114. URL https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00114.

[122] Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. In F. Martins and D. Orchard, editors, *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*, pages 60–79, Prague, Czech Republic, April 2019. EPTCS 291.

[123] Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020. URL http://arxiv.org/abs/2002.04607.

[124] Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Technical report, Carnegie Mellon University, April 2018.

[125] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic Refinements for Relational Cost Analysis. *Proc. ACM Program. Lang.*, 2(POPL), 2017.

[126] Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, January 2009. URL http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf.

[127] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375602. URL https://doi.org/10.1145/1375581.1375602.

[128] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999. ISSN 0734-2071. doi: 10.1145/312203.312214. URL http://doi.acm.org/10.1145/312203.312214.

[129] 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.

[130] Alceste Scalas and Nobuko Yoshida. Lightweight session programming in Scala. In *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016)*, pages 21:1–21:28, Rome, Italy, July 2016. LICIcs 56.

[131] 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.

[132] Miguel Silva, Mário Florido, and Frank Pfenning. Non-blocking concurrent imperative programming with session types. In *Fourth International Workshop on Linearity*, June 2016.

[133] 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.

[134] 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.

[135] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[136] Tachio Terauchi and Adam Megacz. Inferring channel buffer bounds via linear programming. In *ESOP'08*, 2008.

[137] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *Proceedings of the 21st International Conference on Functional Programming (ICFP 2016)*, pages 462–475, Nara, Japan, September 2016. ACM.

[138] 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.

[139] 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.

[140] Pedro Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.

[141] Vasco T. Vasconcelos. Session, from types to programming languages. *Bulletin of the EATCS*, 103:53–73, 2011.

[142] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 546–566. North, 1990.

[143] Philip Wadler. Propositions as sessions. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012.

[144] Max Willsey, Rokhini Prabhu, and Frank Pfenning. Design and implementation of Concurrent C0. In *Fourth International Workshop on Linearity*, June 2016.

[145] Gavin Wood. Ethereum: A secure decentralized transaction ledger, 2014.

[146] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292560. URL `http://doi.acm.org/10.1145/292540.292560`.

[147] Hongwei Xi, Zhiqiang Ren, Hanwen Wu, and William Blair. Session types in a linearly typed multi-threaded lambda-calculus. *CoRR*, abs/1603.03727, 2016. URL `http://arxiv.org/abs/1603.03727`.

[148] Christoph Zenger. Indexed types. *Theor. Comput. Sci.*, 187(1–2):147–165, November 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(97)00062-5. URL `https://doi.org/10.1016/S0304-3975(97)00062-5`.

[149] Fangyi Zhou, Francisco Ferreira, Rumyana Neykova, and Nobuko Yoshida. Fluid Types: Statically Verified Distributed Protocols with Refinements. In *11th Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*, 2019.

[150] 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.