Resource-Aware Session Types for Digital Contracts

Technical Report

ANKUSH DAS, Carnegie Mellon University
STEPHANIE BALZER, Carnegie Mellon University
JAN HOFFMANN, Carnegie Mellon University
FRANK PFENNING, Carnegie Mellon University
ISHANI SANTURKAR, Carnegie Mellon University

Programming digital contracts comes with unique challenges, which include *(i)* expressing and enforcing protocols of interaction, *(ii)* controlling resource usage, and *(iii)* preventing the duplication or deletion of a contract's assets. This article presents the design and type-theoretic foundation of *Nomos*, a programming language for digital contracts that addresses these challenges. To express and enforce protocols, Nomos is based on *shared binary session types*. To control resource usage, Nomos employs *automatic amortized resource analysis*. To prevent the duplication or deletion of assets, Nomos uses a *linear type system*. A monad integrates the effectful session-typed language with a general-purpose functional language. Nomos' *prototype implementation* features *linear-time type checking* and efficient type reconstruction that includes automatic *inference of resource bounds* via off-the-shelf linear optimization. The effectiveness of the language is evaluated with case studies about implementing common smart contracts such as auctions, elections, and currencies. Nomos is completely formalized, including the type system, a cost semantics, and a transactional semantics to instantiate Nomos contracts on a blockchain. The type soundness proof ensures that protocols are followed at run-time and that types establish sound upper bounds on the resource consumption, ruling out re-entrancy and out-of-gas vulnerabilities, respectively.

1 INTRODUCTION

Digital contracts are programs that implement the execution of a contract. With the rise of blockchains and cryptocurrencies such as Bitcoin [Nakamoto 2008], Ethereum [Wood 2014], and Tezos [Goodman 2014], digital contracts have become popular in the form of smart contracts, which provide potentially distrusting parties with programmable money and a distributed consensus mechanism. Smart contracts are used to implement auctions [Auc 2016], investment instruments [Siegel 2016], insurance agreements [Initiative 2008], supply chain management [Law 2017], and mortgage loans [Morabito 2017]. They hold the promise to lower cost, increase fairness, and expand access to the financial infrastructure. However, like all software, smart contracts can contain bugs and security vulnerabilities [Atzei et al. 2017], which have direct financial consequences. A well-known example is the attack on The DAO [Siegel 2016], resulting in a $60 million dollar theft by exploiting a contract vulnerability.

Today's contract languages are typically derived from existing general-purpose languages like JavaScript (Ethereum's Solidity [Auc 2016]), Go (Hyperledger project [Cachin 2016]), or OCaml (Tezos' Liquidity [Liq 2018]), which fail to accommodate the domain-specific requirements of digital contracts. These requirements are: *(i)* expressing and enforcing protocols of interaction, *(ii)* controlling resource (or gas) usage, and *(iii)* preventing duplication or discard of a contract's assets.

Authors' addresses: Ankush Das, Carnegie Mellon University, ankushd@cs.cmu.edu; Stephanie Balzer, Carnegie Mellon University, balzers@cs.cmu.edu; Jan Hoffmann, Carnegie Mellon University, jhoffmann@cmu.edu; Frank Pfenning, Carnegie Mellon University, fp@cs.cmu.edu; Ishani Santurkar, Carnegie Mellon University, ivs@andrew.cmu.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

In this article, we present the type-theoretic foundation of Nomos, a programming language for digital contracts that accommodates the aforementioned requirements by construction.

To express and enforce the protocols underlying a contract, Nomos is based on *session types* [Honda 1993; Honda et al. 1998, 2008], in particular on the works rooted in the Curry-Howard correspondence between linear logic and the session-typed process calculus [Caires and Pfenning 2010; Pfenning and Griffith 2015; Toninho et al. 2013; Wadler 2012]. Session types capture protocols of interactions in the type, and type-checking statically guarantees adherence to those protocols at run-time. Session types make the core functionality of a contract and its intended interactions with various parties explicit, rather than buried in implementation code. Delimiting the sequences of actions that must be executed atomically, session types moreover prevent interception of a contract in an inconsistent state, as is possible through re-entrancy in some contract languages.

To control resource usage, Nomos employs and further develops *automatic amortized resource analysis (AARA)*, a type-based technique for automatically inferring symbolic resource bounds [Carbonneaux et al. 2017; Hoffmann et al. 2011, 2017; Hofmann and Jost 2003; Jost et al. 2010]. AARA is parametric in the cost model, making it directly applicable to track gas cost of Nomos contracts. A unique feature of Nomos' resource-aware type system is that it allows contracts to store gas in internal data structures to amortize the cost of resource intensive transactions. Failure to support estimation of gas usage bares the risk of high losses in case transactions fail due to dynamic out-of-gas exceptions and makes contracts vulnerable to denial-of-service attacks.

To prevent duplication or deletion of assets, Nomos uses *linearity* [Girard 1987], which naturally arises from the Curry-Howard correspondence established between linear logic and the session-typed process calculus [Caires and Pfenning 2010; Pfenning and Griffith 2015; Wadler 2012]. Accidental or malicious duplication and deletion is a source of major concern in today's contract languages [Meredith 2015]. To support the writing of general-purpose programs, Nomos moreover complements the session-typed language with a *functional language*, using a contextual monad to shield expressions from effectful processes.

We formalize Nomos by giving its type system and operational semantics and by proving type safety. Integrating the seemingly disparate approaches (session types, resource analysis, linearity, and functional programming) and combining them with the different roles that arise in a digital contract (contract, asset, transaction) in a way that the result remains consistent, presents unique challenges. For one, both the functional as well as session-typed language use potential annotations to predict the resource consumption, which requires care when functional values are exchanged as messages between processes. For another, prior work on integrating shared and linear session types [Balzer and Pfenning 2017] preclude contracts from persisting their linear assets across transactions, a feature essential to digital contract development; a restriction that we lift in this work. Fundamental is the use of different forms of *typing judgments* for expressions and processes along with *judgmental modes* to distinguish the different roles in a digital contract. The modes are essential in ensuring type safety, as they allow the expression of mode-indexed invariants on the typing contexts and their enforcement by the typing rules.

A challenge in Nomos' design was the sound integration of session types, resource analysis, linearity, and functional programming so that type checking is linear in the size of the program and resource bounds can efficiently inferred with an off-the-shelf LP solver. Efficient type checking is particularly important if type-checking is part of contract validation and can be used for denial-of-service attacks.

To evaluate Nomos, we implemented a publicly available open-source prototype [Nom 2019] and conducted 8 case studies implementing common smart contracts such as auctions, elections, and currencies. Our experiments show that type-checking overhead is less than 0.7 ms for each contract and bound inference (needed once at deployment) takes less than 10 ms. Moreover, gas bounds are tight for most contracts.

Our contributions are:

- design of Nomos, a language that addresses the domain-specific requirements of digital contracts by construction;
- a fine-tuned system of typing judgments (Section 4) that uses *modes* to orchestrate the sound integration of session types (Section 3), functions (Section 5), and resource analysis (Section 6);
- extension of shared session types to support linear assets;
- resource cost amortization by allowing gas storage in internal data structures (Section 6);
- type safety proof of Nomos using a novel asynchronous cost semantics (Section 7);
- a prototype implementation and case study of prominent blockchain applications (Section 8);
- a transactional semantics to instantiate Nomos contracts and transactions on a blockchain (Section 9).

In addition, Section 2 provides an overview of Nomos' main features based on an example. Section 9 discusses known limitations. Section 10 reviews related work, and Section 11 concludes this article with future directions. The appendix formalizes the complete language with typing rules, cost semantics, and the type safety theorem and proof. It also shows the implementation of all the smart contract applications used in the main paper.

2 NOMOS BY EXAMPLE

This section introduces the main features of Nomos using a simple auction contract as an example. The subsequent sections explain each feature in technical detail.

Explicit Protocols of Interaction. Digital contracts, like traditional contracts, follow a *predefined protocol*. For instance, an auction contract follows the protocol of a bidding phase where bidders submit their bids to the auctioneer (possibly multiple times), followed by a collection phase where the highest bidder receives the lot while all other bidders receive their bids back. In existing smart contract languages, like Solidity [Auc 2016], the bidding part of the auction is typically implemented using the bid function below. This function receives a bid (msg.value) from a bidder (msg.sender) and adds it to their total previous bids, stored in the bidValue hash map.

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

The above code does not guarantee that a bid can only be placed in the bidding phase. To enforce this constraint, we can introduce a state variable, such as status, to track the different phases of a contract. Using this variable we can guard the above code block with the precondition

```
require (status == running)
```

checking whether the auction is still running and thus accepts bids. The precondition is checked at run-time, aborting the execution if the condition is not met. It is the responsibility of the programmer to define state variables, update them, and corresponding guard functions.

Rather than burying the contract's interaction protocol in implementation code using state variables and run-time checks, in Nomos, protocols can be expressed explicitly using a *session type*. Type-checking then makes sure that the program implements the protocol defined by the session type. The auction's protocol amounts to the below session type:

$\mathsf{auction} = \uparrow_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{22} \oplus \{\mathbf{running} : \&\{\mathbf{bid} : \mathsf{id} \rightarrow \mathsf{money} \multimap \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction},$ % recv bid from client

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

We first focus on how the session type defines the main interactions of a contract with a bidder, ignoring the operators $\uparrow_{\mathsf{L}}^{\mathsf{S}}$, $\downarrow_{\mathsf{L}}^{\mathsf{S}}$, $\triangleleft$, and $\triangleright$ for now. To distinguish the main two states an auction can be in, the session type uses the internal choice type constructor ($\oplus$), leading the contract to either send the label running or ended, depending on whether the auction still accepts bids or not. 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 still running, the client can choose between placing a bid (label bid) or backing out (cancel). If the client chooses to place a bid, they have to indicate their identifier (type id), followed by a payment (type money). Nomos session types allow transfer of both non-linear values that can be duplicated or discarded (e.g. id), using the arrow ($\rightarrow$) constructor, and linear assets, using the linear implication ($\multimap$) constructor. Using a linear type to represent digital money (money) makes sure that such a value can neither be duplicated nor lost. Should the auction have ended, the client can 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 (commodity being auctioned, represented as a linear type), in the latter case, it will return the client's bid. The linear product ($\otimes$) constructor 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.

Our discussion so far describes the interaction of one client with the auction, prescribing the sequences of steps to be taken according to the protocol defined by the session type. In reality, however, an auction will have several clients. Nomos uses a *shared* session type [Balzer and Pfenning 2017] 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 $\uparrow_{\mathsf{L}}^{\mathsf{S}}$ and $\downarrow_{\mathsf{L}}^{\mathsf{S}}$ type modalities. The $\uparrow_{\mathsf{L}}^{\mathsf{S}}$ modality denotes the beginning of a critical section, the $\downarrow_{\mathsf{L}}^{\mathsf{S}}$ modality denotes its end. Programmatically, $\uparrow_{\mathsf{L}}^{\mathsf{S}}$ translates into an *acquire* of the auction session and $\downarrow_{\mathsf{L}}^{\mathsf{S}}$ into the *release* of the session. As indicated by the auction session type, acquire and release tend to be the begin and end points of a session, framing the critical section that is described by a *linear* session type.

In Nomos, contracts are implemented by *processes*, revealing the concurrent, message-passing nature of session-typed languages. The implementation below shows the process *run* representing the running phase of the auction. It internally stores a linear hash map of bids $b : \mathsf{hashmap}_{\mathsf{bid}}$ and a linear lot l and offers on a shared channel $sa : \mathsf{auction}$. 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 1 shows the syntax for session type definitions.

```
1: stype bid = &{addr : id × bid, val : money},    stype bids = hashmap_bid
2: (b : bids), (l : lot) ⊢ run :: (sa : auction)    %  syntax for process declaration
3:   sa ← run ← b l =                               %  syntax for process definition
4:     la ← accept sa ;                             %  accept a client acquire request
5:     la.running ;                                 %  auction is running
6:     case la ( bid ⇒ r ← recv la ;                %  receive identifier r : id
7:                     m ← recv la ;                %  receive bid m : money
8:                     sa ← detach la ;             %  detach from client
9:                     b′ ← addbid r ← b m ;        %  store bid internally
```

10: $sa \leftarrow \mathit{check} \leftarrow b'\ l$ % check if threshold reached
11: $\mid$ **cancel** $\Rightarrow sa \leftarrow$ detach la ; % detach from client
12: $sa \leftarrow \mathit{run} \leftarrow b\ l)$ % 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. It then waits 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.

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

Re-Entrancy Vulnerabilities. A contract function is re-entrant if, once called by an external user, it can potentially be called again before the previous call is completed. As an illustration, consider the collect function in Solidity of the auction contract (on the left) 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(); }
```

If a bidder creates a dummy contract with a function that calls collect on the auction contract, it causes a re-entrant situation. The send function on the left transfers execution control to the dummy contract, essentially triggering an unnamed *fallback* function (on the right) in the dummy contract code base. The fallback function in turn calls collect on the auction leading to an infinite recursive call to collect, depleting all funds from the auction. This vulnerability was exposed by the infamous DAO attack [Siegel 2016], where $60 million worth of funds were stolen, and detecting them has since been critical [Grossman et al. 2017]. 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, and the second acquire would fail to execute.

Resource Cost. Another important aspect of digital contracts is their *resource usage*. The state of all the contracts is stored on the *blockchain*, a distributed ledger which records the history of all transactions. Executing a contract function, aka *transaction* and updating the blockchain state 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 is important because the sender of a transaction must pay this fee to the miners. 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 [Das et al. 2018] 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. Resource-aware session types express the potential as part of the session type using the type constructors $\triangleleft$ and $\triangleright$. The $\triangleleft$ constructor prescribes that the client must send potential to the contract, with the amount of potential indicated as a superscript. Dually, the $\triangleright$ constructor 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. In the cancel branch of the auction type, on the other hand, the contract returns 21 units of potential to the client using the $\triangleright^{21}$ type constructor. This is analogous to gas usage in smart contracts, where the sender initiates a transaction with some initial gas, and the leftover gas at the end of transaction is returned to the sender. In contrast to existing smart contract languages like Solidity, which provide no support for analyzing the cost of a transaction, Nomos type soundness theorem guarantees that the total initial potential of a process plus the potential it receives during a session reflect the upper bound on the gas usage, assuming that the cost model assigns a cost equivalent to their gas cost to each operation.

Bringing It All Together. A main contribution of this paper is to combine all these features in a single language while retaining type safety. To this end, we introduce four different *modes* of a channel, identifying the role of the process offered along that channel. The mode R denotes *purely linear processes*, typically amounting to linear assets or private data structures, such as b and l in the auction. The modes S and L denote *sharable processes* that are either in their shared phase or linear phase, respectively, and are typically used for contracts, such as sa and la, respectively, in the auction. The mode T, finally, denotes a *transaction process* that can refer to shared and linear processes and is typically issued by a user, such as bidder in the auction. The mode assignment carries over into the process typing judgments (see Section 4) ascertaining certain well-formedness conditions (Definition 1) on their type. This introduction of modes is simply a technical device to preserve the tree structure of linear processes at run-time, establishing type safety.

3 BASE SYSTEM OF SESSION TYPES

Nomos builds on linear session types for message-passing concurrency [Caires and Pfenning 2010; Honda 1993; Honda et al. 1998, 2008; Wadler 2012] and, in particular, on the line of works that have a logical foundation due to the existence of a Curry-Howard correspondence between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Wadler 2012]. Linear logic [Girard 1987] is a substructural logic that exhibits exchange as the only structural property, with no contraction or weakening. As a result, linear propositions can be viewed as resources that must be used *exactly once* in a proof. Under the Curry-Howard correspondence, an intuitionistic linear sequent $A_1, A_2, \ldots, A_n \vdash C$ can be interpreted as the offer of a session C by a process P using the sessions $A_1, A_2, \ldots, A_n$

$$(x_1 : A_1), (x_2 : A_2), \ldots, (x_n : A_n) \vdash P :: (z : C)$$

Session Type	Continuation	Process Term	Continuation	Description
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$c.k \; ; \; P$	P	provider sends label k along c
		$\text{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$	Q_k	client receives label k along c
$c : \&\{\ell : A_\ell\}$	$c : A_k$	$\text{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}$	P_k	provider receives label k along c
		$c.k \; ; \; Q$	Q	client sends label k along c
$c : A \otimes B$	$c : B$	$\text{send}\ c\ w \; ; \; P$	P	provider sends channel $w : A$ on c
		$y \leftarrow \text{recv}\ c \; ; \; Q_y$	$[w/y]Q_y$	client receives channel $w : A$ on c
$c : A \multimap B$	$c : B$	$y \leftarrow \text{recv}\ c \; ; \; P_y$	$[w/y]P_y$	provider receives chan. $w : A$ on c
		$\text{send}\ c\ w \; ; \; Q$	Q	client sends channel $w : A$ on c
$c : \mathbf{1}$	–	$\text{close}\ c$	–	provider sends *end* along c
		$\text{wait}\ c \; ; \; Q$	Q	client receives *end* along c

Table 1. Overview of binary session types with their operational description

We label each antecedent as well as the conclusion with the name of the channel along which the session is provided. The x_i's correspond to channels *used by* P, and z is the channel *provided by* P. As is standard, we use the linear context Δ to combine multiple assumptions.

For the typing of processes in Nomos, we extend the above judgment with two additional contexts (Ψ and Γ), a resource annotation q, and a mode m of the offered channel:

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

We will gradually introduce each concept in the remainder of this article. For future reference, we show the complete typing rules, with additional contexts, resource annotations, and modes henceforth, but highlight the parts that will be discussed in later sections in blue.

The Curry-Howard correspondence gives each connective of linear logic an interpretation as a session type, as demonstrated by the grammar:

$$A, B \quad ::= \quad \oplus\{\ell : A\}_{\ell \in K} \mid \&\{\ell : A\}_{\ell \in K} \mid A \multimap_m B \mid A \otimes_m B \mid \mathbf{1}$$

Each type prescribes the kind of message that must be sent or received along a channel of that type and at which type the session continues after the exchange. Types are defined mutually recursively in a global signature, where type definitions are constrained to be *contractive* [Gay and Hole 2005] (no definitions of the form $V = A$ where A is a type name). This allows us to treat them equi-recursively [Crary et al. 1999], meaning we can silently replace a type variable by its definition for type-checking.

Following previous work on session types [Pfenning and Griffith 2015; Toninho et al. 2013], the process expressions of Nomos are defined as follows.

$$P ::= x.l \; ; \; P \mid \text{case}\ x\ (\ell \Rightarrow P)_{\ell \in K} \mid x \leftarrow y \mid \text{close}\ x \mid \text{wait}\ x \; ; \; P \mid \text{send}\ x\ w \; ; \; P \mid y \leftarrow \text{recv}\ x \; ; \; P$$

Table 1 provides an overview of the types along with their operational meaning. Because we adopt the intuitionistic version of linear logic, session types are expressed from the point of view of the provider. Table 1 provides the viewpoint of the provider in the first line, and that of the client in the second line for each connective. Columns 1 and 3 describe the session type and process term before the interaction. Similarly, columns 2 and 4 describe the type and term after the interaction. Finally, the last column describes the provider and client action. Figure 1 provides the corresponding typing rules. As illustrations of the statics and semantics, we explain internal choice ($\oplus$) and linear implication ($\multimap$) connectives.

$\boxed{\Psi \; ; \; \Gamma \; ; \Delta \vdash^{q} P :: (x_m : A)}$ Process P uses linear channels in Δ and offers type A on channel x

$$\frac{\Psi \; ; \; \Gamma \; ; \Delta \vdash^{q} P :: (x_m : A_l) \qquad (l \in K)}{\Psi \; ; \; \Gamma \; ; \Delta \vdash^{q} x_m.l \; ; \; P :: (x_m : \oplus\{\ell : A_\ell\}_{\ell \in K})} \oplus R$$

$$\frac{\Psi \; ; \; \Gamma \; ; \Delta, (x_m : A_\ell) \vdash^{q} Q_\ell :: (z_k : C) \qquad (\forall \ell \in K)}{\Psi \; ; \; \Gamma \; ; \Delta, (x_m : \oplus\{\ell : A_\ell\}_{\ell \in K}) \vdash^{q} \text{case } x_m \; (\ell \Rightarrow Q_\ell)_{\ell \in K} :: (z_k : C)} \oplus L$$

$$\frac{\Psi \; ; \; \Gamma \; ; \Delta, (y_n : A) \vdash^{q} P :: (x_m : B)}{\Psi \; ; \; \Gamma \; ; \Delta \vdash^{q} y_n \leftarrow \text{recv } x_m \; ; \; P :: (x_m : A \multimap_n B)} \multimap_n R$$

$$\frac{\Psi \; ; \; \Gamma \; ; \Delta, (x_m : B) \vdash^{q} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \Delta, (w_n : A), (x_m : A \multimap_n B) \vdash^{q} \text{send } x_m \; w_n \; ; \; Q :: (z_k : C)} \multimap_n L$$

$$\frac{q = 0}{\Psi \; ; \; \Gamma \; ; (y_m : A) \vdash^{q} x_m \leftarrow y_m :: (x_m : A)} \text{fwd}$$

Fig. 1. Selected typing rules for process communication

Internal Choice. The linear logic connective $A \oplus B$ has been generalized to n-ary labeled sum $\oplus\{\ell : A_\ell\}_{\ell \in K}$. A process that provides $x : \oplus\{\ell : A_\ell\}_{\ell \in K}$ can send any label $l \in K$ along x and then continues by providing $x : A_l$. The corresponding process term is written as $(x.l \; ; \; P)$, where P is the continuation. A client branches on the label received along x using the term case $x \; (\ell \Rightarrow Q_\ell)_{\ell \in K}$. The typing rules for the provider and client are $\oplus R$ and $\oplus L$, respectively, in Figure 1.

The operational semantics is formalized as a system of *multiset rewriting rules* [Cervesato and Scedrov 2009]. We introduce semantic objects $\text{proc}(c_m, w, P)$ and $\text{msg}(c_m, w, N)$ denoting process P and message N, respectively, being provided along channel c at mode m. The resource annotation w indicates the work performed so far, the discussion of which we defer to Section 6. Communication is *asynchronous*, allowing the sender $(c_m.l \; ; \; P)$ to continue with P without waiting for l to be received. As a technical device to ensure that consecutive messages arrive in the order they were sent, the sender also creates a fresh continuation channel c_m^+ so that the message l is actually represented as $(c_m.l \; ; \; c_m \leftarrow c_m^+)$ (read: send l along c_m and continue as c_m^+):

$$(\oplus S) : \quad \text{proc}(c_m, w, c_m.l \; ; \; P) \mapsto \text{proc}(c_m^+, w, [c_m^+/c_m]P), \text{msg}(c_m, 0, c_m.l \; ; \; c_m \leftarrow c_m^+)$$

Receiving the message l corresponds to selecting branch Q_l and substituting continuation c^+ for c:

$$(\oplus C) : \quad \text{msg}(c_m, w, c_m.l \; ; \; c_m \leftarrow c_m^+), \text{proc}(d_k, w', \text{case } c_m \; (\ell \Rightarrow Q_\ell)_{\ell \in K}) \mapsto \text{proc}(d_k, w + w', [c_m^+/c_m]Q_l)$$

The message $\text{msg}(c_m, w, c_m.l \; ; \; c_m \leftarrow c_m^+)$ is just a particular form of process, where $c_m \leftarrow c_m^+$ is *forwarding*, which is explained below. Therefore, no separate typing rules for messages are needed; they can be typed as processes [Balzer and Pfenning 2017].

Channel Passing. Nomos allows the exchange of channels over channels, also referred to as higher-order channels. A process providing $A \multimap_n B$ can receive a channel of type A at mode n and then continue with providing B. The provider process term is $(y_n \leftarrow \text{recv } x_m \; ; \; P)$, where P is the continuation. The corresponding client sends this channel using $(\text{send } x_m \; w_n \; ; \; Q)$. The corresponding typing rules are presented in Figure 1. Operationally, the client creates a message containing the channel:

$$(\multimap_n S) : \text{proc}(d_k, w, \text{send } c_m \; e_n \; ; \; P) \mapsto \text{msg}(c_m^+, 0, \text{send } c_m \; e_n \; ; \; c_m^+ \leftarrow c_m), \text{proc}(d_k, w, [c_m^+/c_m]P)$$

The provider receives this channel, and substitutes it appropriately.

$$(\multimap_n C) : \mathsf{proc}(c_m, w', x_n \leftarrow \mathsf{recv}\ c_m \ ; \ Q), \mathsf{msg}(c_m^+, w, \mathsf{send}\ c_m\ e_n \ ; \ c_m^+ \leftarrow c_m) \mapsto \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m][e_n/x_n]Q)$$

An important distinction from standard session types is that the $\multimap$ and $\otimes$ types are decorated with the mode m of the channel exchanged. Since modes distinguish the status of the channels in Nomos, this mode decoration is necessary to ensure type safety.

Forwarding. A forwarding process $x_m \leftarrow y_m$ (which provides channel x) identifies channels x and y (both at mode m) so that any further communication along x or y occurs on the unified channel. The typing rule fwd is given in Figure 1 and corresponds to the logical rule of *identity*.

$$\begin{array}{llcl} (\mathsf{id}^+C): & \mathsf{msg}(d_m, w', N), \mathsf{proc}(c_m, w, c_m \leftarrow d_m) & \mapsto & \mathsf{msg}(c_m, w + w', [c_m/d_m]N) \\ (\mathsf{id}^-C): & \mathsf{proc}(c_m, w, c_m \leftarrow d_m), \mathsf{msg}(e_k, w', N(c_m)) & \mapsto & \mathsf{msg}(e_k, w + w', N(d_m)) \end{array}$$

Operationally, a process $c \leftarrow d$ *forwards* any message N that arrives along d to c and vice versa. Since linearity ensures that every process has a unique client, forwarding results in terminating the forwarding process and corresponding renaming of the channel in the client process. The full semantics are given in the appendix.

4 SHARING CONTRACTS

Multi-user support is fundamental to digital contract development. Linear session types, as defined in Section 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 [Balzer and Pfenning 2017]. 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* [Benton 1994; Pfenning and Griffith 2015; Reed 2009], 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 3). 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.

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

Fig. 2. Grammar for shared session types

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 [Balzer and Pfenning 2017] 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 [Pruiksma et al. 2018] and also crucial for type preservation.

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 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 $A \in \mathbf{L}(A_{\mathsf{L}})$.
(4) If $m = \mathsf{T}$, then $A \in \mathbf{L}(A_{\mathsf{T}})$.

Figure 2 shows the session type grammar in Nomos. The first sub-judgment in Definition 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 2) 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 arbitrary modes (L, T or R). *This allows contracts to interact with other contracts without compromising type safety.* 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 and the fact that contracts, as the only shared processes, can only access linear channels at mode R, upholds preservation while allowing shared contract processes to rely on linear resources.

Shared session types introduce new typing rules into our system, concerning the *acquire-release* constructs (see Figure 3). An acquire is applied to the shared channel x_{S} along which the shared process offers and yields a linear channel x_{L} when successful. A contract process can *accept* an acquire request along its offering shared channel x_{S}. After the accept is successful, the shared contract process transitions to its linear phase, now offering along the linear channel x_{L}.

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

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

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

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

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

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

Fig. 3. Typing rules corresponding to the shared layer.

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

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

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

5 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 classes of data in the type system, we take inspiration from prior work [Pfenning and Griffith 2015; Toninho et al. 2013] 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, and the contextual monadic types in the last line, which are the topic of this

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

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

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

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

Fig. 4. Typing rules corresponding to the functional layer.

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 $\{A_R \leftarrow \overline{A_R}\}_R$ denotes a process offering a *purely linear* session type A_R and using the purely linear vector of types $\overline{A_R}$. The corresponding introduction form in the functional language is the monadic value constructor $\{c_R \leftarrow P \leftarrow \overline{d_R}\}$, denoting a runnable process offering along channel c_R that uses channels $\overline{d_R}$, all at mode R. The corresponding typing rule for the monad is (ignore the blue portions)

$$\frac{\Delta = \overline{d_R : D} \qquad \Psi \; ; \; \cdot \; ; \; \Delta \vDash^{q} P :: (x_R : A)}{\Psi \Vdash^{q} \{x_R \leftarrow P \leftarrow \overline{d_R}\} : \{A \leftarrow \overline{D}\}_R} \; \{\}I_R$$

The monadic *bind* operation implements process composition and acts as the elimination form for values of type $\{A_R \leftarrow \overline{A_R}\}_R$. The bind operation, written as $c_R \leftarrow M \leftarrow \overline{d_R} \; ; \; Q_c$, composes the process underlying the monadic term M, which offers along channel c_R and uses channels $\overline{d_R}$, with Q_c, which uses c_R. The typing rule for the monadic bind is rule $\{\}E_{RR}$ in Figure 4. 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. The shared context Γ is empty in accordance with the invariants established in Definition 1 *(i)*, since the mode of offered channel z 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 spawn_{RR}) is given as follows:

$$\mathsf{proc}(d_R, w, x_R \leftarrow \{x'_R \leftarrow P_{x'_R, \overline{y}} \leftarrow \overline{y}\} \leftarrow \overline{a} \; ; \; Q) \mapsto \mathsf{proc}(c_R, 0, P_{c_R, \overline{a}}), \mathsf{proc}(d_R, w, [c_R/x_R]Q)$$

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 $\{A_S \leftarrow \overline{A_S} \; ; \; \overline{A_R}\}_S$ and a transaction process $\{A_T \leftarrow \overline{A_S} \; ; \; \overline{A}\}_T$ at mode S and T, respectively. Their rules are analogous to $\{\}I_R$ and $\{\}E_{RR}$.

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.

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

The type $\tau \rightarrow A$ 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 ($\rightarrow R$, $\rightarrow L$) are given in Figure 4 (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:

$$(\rightarrow S) : \mathsf{proc}(d_k, w, \mathsf{send}\ c_m\ M\ ;\ P) \mapsto \mathsf{msg}(c_m^+, 0, \mathsf{send}\ c_m\ M\ ;\ c_m^+ \leftarrow c_m), \mathsf{proc}(d_k, w, [c_m^+/c_m]P)$$

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.

$$(\rightarrow C) : \mathsf{proc}(c_m, w', x \leftarrow \mathsf{recv}\ c_m\ ;\ Q), \mathsf{msg}(c_m^+, w, \mathsf{send}\ c_m\ M\ ;\ c_m^+ \leftarrow c_m) \mapsto \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m][M/x]Q)$$

Tracking Linear Assets. As an illustration, consider the type money introduced in the auction example (Section 2). The type is an abstraction over funds stored in a process and is described as

```
money = &{value : int × money,                                % send value
          add : money ⊸R money,                               % receive money and add it
          subtract : int → ⊕{sufficient : money ⊗R money,     % receive int, send money
                             insufficient : money}            % funds insufficient to subtract
          coins : list_coin}                                  % send list of coins
```

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)

```
(n : int) ; (l_R : list_coin) ⊢ wallet :: (m_R : money)
  m ← wallet n ← l =
    case m                                   %  case analyze on label received on m
      (value ⇒ send m n ;                    %  receive value, send n
                 m ← wallet n ← l
      | add ⇒ m′ ← recv m ;                  %  receive m′ : money to add
               m′.value ;                    %  query value of m′
               v ← recv m′ ;
               m′.coins ;                    %  extract list of coins stored in m′
               k ← append ← l m′ ;           %  append list received to internal list
               m ← wallet (n + v) ← k
      | subtract ⇒ n′ ← recv m ;             %  receive int to subtract
                    if (n′ > n) then
                      m.insufficient ;       %  funds insufficient
                      m ← wallet n ← l
                    else
                      m.sufficient ;         %  funds sufficient
```

```
18:                 l′ ← remove n′ ← l ;     % remove n′ coins from l
19:                 k ← recv l′ ;            % and create its own list
20:                 m′ ← wallet n′ ← k ;     % new wallet process for subtracted funds
21:                 send m m′ ;              % send new money channel to client
22:                 m ← wallet (n − n′) ← l′
23:      | coins ⇒ m ← 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 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 bare the risk of high losses if the contract has flaws or vulnerabilities.

It is not trivial to decide on the right amount for the fee since the gas cost of the contract does not only depend on the requested transaction but also on the (a priori unknown) state of the blockchain. Thus, precise and static estimation of gas cost facilitates transactions and reduces risks. 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 [Avanzini et al. 2015; Cicek et al. 2017; Danner et al. 2015; Lago and Gaboardi 2011; Radiček et al. 2017]. In Nomos, we adapt the work on automatic amortized resource analysis (AARA) [Hoffmann et al. 2011; Hofmann and Jost 2003] that has been implemented in Resource Aware ML (RaML) [Hoffmann et al. 2017]. 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. An imperative version of the same function in Solidity is implemented in Section 8. 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.

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

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 [Hoffmann et al. 2011, 2017].

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 [Hoffmann et al. 2017; Hofmann and Jost 2003] and the appendix.

Process Layer. To bound the resource usage of a process, Nomos features resource-aware session types [Das et al. 2018] 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 [Das et al. 2018].

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

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 [Das et al. 2018], 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.

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

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

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

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

Fig. 5. Selected typing rules corresponding to 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 5 shows the rules that interact with the potential annotations. In the rule $\triangleleft R$, process P storing potential q receives r units along the offered channel x_m using the *get* construct and the continuation executes with $p = q + r$ units of potential. In the dual rule $\triangleleft L$, a process storing potential $q = p + r$ sends r units along the channel x_m in its context using the *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.

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 4. A process storing potential $r = p + q$ can spawn a process corresponding to the monadic value M, if M needs p units of potential to evaluate, while the continuation needs q units of potential to execute. Moreover, the functional context Ψ is shared in the two premises as Ψ_1 and Ψ_2 using the judgment $\Psi \curlyvee (\Psi_1, \Psi_2)$. This judgment, already explored in prior work [Hoffmann et al. 2017] 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 $\rightarrow L$ follows a similar pattern. 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. As an illustration, we revisit the auction protocol introduced in Section 2. Suppose the bids were stored in a list, instead of a hash map, thus making the cost of collection of winnings linear in the worst case, rather than constant. A user would then be required to send a linear potential after acquiring the contract. This can be done by sending a natural number $n : \mathsf{nat}^q$, storing potential $q \cdot |n|$ (like a unary list), where q is the cost of iterating over an element in the list of bids. The contract would then iterate over the first n elements of the list and refund the remaining gas if n exceeds the length. Since the auction state is public, a user can view the size of the list of bids, compute the required potential, store it in a natural number, and transfer it. It would still be possible that a user does not provide enough fuel to reach the sought-after element in the list. However, this behavior is clearly visible in the protocol and code and out-of-gas exceptions are not possible.

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.

7 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 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 1 (Invariants). *The typing rules on the judgment $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} (x_m : A)$ preserve the invariants outlined in Definition 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 imposes 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 appendix 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* [Gay and Hole 2005] 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 1 (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 [Hoffmann et al. 2017]. 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 [Pfenning and Griffith 2015]. 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* [Balzer and Pfenning 2017] 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 2 (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.

8 IMPLEMENTATION AND EVALUATION

We have developed an open-source prototype implementation [Nom 2019] of Nomos in OCaml. This prototype contains a lexer and parser (369 lines of code), a type checker (3039 lines of code), a pretty printer (500 lines of code), and an LP solver interface (914 lines of code).

Syntax. The lexer and parser for Nomos have been implemented in Menhir [Pottier and Régis-Gianas 2019], an LR(1) parser generator for OCaml. A Nomos program is a list of mutually recursive type and process definitions. 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. We also require the programmer to indicate the mode of the process being defined: *asset*, *contract* or *transaction*, assigning the respective modes R, S and T to the offered channel. The modes for all other channels are inferred automatically (explained later). The initial potential $\{q\}$ of a process is marked on the turnstile in the declaration. The syntax for definitions is

```
stype v = A
proc <mode> f : (x1 : T), ($c2 : A), ... |{q}- ($c : A) = M
```

In the context, T is the functional type for variable x1, while A is the session type for channel $c2 and M is a functional expression implementing the process. We add syntactic sugar, such as the forms let x = M; P and if M then P_1 else P_2, to the process layer to ease programming. Finally, a functional expression can enter the session type monad using {}, i.e., M = {P} where *P* is a session-typed expression.

Type Checking. We implemented a bi-directional [Pierce and Turner 2000] type checker with a specific focus on the quality of error messages, which include, for example, *extent* (source code location) information for each definition and expression. 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. Type equality is implemented using a standard co-inductive algorithm [Gay and Hole 2005]. *Type checking is linear time in the size of the program*, which is important in settings where type checking is part of the attack surface.

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 inference algorithm of both these annotations.

Using ideas from existing techniques for type inference for AARA [Hoffmann et al. 2017; Hofmann and Jost 2003], we reduce the reconstruction of potential annotations to linear optimization. To this end, Nomos' type checker 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.

The inference engine iterates over the program and substitutes the star annotations with potential variables and 'unknown' with mode variables. Then, the bidirectional typing rules are applied, approximately checking the program (modulo potential and mode annotations) while also generating linear constraints for potential annotations (see Figure 4). and mode annotations (see Definition 1 and Figure 3). Finally, these constraints are shipped to the LP solver, which is minimizing 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. The final program is pretty printed for the programmer to view and verify the potential and mode annotations.

8.1 Case Studies

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. Tokens are a representation of a particular asset or utility, that resides on top of an existing blockchain. ERC-20 [ERC 2018] 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 standard requires the following functions to be implemented:

- totalSupply() : returns the total number of tokens in supply as an integer.
- balanceOf(id *owner*) : returns the account balance of *owner*.
- transfer(id *to*, int *value*) : transfers *value* tokens from sender's account to identifier *to*.
- transferFrom(id *from*, id *to*, int *value*) : transfers *value* number of tokens from identifier *from* to identifier *to*.
- approve(id *spender*, int *value*) : allows *spender* to withdraw from sender's account up to *value* number of tokens.
- allowance(id *owner*, id *spender*) : returns the number of tokens *spender* is allowed to withdraw from *owner*.

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

```
type 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.

A programmer can design their own implementation (contract) of the erc20token session type. Internally, the contract relies on custom coins created and named by its owner and used exclusively for exchanges among private accounts. These coins can be minted by a special transaction that can only be issued by the owner and that creates coins out of thin air (consuming gas to create coins). Depending on the functionality intended by the owner, they can employ different types to represent their coins. For instance, choosing type $\mathbf{1}$, the multiplicative unit from linear logic, will allow both creation and destruction of coins "for free". A *mint-one* process, typed as $\cdot \vdash$ *mint-one* :: $(c : \mathbf{1})$, can create coin c out of thin air (by closing channel c) and a *burn-one* process, typed as $(c : \mathbf{1}) \vdash$ *burn-one* :: $(d : \mathbf{1})$, will destroy the coin c (by waiting on channel c). Nomos' linear type system enforces that the coins are treated linearly modulo minting and burning. Any transaction that does not involve minting or burning ensures linearity of these coins.

One specific implementation of the erc20token session type can be achieved by storing two lists, one for the balance of each account, and one for the allowance between each pair of accounts. The account balance needs to be treated linearly, hence we place this balance list in the linear context, while we store the allowance list in the functional context. In this contract, we call the custom coin plcoin, and use plcoins to mean $\text{list}_{\text{plcoin}}$. The account balance is abstracted using the account type:

$$\begin{array}{lll} \text{account} = \&\{\textbf{addr} : \text{id} \times \text{account}, & \% & \text{send identifier} \\ \qquad\qquad \textbf{add} : \text{plcoins} \multimap \text{account}, & \% & \text{receive pl coins and add internally} \\ \qquad\qquad \textbf{subtract} : \text{int} \rightarrow \text{plcoins} \otimes \text{account}\} & \% & \text{receive integer, send pl coins} \end{array}$$

This allows a client to query for the identifier stored in the account, as well as add and subtract from the account balance. We ignore the resource consumption as it is not relevant to the example. The *balance* process provides the account abstraction. It internally stores the identifier in its functional context and pl coins in its linear context, and offers along the linear account type.

$$(r : \text{id}) \ ; \ (M : \text{plcoins}) \vdash \mathit{balance} :: (\mathit{acc} : \text{account})$$

Finally, the contract process stores the allowances as a list of triples storing the owner's and sender's address and allowance value, typed as id × id × int. Thus, the *plcontract* process stores the allowance in the functional context, and the list of accounts in its linear context and offers along the erc20token type introduced earlier.

$$(\mathit{allow} : \text{list}_{\text{id}\times\text{id}\times\text{int}}) \ ; \ (\mathit{accs} : \text{list}_{\text{account}}) \vdash \mathit{plcontract} :: (\mathit{st} : \text{erc20token})$$

As an illustration, we show the part of the implementation for initiating a transfer.

```
1:  st ← plcontract allow ← accs =
2:    lt ← accept st ;                      % accept a client acquire request
3:    case lt ...                           % switch on label on lt
4:      | transfer ⇒ s ← recv lt ;          % receive sender's identifier s : id
5:                   r ← recv lt ;          % receive receiver's identifier r : id
6:                   v ← recv lt ;          % receive transfer value v : int
7:                   st ← detach lt ;       % detach from client
8:                   ...                    % extract sender and receiver's account ...
9:                   ...                    % and store in sa and ra resp.
10:                  sa.subtract ;          % subtract pl coins corresponding to ...
11:                  sa.v ;                 % v from account channel sa
12:                  m ← recv sa            % receive transfer amount m
13:                  ra.add ;               % add m : plcoins to ...
14:                  send ra m ;            % account channel ra
15:                  st ← plcontract allow ← accs
```

The contract first receives the sender and receiver's identifiers (lines 4 and 5) and the transfer value v. The contract then detaches from the client (line 7). We skip the code of extracting the sender's and receiver's account from the list *accs* and store them in *sa* and *ra* of type account, respectively. The contract then subtracts the pl coins from account *sa* (lines 10 and 11) and receives and stores them in m (line 12). This balance is then added to *ra*'s account (lines 13 and 14). An important point here is that Nomos enforces linearity of the transfer transaction. Since m : plcoins is typed as a linear asset, it cannot be discarded or modified. The amount deducted from sender must be transferred to the receiver (since no minting is involved here).

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 [HKG 2017].

The typographical error in the contract came about when updating the receiver's balance during a transfer. Instead of writing `balance += value`, the programmer mistakenly wrote `balance =+ value` (semantically meaning `balance = value`). Moreover, while testing this error was missed, because the first transfer always succeeds (since the two statements are semantically equivalent when `balance = 0`). 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 [Luu et al. 2016] 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

```
type 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.

```
type ballot = /\ <{22}| +{
  open : id -> +{ vote : id -> |{0}> \/ ballot,
                  novote : |{14}> \/ ballot },
  closed : id ^ |{19}> \/ 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 (voting in Table 2) stores a counter for each candidate that is updated after each vote is cast; the second (voting-aa in Table 2) does not use a counter but stores potential inside the vote list that is consumed for counting the votes at the end. This stored potential is provided by the voter to amortize the cost of counting.

Escrow. A contract can act as a reliable third party for custody of a bond that takes effect once both the buyer and the seller approve.

```
type escrow = /\ <{7}| &{
  approve : id -> |{0}> \/ escrow,
  cancel : id -> |{0}> \/ escrow,
  deposit : id -> bond -o[R] |{4}> \/ escrow,
  withdraw : id -> bond *[R] |{0}> \/ escrow }
```

This session type describes the implementation of an escrow, allowing the seller to deposit the bond, the buyer to withdraw the bond, and both the buyer and seller to approve or cancel the whole transaction. The withdrawal succeeds only after the bond has been deposited, and both the buyer and seller approve it.

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 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 implemented 8 case studies in Nomos. We have already discussed auction (Section 2), ERC 20, puzzle, and voting. 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.
- An insurance contract that processes flight delay insurance claims after verifying them with a trusted third party. This contract involves inter-contract communication since the insurance and the third-party verifier are implemented as separate contracts.

Table 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 of the contract. 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. 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 using $*$ for potential annotations meant we could remain unaware of the exact gas cost of operations.

9 BLOCKCHAIN INTEGRATION

Although Nomos has been designed to be applicable for implementing general digital contracts, the standard semantics needs some adaptation for a contract to be run on a blockchain. To integrate with a blockchain, we need a mechanism to *(i)* represent the contracts and their addresses in the current blockchain state, *(ii)* create and send transactions to the appropriate addresses, and most importantly, *(iii)* construct the global distributed ledger, which stores the history of all transactions. This section addresses these challenges and also highlights the main limitation of the language.

Nomos on a Blockchain. To describe a possible blockchain implementation of Nomos, we assume a blockchain like Ethereum that contains a set of Nomos contracts $C_1, \ldots, C_n$ together with their type information $\Psi^i \; ; \; \Gamma^i \; ; \; \Delta_{\mathsf{R}}^i \vDash^{q_i} C_i :: (x_{\mathsf{S}}^i : A_{\mathsf{S}}^i)$. The functional contexts Ψ^i type the contract data, while the shared contexts Γ^i type the shared contracts that C_i refer to, and the linear contexts Δ_{R}^i type the contract's linear assets. We allow contracts to carry potential given by the annotations q_i and the potential defined by the annotations in Ψ^i and Δ_{R}^i. This potential is useful to amortize gas cost over different transactions. If this behavior is not desired then one can require $q_i = 0$ for every i. Together, these contracts define the blockchain state. The channel name x_{S}^i of a contract is its address and has to be globally unique. We assume the existence of a deterministic mechanism that produces fresh names.

To perform a transaction with a contract, an external user submits a script that is well-typed with respect to the existing contracts using the judgment

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

Here, $\Gamma \subseteq x_{\mathsf{S}}^1 : A_{\mathsf{S}}^1, \ldots, x_{\mathsf{S}}^n : A_{\mathsf{S}}^n$ stores references to the Nomos contracts accessible by the transaction. Ψ stores the functional part of the script, and since the script cannot refer to linear data, its linear context is empty. Additionally, we mandate that the transaction offers along a channel of type $\mathbf{1}$, and that it terminates by sending a close message on its offered channel. For instance, the transaction Q must end with the operation (close x_{T}). This ensures that transactions are sequentialized and executed in the order they are queued (explained below).

A transaction script is connected to the blockchain state using a server process. This process, named bc−server stores the entire transaction history and offers along channel bc : tx_interface where the transaction code is received and relayed to the blockchain state. It is defined as follows.

```
1:  type tx_code = {1}       type tx_queue = list tx_code
2: stype tx_interface = tx_code → tx_interface
3: (txns : tx_queue) ; · ; · ⊨⁰ bc−server :: (bc : tx_interface)
4:    bc ← bc−server txns =
5:       tx ← recv bc ; x_T ← tx ; wait x_T ;
6:       bc ← bc−server (tx :: txns)
```

The transaction script is packaged as a value of the contextual monadic type introduced in Section 5. For instance, the transaction Q is packaged as $\{x_{\mathsf{T}} \leftarrow Q\} : \{\mathbf{1}\}$ = tx_code. The bc−server process receives this code, spawns a process corresponding to it and waits for the transaction to terminate (line 5). Note that the transaction is required to terminate with a (close x_{T}) message which matches with the (wait x_{T}) being executed by the server, ensuring the execution order of the transactions. Finally, the latest transaction is added to the queue of transactions $txns$: type tx_queue = list tx_code, and the bc−server process recurses.

A transaction can either update the state of existing contracts, or create new ones. In the former case, it *acquires* the contracts it wishes to interact with, followed by an update in the contracts' internal state and *releases* them. Since the contract types are equi-synchronizing, they remain unchanged at the end of transaction execution. This ensures that the subsequent transactions can access the same contracts at the same type. In the future we plan to allow *sub-synchronizing* types that enable a client to release a contract channel not at the same type, but a *subtype*. The subtype can then describe the phase of the contract. For instance, the ended phase of auction contract will be a subtype of the running phase. In the latter case, new contracts are added to the blockchain state, making them visible in the type of the configuration for subsequent transactions to access. Thus, in either case, the blockchain state remains well-formed between transactions. A successful execution of a transaction will lead to the bc−server process recursing and accepting further transactions.

Concurrent execution of transactions is missing from blockchain systems today [Herlihy 2019]. To reconstruct the blockchain state, each miner must re-execute every transaction sequentially; simply executing them in parallel is unsafe when contracts depend on each other. However, Nomos naturally has a concurrent semantics, and we can support concurrent transactions with a slight modification to the tx_interface type. One caveat is that we need to ensure deterministic execution of a transaction. The only source of non-determinism in the Nomos semantics is the *acquire-accept* pair. A contract executing an accept can attach with any process that tries to acquire it. One approach to resolve this non-determinism is *record-and-replay* [Lidbury and Donaldson 2019; Ronsse and De Bosschere 1999]. The miner records the order in which the contracts are acquired in the ledger, which is then replayed by others to compute the current blockchain state. Another promising approach is *speculation* [Dickerson et al. 2017] where transactions are executed in parallel and their read and write sets are tracked. If there is a conflict in these sets, then they are sequentialized and this schedule is repeated by validators. This speculative technique is known to provide speed-ups to the overall throughput of the blockchain system [Saraph and Herlihy 2019].

When selecting a request, a miner first creates a configuration, and then type checks the transaction script Q against its submitted type information and the existing types of the contracts $C_1, \ldots, C_n$ and the server process. If type checking were too costly here, that can lead to yet another source of denial-of-service attacks. In Nomos however, since the type of transaction script is provided by the programmer, this form of bi-directional type checking is linear time in the size of the script. The gas cost of the transaction is statically bounded by the potential given by q and Ψ. If we allow amortization then the potential in the contracts C_i's is also available to cover the gas cost. This internal potential is not available to the user but can only be accessed according to the protocol that is given in the contract session type.

Miner's Transaction Fee. Mining rewards in blockchains like Ethereum are realized by special transactions that transfer coins to the miner at the beginning of a block. In Nomos, such a transaction could, for example, be represented by an interaction with a special *mining reward contract* that sends linear coins to every client who requests them. Like in Ethereum, a block with transactions is only valid if only the first transaction interacts with the reward contract. This can be ensured by the miner with a dynamic check or statically by removing the reward contract from the list of available contracts before executing user transactions.

Deadlocks. The only language specific reason a transaction can fail is a deadlock in the transaction code. Our progress theorem accounts for the possibility of deadlocks. Deadlocks may arise due to cyclic interdependencies on the contracts that a transaction attempts to acquire. While it is of course desirable to rule out deadlocks, we felt that this is orthogonal to the design of Nomos. Any extensions for shared session types that prevent deadlocks (e.g., [Balzer et al. 2019]) will be readily transferable to our setting. Another possibility is to employ dynamic deadlock detection [Chandy et al. 1983; Mitchell and Merritt 1984] and abort the transaction if a deadlock is detected.

10 RELATED WORK

We classify the related work into 3 categories - i) new programming languages for smart contracts, ii) static analysis techniques for existing languages and bytecode, and iii) session-typed and type-based resource analysis systems technically related to Nomos.

Smart Contract Languages. Existing smart contracts on Ethereum are predominantly implemented in Solidity [Auc 2016], a statically typed object-oriented language influenced by Python and Javascript. However, the language provides no information about the resource usage of a contract. Languages like Vyper [Vyp 2018] 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 [Bam 2018], on the other hand, makes state transitions explicit and avoids re-entrance by design. In contrast to our work, none of these languages use linear type systems to track assets stored in a contract.

Domain specific languages have also been designed for other blockchains apart from Ethereum. Typecoin [Crary and Sullivan 2015] uses affine logic to solve the peer-to-peer affine commitment problem using a generalization of Bitcoin where transactions deal in types rather than numbers. Although Typecoin does not provide a mechanism for expressing protocols, it also uses a linear type system to prevent resources from being discarded or duplicated. Rholang [Rho 2018] is formally modeled by the ρ-calculus, a reflective higher-order extension of the π-calculus. Michelson [Mic 2018] is a purely functional stack-based language that has no side effects. Scilla [Sergey et al. 2019] is an intermediate-level language where contracts are structured as communicating automata providing a continuation-passing style computational model to the language semantics. However, none of these languages describe and enforce communication protocols statically.

Static Analysis. Analysis of smart contracts has received substantial attention recently due to their security vulnerabilities that can be exploited by malicious users. KEVM [Hildenbrandt et al. 2018] creates a program verifier based on reachability logic that given an EVM program and specification, tries to automatically prove the corresponding reachability theorems. However, the verifier requires significant manual intervention, both in specification and proof construction. Oyente [Luu et al. 2016] is a symbolic execution tool that checks for 4 kinds of security bugs in smart contracts, transaction-order dependence, timestamp dependence, mishandled exceptions and re-entrancy vulnerabilities. MadMax [Grech et al. 2018] automatically detects gas-focused vulnerabilities with high confidence. The analysis is based on a decompiler that extracts control and data flow information from EVM bytecode, and a logic-based analysis specification that produces a high-level program model. Bhargavan et al. [2016] translate Ethereum contracts to F* to prove runtime safety and functional correctness, although they do not support all syntactic features. VeriSol [Lahiri et al. 2018] is a highly-automated formal verifier for Solidity that can produce proofs as well as counterexamples and proves semantic conformance of smart contracts against a state machine model with access-control policy. However, in contrast to Nomos, where guarantees are proved by a soundness proof of the type system, static analysis techniques often do not explore all program paths, can report false positives that need to be manually filtered, and miss bugs due to timeouts and other sources of incompleteness.

Session types and Resource analysis. Session types were introduced by Honda [Honda 1993] as a typed formalism for inter-process dyadic interaction. They have been integrated into a functional language in prior work [Toninho et al. 2013]. However, this integration does not account for resource usage or sharing. Sharing in session types has also been explored in prior work [Balzer and Pfenning 2017], but with the strong restriction that shared processes cannot rely on linear resources that we lift in Nomos. Shared session types were also never integrated with a functional layer or tracked for resource usage. While we consider binary session types that express local interactions, global protocols can be expressed using multi-party session types [Honda et al. 2008; Scalas and Yoshida 2019]. Automatic amortized resource analysis (AARA) has been introduced as a type system to derive linear [Hofmann and Jost 2003] and polynomial bounds [Hoffmann et al. 2017] for functional programming languages. Resource usage has also previously been explored separately for the purely linear process layer [Das et al. 2018], but were never combined with shared session types or integrated with the functional layer.

11 CONCLUSION

We have described the programming language Nomos, its type-theoretic foundation, a prototype implementation and evaluated its feasibility on several real world smart contract applications. Nomos builds on linear logic, shared session types, and automatic amortized resource analysis to address the challenges that programmers are faced with when implementing digital contracts. Our main contributions are the design and implementation of Nomos' multi-layered resource-aware type system and its type soundness proof.

In future work, we plan to explore refinement session types for expressing and verifying functional correctness of contracts against their specifications and to target open questions regarding a blockchain integration. These include the exact cost model, fluctuation of gas prices, and potential compilation to a lower-level language. Since Nomos has a concurrent semantics, we also plan to support parallel execution of transactions using speculation techniques [Saraph and Herlihy 2019].

A OVERVIEW

This appendix supplements the tech report "Resource-Aware Session Types for Digital Contracts". The main contributions of the appendix are as follows.

- Appendix B presents the Nomos code for standard smart contract applications.
- Appendix C presents the type grammar.
- Appendix D presents the process typing rules, concerning the judgment $\Psi \; ; \; \Gamma \; ; \; \Delta \overset{q}{\vdash} P :: (x_m : A)$. This judgment types a process in state P providing service of type A along channel x at mode m. Moreover, the process uses functional variables from Ψ, shared channels from Γ and linear channels from Δ. Finally, the process stores potential q.
- Appendix E presents the rules of the operational cost semantics. These discuss the behavior of the semantic objects $\text{proc}(c_m, w, P)$ and $\text{msg}(c_m, w, N)$ defining a process P (or message N) offering along channel c at mode m which has performed work w so far.
- Appendix F presents the rules corresponding to configuration typing and other helper judgments. The configuration typing judgment $\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta)$ describes a well-typed configuration Ω which offers shared channels in Γ and linear channels in Δ.
- Appendix G is the main contribution of the supplementary material. It presents and proves the main theorem of type safety of our language. This is split into a type preservation and a progress theorem. The appendix also proves the lemmas necessary for the type safety theorems.

B IMPLEMENTATION OF SMART CONTRACT APPLICATIONS IN NOMOS

B.1 Auction

```
type money = &{ value : <{2}| int ^ money,
                coins : <{0}| lcoin }
type lcoin = 1
```

```
proc asset emp : . |{1}- ($l[R] : lcoin) =
  {
    work ;
    close $l[R]
  }
proc asset empty_wallet : . |{3}- ($m[R] : money) =
  {
    $l[R] <- emp <- ;
    work ;
    let n = (tick ; 0) ;
    $m[R] <- wallet <- n $l[R]
  }
proc asset wallet : (n : int), ($l[R] : lcoin) |- ($m[R] : money) =
  {
    case $m[R] ( value => get $m[R] {2};
                        work ;
                        send $m[R] ((tick ; n)) ;
                        $m[R] <- wallet <- n $l[R]
               | coins => get $m[R] {0};
                        $m[R] <- $l[R] )
  }
type dictionary = &{ add : <{5}| int -> money -o[R] dictionary,
                     delete : <{6}| int -> money *[R] dictionary,
                     check : <{4}| int -> bool ^ dictionary,
                     size : <{2}| int ^ dictionary }
proc asset dummy : (n : int) |- ($d[R] : dictionary) =
  {
    case $d[R] ( add => get $d[R] {5};
                      key = recv $d[R] ;
                      work ;
                      $v[R] <- recv $d[R] ;
                      $v[R].coins ;
                      pay $v[R] {0};
                      work ;
                      wait $v[R] ;
                      let n = (tick ; (tick ; n) + (tick ; 1)) ;
                      $d[R] <- dummy <- n
               | delete => get $d[R] {6};
                         key = recv $d[R] ;
                         $v[R] <- empty_wallet <- ;
                         send $d[R] $v[R] ;
                         let n = (tick ; (tick ; n) - (tick ; 1)) ;
                         $d[R] <- dummy <- n
               | check => get $d[R] {4};
                        key = recv $d[R] ;
                        if (tick ; (tick ; key) > (tick ; 0))
                        then
                          send $d[R] ((tick ; true)) ;
                          $d[R] <- dummy <- n
                        else
                          send $d[R] ((tick ; false)) ;
                          $d[R] <- dummy <- n
```

```
             | size => get $d[R] {2};
                      work ;
                      send $d[R] ((tick ; n)) ;
                      $d[R] <- dummy <- n )
  }
type lot = 1
proc asset addbid : (r : int), ($m[R] : money), ($bs[R] : dictionary)
                           |{8}- ($newbs[R] : dictionary) =
  {
    work ;
    $bs[R].add ;
    pay $bs[R] {5};
    work ;
    send $bs[R] ((tick ; r)) ;
    send $bs[R] $m[R] ;
    $newbs[R] <- $bs[R]
  }
type auction = /\ <{22}|
  +{ running : &{ bid : int -> money -o[R] |{0}> \/ auction,
                cancel : |{21}> \/ auction },
     ended : &{ collect : int -> +{ won : lot *[R] |{0}> \/ auction,
                                  lost : money *[R] |{7}> \/ auction },
              cancel : |{21}> \/ auction } }
proc contract run : (T : int), (w : int), (v : int),
                  ($b[R] : dictionary), ($l[R] : lot)
                                         |- (#sa[S] : auction) =
  {
    $la[L] <- accept #sa[S] ;
    get $la[L] {22};
    work ;
    $la[L].running ;
    case $la[L] ( bid => r = recv $la[L] ;
                      work ;
                      $m[R] <- recv $la[L] ;
                      pay $la[L] {0};
                      #sa[S] <- detach $la[L] ;
                      $m[R].value ;
                      pay $m[R] {2};
                      work ;
                      bv = recv $m[R] ;
                      $newb[R] <- addbid <- r $m[R] $b[R] ;
                      if (tick ; (tick ; bv) > (tick ; v))
                      then
                        #sa[S] <- check <- T r bv $newb[R] $l[R]
                      else
                        #sa[S] <- check <- T w v $newb[R] $l[R]
              | cancel => pay $la[L] {21};
                         #sa[S] <- detach $la[L] ;
                         #sa[S] <- run <- T w v $b[R] $l[R] )
  }
proc contract check : (T : int), (w : int), (v : int),
                    ($b[R] : dictionary), ($l[R] : lot)
```

```
                                        |{6}- (#sa[S] : auction) =
  {
    work ;
    $b[R].size ;
    pay $b[R] {2};
    n = recv $b[R] ;
    if (tick ; (tick ; n) = (tick ; T))
    then
      #sa[S] <- end_lot <- T w $b[R] $l[R]
    else
      #sa[S] <- run <- T w v $b[R] $l[R]
  }
proc asset removebid : (r : int), ($bs[R] : dictionary)
                          |{10}- ($newbs[R] : money *[R] dictionary) =
  {
    work ;
    $bs[R].delete ;
    pay $bs[R] {6};
    work ;
    send $bs[R] ((tick ; r)) ;
    work ;
    $m[R] <- recv $bs[R] ;
    send $newbs[R] $m[R] ;
    $newbs[R] <- $bs[R]
  }
proc contract end_lot : (T : int), (w : int),
                      ($b[R] : dictionary), ($l[R] : lot)
                                          |- (#sa[S] : auction) =
  {
    $la[L] <- accept #sa[S] ;
    get $la[L] {22};
    work ;
    $la[L].ended ;
    case $la[L] ( collect => r = recv $la[L] ;
                           if (tick ; (tick ; w) = (tick ; r))
                           then
                             $la[L].won ;
                             send $la[L] $l[R] ;
                             pay $la[L] {0};
                             #sa[S] <- detach $la[L] ;
                             #sa[S] <- end_nolot <- T w $b[R]
                           else
                             $la[L].lost ;
                             $newb[R] <- removebid <- r $b[R] ;
                             work ;
                             $m[R] <- recv $newb[R] ;
                             send $la[L] $m[R] ;
                             pay $la[L] {7};
                             #sa[S] <- detach $la[L] ;
                             #sa[S] <- end_lot <- T w $newb[R] $l[R]
                | cancel => pay $la[L] {21};
                          #sa[S] <- detach $la[L] ;
```

```
                        #sa[S] <- end_lot <- T w $b[R] $l[R] )
  }
proc contract end_nolot : (T : int), (w : int), ($b[R] : dictionary)
                                          |{18}- (#sa[S] : auction) =
  {
    $la[L] <- accept #sa[S] ;
    get $la[L] {22};
    work ;
    $la[L].ended ;
    case $la[L] ( collect => r = recv $la[L] ;
                             $la[L].lost ;
                             $newb[R] <- removebid <- r $b[R] ;
                             work ;
                             $m[R] <- recv $newb[R] ;
                             send $la[L] $m[R] ;
                             pay $la[L] {7};
                             #sa[S] <- detach $la[L] ;
                             work {3};
                             #sa[S] <- end_nolot <- T w $newb[R]
               | cancel => pay $la[L] {21};
                           #sa[S] <- detach $la[L] ;
                           work {0};
                           #sa[S] <- end_nolot <- T w $b[R] )
  }
TC time: 2.0809173584
Inference time: 9.94420051575
# Vars = 229
# Constraints = 730
% compilation successful!
% runtime successful!
```

B.2 Bank Account

```
type money = &{ value : <{2}| int ^ money,
              coins : <{0}| lcoin,
              check_pwd : <{4}| int -> bool ^ money }
type lcoin = 1
proc asset emp : . |{1}- ($l[R] : lcoin) =
  {
    work ;
    close $l[R]
  }
proc asset empty_wallet : (pwd : int) |{3}- ($m[R] : money) =
  {
    $l[R] <- emp <- ;
    work ;
    let n = (tick ; 0) ;
    $m[R] <- wallet <- pwd n $l[R]
  }
proc asset wallet : (pwd : int), (n : int), ($l[R] : lcoin) |- ($m[R] : money) =
  {
    case $m[R] ( value => get $m[R] {2};
                        work ;
```

```
                      send $m[R] ((tick ; n)) ;
                      $m[R] <- wallet <- pwd n $l[R]
             | coins => get $m[R] {0};
                      $m[R] <- $l[R]
             | check_pwd => get $m[R] {4};
                          p = recv $m[R] ;
                          if (tick ; (tick ; p) = (tick ; pwd))
                          then
                            send $m[R] ((tick ; true)) ;
                            $m[R] <- wallet <- pwd n $l[R]
                          else
                            send $m[R] ((tick ; false)) ;
                            $m[R] <- wallet <- pwd n $l[R] )
  }
type dictionary = &{ add : <{5}| int -> money -o[R] dictionary,
                     delete : <{6}| int -> money *[R] dictionary,
                     check : <{5}| int -> int -> bool ^ dictionary,
                     size : <{2}| int ^ dictionary }
proc asset dummy : (n : int) |- ($d[R] : dictionary) =
  {
    case $d[R] ( add => get $d[R] {5};
                      key = recv $d[R] ;
                      work ;
                      $v[R] <- recv $d[R] ;
                      $v[R].coins ;
                      pay $v[R] {0};
                      work ;
                      wait $v[R] ;
                      let n = (tick ; (tick ; n) + (tick ; 1)) ;
                      $d[R] <- dummy <- n
             | delete => get $d[R] {6};
                         key = recv $d[R] ;
                         $v[R] <- empty_wallet <- key ;
                         send $d[R] $v[R] ;
                         let n = (tick ; (tick ; n) - (tick ; 1)) ;
                         $d[R] <- dummy <- n
             | check => get $d[R] {5};
                        key = recv $d[R] ;
                        work ;
                        pwd = recv $d[R] ;
                        if (tick ; (tick ; pwd) > (tick ; 0))
                        then
                          send $d[R] ((tick ; true)) ;
                          $d[R] <- dummy <- n
                        else
                          send $d[R] ((tick ; false)) ;
                          $d[R] <- dummy <- n
             | size => get $d[R] {2};
                       work ;
                       send $d[R] ((tick ; n)) ;
                       $d[R] <- dummy <- n )
  }
```

```
type account = /\ <{29}|
   &{ signup : int -> int -> |{19}> \/ account,
      login : int -> int ->
         +{ failure : |{19}> \/ account,
            success : &{ deposit : money -o[R] |{11}> \/ account,
                        balance : int ^ |{0}> \/ account,
                        withdraw : int -> money *[R] |{9}> \/ account } } }
proc contract bank : ($accts[R] : dictionary) |- (#sa[S] : account) =
  {
    $la[L] <- accept #sa[S] ;
    get $la[L] {29};
    case $la[L] ( signup => id = recv $la[L] ;
                         work ;
                         pwd = recv $la[L] ;
                         $m[R] <- empty_wallet <- pwd ;
                         $accts[R].add ;
                         pay $accts[R] {5};
                         send $accts[R] ((tick ; id)) ;
                         send $accts[R] $m[R] ;
                         pay $la[L] {19};
                         #sa[S] <- detach $la[L] ;
                         #sa[S] <- bank <- $accts[R]
                | login => id = recv $la[L] ;
                         work ;
                         pwd = recv $la[L] ;
                         $accts[R].check ;
                         pay $accts[R] {5};
                         send $accts[R] ((tick ; id)) ;
                         send $accts[R] ((tick ; pwd)) ;
                         work ;
                         r = recv $accts[R] ;
                         if (tick ; r)
                         then
                           $la[L].success ;
                           work ;
                           case $la[L]
                            ( deposit => work ;
                                        $m[R] <- recv $la[L] ;
                                        $accts[R].add ;
                                        pay $accts[R] {5};
                                        send $accts[R] ((tick ; id)) ;
                                        send $accts[R] $m[R] ;
                                        pay $la[L] {11};
                                        #sa[S] <- detach $la[L] ;
                                        #sa[S] <- bank <- $accts[R]
                            | balance => $accts[R].delete ;
                                        pay $accts[R] {6};
                                        send $accts[R] ((tick ; id)) ;
                                        work ;
                                        $m[R] <- recv $accts[R] ;
                                        $m[R].value ;
                                        pay $m[R] {2};
```

```
                                    work ;
                                    val = recv $m[R] ;
                                    send $la[L] ((tick ; val)) ;
                                    $accts[R].add ;
                                    pay $accts[R] {5};
                                    send $accts[R] ((tick ; id)) ;
                                    send $accts[R] $m[R] ;
                                    pay $la[L] {0};
                                    #sa[S] <- detach $la[L] ;
                                    #sa[S] <- bank <- $accts[R]
                        | withdraw => $accts[R].delete ;
                                     pay $accts[R] {6};
                                     send $accts[R] ((tick ; id)) ;
                                     work ;
                                     v = recv $la[L] ;
                                     work ;
                                     $m[R] <- recv $accts[R] ;
                                     send $la[L] $m[R] ;
                                     pay $la[L] {9};
                                     #sa[S] <- detach $la[L] ;
                                     #sa[S] <- bank <- $accts[R] )
                       else
                         $la[L].failure ;
                         pay $la[L] {19};
                         #sa[S] <- detach $la[L] ;
                         #sa[S] <- bank <- $accts[R] )
  }
TC time: 0.648975372314
Inference time: 4.99391555786
# Vars = 173
# Constraints = 561
% compilation successful!
% runtime successful!
```

B.3 ERC-20 Token

```
type money = &{ add : <{8}| money -o[R] money,
               subtract : <{6}| int -> +{ sufficient : money *[R] money,
                                          insufficient : money },
               value : <{2}| int ^ money,
               coins : <| 1 }
proc asset wallet : (n : int) |- ($m[R] : money) =
  {
    case $m[R] ( add => get $m[R] {8};
                      $m1[R] <- recv $m[R] ;
                      work ;
                      $m1[R].value ;
                      pay $m1[R] {2};
                      n1 = recv $m1[R] ;
                      $m1[R].coins ;
                      pay $m1[R] ;
                      work ;
                      wait $m1[R] ;
```

```
                    let n = (tick ; (tick ; n) + (tick ; n1)) ;
                    $m[R] <- wallet <- n
            | subtract => get $m[R] {6};
                        n1 = recv $m[R] ;
                        if (tick ; (tick ; n) > (tick ; n1))
                        then
                          $m[R].sufficient ;
                          $m1[R] <- wallet <- n1 ;
                          send $m[R] $m1[R] ;
                          let n = (tick ; (tick ; n) - (tick ; n1)) ;
                          work {0};
                          $m[R] <- wallet <- n
                        else
                          $m[R].insufficient ;
                          work {3};
                          $m[R] <- wallet <- n
            | value => get $m[R] {2};
                     work ;
                     send $m[R] ((tick ; n)) ;
                     $m[R] <- wallet <- n
            | coins => get $m[R] ;
                     work ;
                     close $m[R] )
  }
type erc20token = /\ <{11}|
   &{ totalSupply : int ^ |{9}> \/ erc20token,
      balanceOf : int -> int ^ |{8}> \/ erc20token,
      transfer : int -> int -> int -> |{0}> \/ erc20token,
      transferFrom : int -> int -> int -> |{0}> \/ erc20token,
      approve : int -> int -> int -> |{6}> \/ erc20token,
      allowance : int -> int -> int ^ |{6}> \/ erc20token }
type balance_dict = &{ get_balance : int -> int ^ balance_dict,
                     transfer : int -> int -> int -> balance_dict }
type allowance_dict = &{ get_allowance : int -> int -> int ^ allowance_dict,
                       set_allowance : int -> int -> int -> allowance_dict }
proc contract erc20contract : ($allows[R] : allowance_dict),
                           ($bals[R] : balance_dict), (N : int)
                                     |- (#se[S] : erc20token) =
  {
    $le[L] <- accept #se[S] ;
    get $le[L] {11};
    case $le[L] ( totalSupply => work ;
                            send $le[L] ((tick ; N)) ;
                            pay $le[L] {9};
                            #se[S] <- detach $le[L] ;
                            #se[S] <- erc20contract <- $allows[R] $bals[R] N
              | balanceOf => addr = recv $le[L] ;
                          $bals[R].get_balance ;
                          send $bals[R] ((tick ; addr)) ;
                          work ;
                          val = recv $bals[R] ;
                          send $le[L] ((tick ; val)) ;
```

```
                pay $le[L] {8};
                #se[S] <- detach $le[L] ;
                #se[S] <- erc20contract <- $allows[R] $bals[R] N
  | transfer => from = recv $le[L] ;
               work ;
               to = recv $le[L] ;
               work ;
               amt = recv $le[L] ;
               $allows[R].get_allowance ;
               send $allows[R] ((tick ; from)) ;
               send $allows[R] ((tick ; to)) ;
               work ;
               allowance = recv $allows[R] ;
               if (tick ; (tick ; amt) > (tick ; allowance))
               then
                 pay $le[L] {0};
                 #se[S] <- detach $le[L] ;
                 work {3};
                 #se[S] <- erc20contract <- $allows[R] $bals[R] N
               else
                 $bals[R].transfer ;
                 send $bals[R] ((tick ; from)) ;
                 send $bals[R] ((tick ; to)) ;
                 send $bals[R] ((tick ; amt)) ;
                 pay $le[L] {0};
                 #se[S] <- detach $le[L] ;
                 work {0};
                 #se[S] <- erc20contract <- $allows[R] $bals[R] N
| transferFrom => from = recv $le[L] ;
                 work ;
                 to = recv $le[L] ;
                 work ;
                 amt = recv $le[L] ;
                 $allows[R].get_allowance ;
                 send $allows[R] ((tick ; from)) ;
                 send $allows[R] ((tick ; to)) ;
                 work ;
                 allowance = recv $allows[R] ;
                 if (tick ; (tick ; amt) > (tick ; allowance))
                 then
                   pay $le[L] {0};
                   #se[S] <- detach $le[L] ;
                   work {3};
                   #se[S] <- erc20contract <- $allows[R] $bals[R] N
                 else
                   $bals[R].transfer ;
                   send $bals[R] ((tick ; from)) ;
                   send $bals[R] ((tick ; to)) ;
                   send $bals[R] ((tick ; amt)) ;
                   pay $le[L] {0};
                   #se[S] <- detach $le[L] ;
                   work {0};
```

```
                              #se[S] <- erc20contract <- $allows[R] $bals[R] N
               | approve => from = recv $le[L] ;
                           work ;
                           to = recv $le[L] ;
                           work ;
                           allowance = recv $le[L] ;
                           $allows[R].set_allowance ;
                           send $allows[R] ((tick ; from)) ;
                           send $allows[R] ((tick ; to)) ;
                           send $allows[R] ((tick ; allowance)) ;
                           pay $le[L] {6};
                           #se[S] <- detach $le[L] ;
                           #se[S] <- erc20contract <- $allows[R] $bals[R] N
               | allowance => from = recv $le[L] ;
                             work ;
                             to = recv $le[L] ;
                             $allows[R].get_allowance ;
                             send $allows[R] ((tick ; from)) ;
                             send $allows[R] ((tick ; to)) ;
                             work ;
                             allowance = recv $allows[R] ;
                             send $le[L] ((tick ; allowance)) ;
                             pay $le[L] {6};
                             #se[S] <- detach $le[L] ;
                             #se[S] <- erc20contract <- $allows[R] $bals[R] N )
  }
TC time: 1.89590454102
Inference time: 4.7709941864
# Vars = 161
# Constraints = 561
% compilation successful!
% runtime successful!
```

B.4 Escrow

```
type escrow = /\ <{7}| &{ approve : int -> |{0}> \/ escrow,
                       cancel : int -> |{0}> \/ escrow,
                       deposit : int -> bond -o[R] |{4}> \/ escrow,
                       withdraw : int -> bond *[R] |{0}> \/ escrow }
type bond = 1
proc asset emp : . |{1}- ($l[R] : bond) =
  {
    work ;
    close $l[R]
  }
proc contract escrow_con : (buyer : int), (seller : int),
                        (buyerOk : bool), (sellerOk : bool),
                        ($l[R] : bond)
                                    |- (#se[S] : escrow) =
  {
    $le[L] <- accept #se[S] ;
    get $le[L] {7};
    case $le[L]
```

```
 ( approve =>
    r = recv $le[L] ;
    if (tick ; (tick ; r) = (tick ; buyer))
    then
      let buyerOk = (tick ; true) ;
      pay $le[L] {0};
      #se[S] <- detach $le[L] ;
      work {3};
      #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
    else
      if (tick ; (tick ; r) = (tick ; seller))
      then
        let sellerOk = (tick ; true) ;
        pay $le[L] {0};
        #se[S] <- detach $le[L] ;
        work {0};
        #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
      else
        pay $le[L] {0};
        #se[S] <- detach $le[L] ;
        work ;
        #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
| cancel =>
    r = recv $le[L] ;
    if (tick ; (tick ; r) = (tick ; buyer))
    then
      let buyerOk = (tick ; false) ;
      pay $le[L] {0};
      #se[S] <- detach $le[L] ;
      work {3};
      #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
    else
      if (tick ; (tick ; r) = (tick ; seller))
      then
        let sellerOk = (tick ; false) ;
        pay $le[L] {0};
        #se[S] <- detach $le[L] ;
        work {0};
        #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
      else
        pay $le[L] {0};
        #se[S] <- detach $le[L] ;
        work ;
        #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
| deposit =>
    r = recv $le[L] ;
    work ;
    $m[R] <- recv $le[L] ;
    let seller = (tick ; r) ;
    pay $le[L] {4};
    #se[S] <- detach $le[L] ;
    work ;
```

```
        wait $m[R] ;
        work {0};
        #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
     | withdraw =>
        r = recv $le[L] ;
        if (tick ; (tick ; r) = (tick ; buyer))
        then
          send $le[L] $l[R] ;
          $l[R] <- emp <- ;
          pay $le[L] {0};
          #se[S] <- detach $le[L] ;
          work {3};
          #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R]
        else
          $m[R] <- emp <- ;
          send $le[L] $m[R] ;
          pay $le[L] {0};
          #se[S] <- detach $le[L] ;
          work {3};
          #se[S] <- escrow_con <- buyer seller buyerOk sellerOk $l[R] )
  }
TC time: 1.9428730011
Inference time: 5.47099113464
# Vars = 95
# Constraints = 321
% compilation successful!
% runtime successful!
```

B.5 Insurance

```
type insurance = /\ <{6}|
   &{ submit : int -> +{ success : money *[R] |{0}> \/ insurance,
      failure : |> \/ insurance } }
type verifier = /\ <{3}| &{ verify : int -> +{ valid : |{0}> \/ verifier,
                                             invalid : |{0}> \/ verifier } }
proc contract verify : . |- (#sv[S] : verifier) =
  {
    $lv[L] <- accept #sv[S] ;
    get $lv[L] {3};
    case $lv[L] ( verify => claim = recv $lv[L] ;
                         if (tick ; (tick ; claim) > (tick ; 0))
                         then
                           $lv[L].valid ;
                           pay $lv[L] {0};
                           #sv[S] <- detach $lv[L] ;
                           #sv[S] <- verify <-
                         else
                           $lv[L].invalid ;
                           pay $lv[L] {0};
                           #sv[S] <- detach $lv[L] ;
                           #sv[S] <- verify <- )
  }
type money = &{ subtract : money *[R] money }
```

```
proc contract insurer : (#sv[S] : verifier), ($m[R] : money)
                                      |- (#si[S] : insurance) =
  {
    $li[L] <- accept #si[S] ;
    get $li[L] {6};
    case $li[L] ( submit => claim = recv $li[L] ;
                       $lv[L] <- acquire #sv[S] ;
                       pay $lv[L] {3};
                       $lv[L].verify ;
                       send $lv[L] ((tick ; claim)) ;
                       work ;
                       case $lv[L]
                          ( valid => get $lv[L] {0};
                                     $li[L].success ;
                                     $m[R].subtract ;
                                     work ;
                                     $r[R] <- recv $m[R] ;
                                     send $li[L] $r[R] ;
                                     pay $li[L] {0};
                                     #sv[S] <- release $lv[L] ;
                                     #si[S] <- detach $li[L] ;
                                     #si[S] <- insurer <- #sv[S] $m[R]
                          | invalid => get $lv[L] {0};
                                       #li[L].failure ;
                                       pay $li[L] ;
                                       #sv[S] <- release $lv[L] ;
                                       #si[S] <- detach $li[L] ;
                                       #si[S] <- insurer <- #sv[S] $m[R] ) )
  }
TC time: 1.36709213257
Inference time: 3.58390808105
# Vars = 76
# Constraints = 224
% compilation successful!
% runtime successful!
```

B.6 Puzzle

```
type puzzle = /\ <{14}|
    &{ update : int -> money -o[R] |{0}> \/ puzzle,
       submit : int ^ &{ success : int -> money *[R] |{5}> \/ puzzle,
                        failure : |{9}> \/ puzzle } }
type money = &{ value : <{2}| int ^ money,
               coins : <{0}| lcoin }
type lcoin = 1
proc asset join : ($m[R] : lcoin), ($n[R] : lcoin) |{1}- ($o[R] : lcoin) =
  {
    wait $m[R] ;
    wait $n[R] ;
    work ;
    close $o[R]
  }
proc asset consume : ($m[R] : money) |{1}- ($o[R] : 1) =
```

```
  {
    work ;
    $m[R].coins ;
    pay $m[R] {0};
    $o[R] <- $m[R]
  }
proc asset add : ($m[R] : money), ($n[R] : money) |{10}- ($o[R] : money) =
  {
    work ;
    $m[R].value ;
    pay $m[R] {2};
    mval = recv $m[R] ;
    $n[R].value ;
    pay $n[R] {2};
    work ;
    nval = recv $n[R] ;
    let oval = (tick ; (tick ; mval) + (tick ; nval)) ;
    $m[R].coins ;
    pay $m[R] {0};
    $n[R].coins ;
    pay $n[R] {0};
    $ocoin[R] <- join <- $m[R] $n[R] ;
    $o[R] <- wallet <- oval $ocoin[R]
  }
proc asset wallet : (n : int), ($l[R] : lcoin) |- ($m[R] : money) =
  {
    case $m[R] ( value => get $m[R] {2};
                          work ;
                          send $m[R] ((tick ; n)) ;
                          $m[R] <- wallet <- n $l[R]
               | coins => get $m[R] {0};
                          $m[R] <- $l[R] )
  }
proc asset emp : . |{1}- ($l[R] : lcoin) =
  {
    work ;
    close $l[R]
  }
proc asset empty_wallet : . |{3}- ($m[R] : money) =
  {
    $l[R] <- emp <- ;
    work ;
    let n = (tick ; 0) ;
    $m[R] <- wallet <- n $l[R]
  }
proc contract game : (addr : int), ($m[R] : money) |- (#sp[S] : puzzle) =
  {
    $lp[L] <- accept #sp[S] ;
    get $lp[L] {14};
    case $lp[L] ( update => n = recv $lp[L] ;
                            work ;
                            $r[R] <- recv $lp[L] ;
```

```
                    if (tick ; (tick ; n) = (tick ; addr))
                    then
                      $newm[R] <- add <- $m[R] $r[R] ;
                      pay $lp[L] {0};
                      #sp[S] <- detach $lp[L] ;
                      #sp[S] <- game <- addr $newm[R]
                    else
                      $tmp[R] <- consume <- $r[R] ;
                      work ;
                      wait $tmp[R] ;
                      pay $lp[L] {0};
                      #sp[S] <- detach $lp[L] ;
                      work {8};
                      #sp[S] <- game <- addr $m[R]
         | submit => work ;
                    $m[R].value ;
                    pay $m[R] {2};
                    mval = recv $m[R] ;
                    send $lp[L] ((tick ; mval)) ;
                    work ;
                    case $lp[L]
                       ( success => work ;
                                    sol = recv $lp[L] ;
                                    send $lp[L] $m[R] ;
                                    pay $lp[L] {5};
                                    #sp[S] <- detach $lp[L] ;
                                    $emp[R] <- empty_wallet <- ;
                                    #sp[S] <- game <- addr $emp[R]
                       | failure => pay $lp[L] {9};
                                    #sp[S] <- detach $lp[L] ;
                                    #sp[S] <- game <- addr $m[R] ) )
  }
TC time: 1.64389610291
Inference time: 4.714012146
# Vars = 126
# Constraints = 389
% compilation successful!
% runtime successful!
```

B.7 Amortized Voting

```
type ballot = /\ <{16}| +{ open : int -> +{ vote : int -> |{0}> \/ ballot,
                                            novote : |{9}> \/ ballot },
                        closed : int ^ |{13}> \/ ballot }
type vote_list = +{ cons : |{4}> vote_list,
                  nil : 1 }
proc asset cons : ($t[P] : vote_list) |{5}- ($l[P] : vote_list) =
  {
    work ;
    $l[P].cons ;
    pay $l[P] {4};
    $l[P] <- $t[P]
  }
```

```
type voters = &{ check : <{0}| int -> +{ success : voters,
                                        failure : voters },
                size : <{0}| int ^ voters }
proc contract open_election : (T : int), ($vs[P] : voters),
                              ($c1[P] : vote_list), ($c2[P] : vote_list)
                                                  |{14}- (#sb[S] : ballot) =
  {
    $lb[L] <- accept #sb[S] ;
    get $lb[L] {16};
    work ;
    $lb[L].open ;
    v = recv $lb[L] ;
    $vs[P].check ;
    pay $vs[P] {0};
    send $vs[P] ((tick ; v)) ;
    work ;
    case $vs[P] ( success => $lb[L].vote ;
                             work ;
                             c = recv $lb[L] ;
                             if (tick ; (tick ; c) > (tick ; 0))
                             then
                               $c1n[P] <- cons <- $c1[P] ;
                               pay $lb[L] {0};
                               #sb[S] <- detach $lb[L] ;
                               #sb[S] <- check <- T $vs[P] $c1n[P] $c2[P]
                             else
                               $c2n[P] <- cons <- $c2[P] ;
                               pay $lb[L] {0};
                               #sb[S] <- detach $lb[L] ;
                               #sb[S] <- check <- T $vs[P] $c1[P] $c2n[P]
                | failure => $lb[L].novote ;
                             pay $lb[L] {9};
                             #sb[S] <- detach $lb[L] ;
                             #sb[S] <- check <- T $vs[P] $c1[P] $c2[P] )
  }
proc asset count_helper : (n : int), ($c[P] : vote_list) |{2}- ($s[P] : int ^ 1) =
  {
    case $c[P] ( cons => get $c[P] {4};
                         work ;
                         let n = (tick ; (tick ; n) + (tick ; 1)) ;
                         $s[P] <- count_helper <- n $c[P]
               | nil => wait $c[P] ;
                        work ;
                        send $s[P] ((tick ; n)) ;
                        close $s[P] )
  }
proc asset count_list : ($c[P] : vote_list) |{4}- ($s[P] : int ^ 1) =
  {
    work ;
    let n = (tick ; 0) ;
    $s[P] <- count_helper <- n $c[P]
  }
```

```
proc contract count : (T : int), ($vs[P] : voters),
                    ($c1[P] : vote_list), ($c2[P] : vote_list)
                                        |{14}- (#sb[S] : ballot) =
  {
    $s1[P] <- count_list <- $c1[P] ;
    $s2[P] <- count_list <- $c2[P] ;
    s1 = recv $s1[P] ;
    work ;
    s2 = recv $s2[P] ;
    work ;
    wait $s1[P] ;
    work ;
    wait $s2[P] ;
    if (tick ; (tick ; s1) > (tick ; s2))
    then
      #sb[S] <- closed_election <- s1 $vs[P]
    else
      #sb[S] <- closed_election <- s2 $vs[P]
  }
proc contract check : (T : int), ($vs[P] : voters),
                    ($c1[P] : vote_list), ($c2[P] : vote_list)
                                          |{18}- (#sb[S] : ballot) =
  {
    work ;
    $vs[P].size ;
    pay $vs[P] {0};
    n = recv $vs[P] ;
    if (tick ; (tick ; n) = (tick ; T))
    then
      #sb[S] <- count <- T $vs[P] $c1[P] $c2[P]
    else
      #sb[S] <- open_election <- T $vs[P] $c1[P] $c2[P]
  }
proc contract closed_election : (w : int), ($vs[P] : voters)
                                        |- (#sb[S] : ballot) =
  {
    $lb[L] <- accept #sb[S] ;
    get $lb[L] {16};
    work ;
    $lb[L].closed ;
    work ;
    send $lb[L] ((tick ; w)) ;
    pay $lb[L] {13};
    #sb[S] <- detach $lb[L] ;
    #sb[S] <- closed_election <- w $vs[P]
  }
TC time: 0.34499168396
Inference time: 11.1479759216
# Vars = 140
# Constraints = 457
% compilation successful!
% runtime successful!
```

B.8 Wallet

```
type coin = 1
type lcoin = +{ cons : coin *[R] lcoin,
               nil : 1 }
type money = /\ &{ value : <{2}| int ^ \/ money,
                  coins : <{5}| lcoin *[R] \/ money }
proc asset emp : . |{2}- ($l[R] : lcoin) =
  {
    work ;
    $l[R].nil ;
    work ;
    close $l[R]
  }
proc contract wallet : (n : int), ($l[R] : lcoin) |- (#sm[S] : money) =
  {
    $m[L] <- accept #sm[S] ;
    case $m[L] ( value => get $m[L] {2};
                          work ;
                          send $m[L] ((tick ; n)) ;
                          work {0};
                          #sm[S] <- detach $m[L] ;
                          #sm[S] <- wallet <- n $l[R]
               | coins => get $m[L] {5};
                          work {0};
                          work ;
                          send $m[L] $l[R] ;
                          $l[R] <- emp <- ;
                          work ;
                          let n = (tick ; 0) ;
                          #sm[S] <- detach $m[L] ;
                          #sm[S] <- wallet <- n $l[R] )
  }
TC time: 0.217914581299
Inference time: 6.70695304871
# Vars = 32
# Constraints = 102
% compilation successful!
% runtime successful!
```

C TYPES

First, I present the grammar for ordinary functional types τ with potential.

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

Next, I define the purely linear session types.

$$\begin{array}{lll} A_\mathsf{R} & ::= & V \mid \oplus\{\ell : A_\mathsf{R}\}_{\ell \in L} \mid \&\{\ell : A_\mathsf{R}\}_{\ell \in L} \mid A_m \multimap_m A_\mathsf{R} \mid A_m \otimes_m A_\mathsf{R} \mid \mathbf{1} \\ & \mid & \tau \to A_\mathsf{R} \mid \tau \times A_\mathsf{R} \mid \triangleright^r A_\mathsf{R} \mid \triangleleft^r A_\mathsf{R} \end{array}$$

Next, the shared linear session types.

$$\begin{array}{rcl} A_{\mathsf{L}} & ::= & V \mid \oplus\{\ell : A_{\mathsf{L}}\}_{\ell \in L} \mid \&\{\ell : A_{\mathsf{L}}\}_{\ell \in L} \mid A_m \multimap_m A_{\mathsf{L}} \mid A_m \otimes_m A_{\mathsf{L}} \\ & \mid & \tau \to A_{\mathsf{L}} \mid \tau \times A_{\mathsf{L}} \mid \triangleright^r A_{\mathsf{L}} \mid \triangleleft^r A_{\mathsf{L}} \\ & \mid & \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}} \end{array}$$

Finally, the shared session type.

$$A_{\mathsf{S}} \;::=\; \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}}$$

The client linear types follow the same grammar as purely linear types. The combined type is represented using A which denotes the type of either a client or contract process in linear mode.

$$\begin{array}{rcl} A_{\mathsf{T}} & ::= & A_{\mathsf{R}} \\ A & ::= & A_{\mathsf{T}} \mid A_{\mathsf{L}} \end{array}$$

First, the expressions at the functional layer are as follows (usual terms from a functional language).

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

The processes (proof terms) are as follows.

P, Q	$::=$	$c \leftarrow M \leftarrow \overline{a}\ ;\ P_c$	spawn process computed by M and continue with P_a, both communicating along fresh channel a
	$\mid$	$x \leftarrow y$	forward between x and y
	$\mid$	$x.l_k\ ;\ P$	send label l_k along x
	$\mid$	$\mathsf{case}\ x\ (l_i \Rightarrow P)$	branch on received label along x
	$\mid$	$\mathsf{send}\ x\ w\ ;\ P$	send channel/value w along x
	$\mid$	$y \leftarrow \mathsf{recv}\ x\ ;\ P$	receive channel/value along x and bind it to y
	$\mid$	$\mathsf{close}\ x$	close channel x
	$\mid$	$\mathsf{wait}\ x\ ;\ P$	wait on closing channel x
	$\mid$	$\mathsf{work}\ \{p\}\ ;\ P$	do work p, continue with P
	$\mid$	$\mathsf{get}\ x\ \{p\}\ ;\ P$	get potential p on channel x
	$\mid$	$\mathsf{pay}\ x\ \{p\}\ ;\ P$	pay potential p on channel x
	$\mid$	$x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ x_{\mathsf{S}}\ ;\ P_{x_{\mathsf{L}}}$	send an acquire request along x_{S}
	$\mid$	$x_{\mathsf{L}} \leftarrow \mathsf{accept}\ x_{\mathsf{S}}\ ;\ P_{x_{\mathsf{L}}}$	accept an acquire request along x_{S}
	$\mid$	$x_{\mathsf{S}} \leftarrow \mathsf{detach}\ x_{\mathsf{L}}\ ;\ P_{x_{\mathsf{S}}}$	send a detach request along x_{L}
	$\mid$	$x_{\mathsf{S}} \leftarrow \mathsf{release}\ x_{\mathsf{L}}\ ;\ P_{x_{\mathsf{S}}}$	receive a detach request along x_{L}

D TYPE SYSTEM

We first define the judgments we use in our type system.

$\Psi \Vdash^{q} M : \tau$	term M has type τ and needs potential q for evaluation
$\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (c_m : A)$	process P offers service of type A along channel c at mode $m = (\mathsf{S}, \mathsf{L}, \mathsf{T}, \mathsf{R})$ and uses shared channels from Γ and linear channels from Δ and functional variables from Ψ and stores potential q

Mode S stands for channels in shared mode. Mode L stands for shared channels in their linear mode. Mode T stands for linear channels that internally depend on shared processes. Mode R stands for purely linear channels offered by purely linear processes.

D.1 Monad

First, I present the rules concerning the monad.

Introduction Rules.

$$\frac{\Delta = \overline{d_R : D_R} \qquad \Psi \,;\, \cdot \,;\, \Delta \vDash^{q} P :: (x_R : A_R)}{\Psi \Vdash^{q} \{x_R \leftarrow P \leftarrow \overline{d_R}\} : \{A_R \leftarrow \overline{D_R}\}_R} \; \{\}I_R$$

$$\frac{\Gamma = \overline{a_S : A_S} \qquad \Delta = \overline{d_R : D_R} \qquad \Psi \,;\, \Gamma \,;\, \Delta \vDash^{q} P :: (x_S : A)}{\Psi \Vdash^{q} \{x_S \leftarrow P \leftarrow \overline{a_S} \,;\, \overline{d_R}\} : \{A \leftarrow \overline{A_S} \,;\, \overline{D_R}\}_S} \; \{\}I_S$$

$$\frac{\Gamma = \overline{a_S : A_S} \qquad \Delta = \overline{d : D} \qquad \Psi \,;\, \Gamma \,;\, \Delta \vDash^{q} P :: (x_T : A)}{\Psi \Vdash^{q} \{x_T \leftarrow P \leftarrow \overline{a_S} \,;\, \overline{d}\} : \{A \leftarrow \overline{A_S} \,;\, \overline{D}\}_T} \; \{\}I_T$$

Elimination Rules.

$$\frac{\begin{array}{c} r = p + q \qquad \Delta = \overline{d_R : D_R} \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \\ \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{D_R}\}_R \qquad \Psi_2 \,;\, \Gamma \,;\, \Delta', (x_R : A) \vDash^{q} Q :: (z_m : C) \end{array}}{\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \vDash^{r} x_R \leftarrow M \leftarrow \overline{d_R} \,;\, Q :: (z_m : C)} \; \{\}E_{Rm(\in\{R,S,L,T\})}$$

$$\frac{\begin{array}{c} r = p + q \qquad \Gamma \supseteq \overline{a_S : A_S} \qquad \Delta = \overline{d_R : D_R} \qquad (A_S, A_S)\ \mathsf{esync} \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \\ \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{A_S} \,;\, \overline{D_R}\}_S \qquad \Psi_2 \,;\, \Gamma, (x_S : A) \,;\, \Delta' \vDash^{q} Q :: (z_m : C) \end{array}}{\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \vDash^{r} x_S \leftarrow M \leftarrow \overline{d_R} \,;\, Q :: (z_m : C)} \; \{\}E_{Sm(\in\{S,L,T\})}$$

$$\frac{\begin{array}{c} r = p + q \qquad \Gamma \supseteq \overline{a_S : A_S} \qquad \Delta = \overline{d : D} \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \\ \Psi_1 \Vdash^{p} M : \{A \leftarrow \overline{A_S} \,;\, \overline{D}\}_T \qquad \Psi_2 \,;\, \Gamma \,;\, (x_T : A), \Delta' \vDash^{q} Q :: (z_m : C) \end{array}}{\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \vDash^{r} x_T \leftarrow M \leftarrow \overline{a_S} \,;\, \overline{d} \,;\, Q :: (z_m : C)} \; \{\}E_{Tm(\in\{L,T\})}$$

The rest of the rules for expressions in the functional layer are standard. We skip them and discuss the process layer.

D.2 Forwarding

$$\frac{q = 0}{\Psi \,;\, \Gamma \,;\, (y_m : A) \vDash^{q} x_m \leftarrow y_m :: (x_m : A)} \; \mathsf{fwd}_{m(\in\{R,T\})}$$

D.3 Labels and Branching

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta \vDash^{q} P :: (x_m : A_k) \qquad (k \in L)}{\Psi \,;\, \Gamma \,;\, \Delta \vDash^{q} x_m.k \,;\, P :: (x_m : \oplus\{\ell : A_\ell\}_{\ell \in L})} \; \oplus R$$

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta, (x_m : A_\ell) \vDash^{q} Q_\ell :: (z_k : C) \qquad (\forall \ell \in L)}{\Psi \,;\, \Gamma \,;\, \Delta, (x_m : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vDash^{q} \mathsf{case}\ x_m\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z_k : C)} \; \oplus L$$

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta \vDash^{q} P :: (x_m : A_\ell) \qquad (\forall \ell \in L)}{\Psi \,;\, \Gamma \,;\, \Delta \vDash^{q} \mathsf{case}\ x_m\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x_m : \&\{\ell : A_\ell\}_{\ell \in L})} \; \& R$$

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : A_\ell) \vDash^{q} Q_\ell :: (z_k : C) \qquad (k \in L)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : \&\{\ell : A_\ell\}_{\ell \in L}) \vDash^{q} x_m.k \; ; \; P :: (z_k : C)} \; \&L$$

D.4 Linear Channel Communication

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta \vDash^{q} P :: (x_m : B)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (w_n : A) \vDash^{q} \mathsf{send}\; x_m\; w_n \; ; \; P :: (x_m : A \otimes_n B)} \; \otimes_n R$$

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (y_n : A), (x_m : B) \vDash^{q} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : A \otimes_n B) \vDash^{q} y_n \leftarrow \mathsf{recv}\; x_m \; ; \; Q :: (z_k : C)} \; \otimes_n L$$

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (y_n : A) \vDash^{q} P :: (x_m : B)}{\Psi \; ; \; \Gamma \; ; \; \Delta \vDash^{q} y_n \leftarrow \mathsf{recv}\; x_m \; ; \; P :: (x_m : A \multimap B)} \; \multimap_n R$$

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : B) \vDash^{q} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (w_n : A), (x_m : A \multimap B) \vDash^{q} \mathsf{send}\; x_m\; w_n \; ; \; Q :: (z_k : C)} \; \multimap_n L$$

D.5 Value Communication

$$\frac{r = p + q \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \qquad \Psi_1 \Vdash^{p} M : \tau \qquad \Psi_2 \; ; \; \Gamma \; ; \; \Delta \vDash^{q} P :: (x_m : A)}{\Psi \; ; \; \Gamma \; ; \; \Delta \vDash^{r} \mathsf{send}\; x_m\; M \; ; \; P :: (x_m : \tau \times A)} \; \times R$$

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

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

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

D.6 Termination

$$\frac{q = 0}{\Psi \; ; \; \Gamma \; ; \; \cdot \vDash^{q} \mathsf{close}\; x_m :: (x_m : \mathbf{1})} \; \mathbf{1}R \qquad \frac{\Psi \; ; \; \Gamma \; ; \; \Delta \vDash^{q} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : \mathbf{1}) \vDash^{q} \mathsf{wait}\; x_m \; ; \; Q :: (z_k : C)} \; \mathbf{1}L$$

D.7 Potential

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

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

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

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

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

D.8 Acquiring and Releasing

$$\frac{\Delta \text{ purelin} \qquad \Psi \,;\, \Gamma \,;\, \Delta \vDash^{g} P :: (x_{\mathsf{L}} : A_{\mathsf{L}})}{\Psi \,;\, \Gamma \,;\, \Delta \vDash^{g} x_{\mathsf{L}} \leftarrow \text{accept } x_{\mathsf{S}} \,;\, P :: (x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}})} \uparrow^{\mathsf{S}}_{\mathsf{L}} R$$

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta, (x_{\mathsf{L}} : A_{\mathsf{L}}) \vDash^{g} Q :: (z_m : C)}{\Psi \,;\, \Gamma, (x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}) \,;\, \Delta \vDash^{g} x_{\mathsf{L}} \leftarrow \text{acquire } x_{\mathsf{S}} \,;\, Q :: (z_m : C)} \uparrow^{\mathsf{S}}_{\mathsf{L}} L_{m(=\mathsf{L},\mathsf{T})}$$

$$\frac{\Delta \text{ purelin} \qquad \Psi \,;\, \Gamma \,;\, \Delta \vDash^{g} P :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Psi \,;\, \Gamma \,;\, \Delta \vDash^{g} x_{\mathsf{S}} \leftarrow \text{detach } x_{\mathsf{L}} \,;\, P :: (x_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}})} \downarrow^{\mathsf{S}}_{\mathsf{L}} R$$

$$\frac{\Psi \,;\, \Gamma, (x_{\mathsf{S}} : A_{\mathsf{S}}) \,;\, \Delta \vDash^{g} Q :: (z_m : C)}{\Psi \,;\, \Gamma \,;\, \Delta, (x_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}) \vDash^{g} x_{\mathsf{S}} \leftarrow \text{release } x_{\mathsf{L}} \,;\, Q :: (z_m : C)} \downarrow^{\mathsf{S}}_{\mathsf{L}} L_{m(=\mathsf{L},\mathsf{T})}$$

E OPERATIONAL COST SEMANTICS

First, we define the judgments for expressions. The first judgment is a small step semantics for expressions, $M \mapsto M'$ and M val. Finally, we introduce another judgment for processes, $\text{proc}(c_m, w, P) \mapsto \text{proc}(c'_m, w', P')$ and a new predicate $\text{msg}(c_m, w, M)$ to denote a message. Additionally, we define processes with a hole for a compact representation of the cost semantics.

$$\begin{array}{rcl} P[\cdot] & ::= & c \leftarrow [\cdot] \leftarrow a_i \,;\, P_c \\ & | & \text{send } c\ [\cdot] \,;\, P \end{array}$$

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

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

$$\frac{(c_{\mathsf{S}} \text{ fresh})}{\begin{array}{c}\text{proc}(d_m, w, x_{\mathsf{S}} \leftarrow \{x'_{\mathsf{S}} \leftarrow P_{x'_{\mathsf{S}},\overline{y},\overline{z}} \leftarrow \overline{y} \,;\, \overline{z}\} \leftarrow \overline{a} \,;\, \overline{b} \,;\, Q) \mapsto \\ \text{proc}(c_{\mathsf{S}}, 0, P_{c_{\mathsf{S}},\overline{a},\overline{b}}) \quad \text{proc}(d_m, w, [c_{\mathsf{S}}/x_{\mathsf{S}}]Q)\end{array}} \{\}E_{\mathsf{S}m}$$

$$\frac{(c_{\mathsf{T}} \text{ fresh})}{\begin{array}{c}\text{proc}(d_{\mathsf{T}}, w, x_{\mathsf{T}} \leftarrow \{x'_{\mathsf{T}} \leftarrow P_{x'_{\mathsf{T}},\overline{y},\overline{z}} \leftarrow \overline{y} \,;\, \overline{z}\} \leftarrow \overline{a} \,;\, \overline{b} \,;\, Q) \mapsto \\ \text{proc}(c_{\mathsf{T}}, 0, P_{c_{\mathsf{T}},\overline{a},\overline{b}}) \quad \text{proc}(d_{\mathsf{T}}, w, [c_{\mathsf{T}}/x_{\mathsf{L}}]Q)\end{array}} \{\}E_{\mathsf{TT}}$$

$$\frac{}{\text{msg}(d_m, w', M) \quad \text{proc}(c_m, w, c_m \leftarrow d_m) \mapsto \text{msg}(c_m, w + w', [c_m/d_m]M)} \text{ fwd}^{+}$$

$$\frac{}{\text{proc}(c_m, w, c_m \leftarrow d_m) \quad \text{msg}(e_l, w', M(c_m)) \mapsto \text{msg}(e_l, w + w', M(d_m))} \text{ fwd}^{-}$$

$$\frac{(c^+_m \text{ fresh})}{\text{proc}(c_m, w, c_m.\ell \,;\, P) \mapsto \text{proc}(c^+_m, w, [c^+_m/c_m]P) \quad \text{msg}(c_m, 0, c_m.\ell \,;\, c_m \leftarrow c^+_m)} \oplus C_s$$

$$\frac{}{\begin{array}{c}\text{msg}(c_m, w, c_m.\ell \,;\, c_m \leftarrow c^+_m) \quad \text{proc}(d_k, w', \text{case } c_m\ (l \Rightarrow Q_l)_{l \in L}) \mapsto \\ \text{proc}(d_k, w + w', [c^+_m/c_m]Q_\ell)\end{array}} \oplus C_r$$

$$\frac{(c_m^+ \text{ fresh})}{\mathsf{proc}(d_k, w, c_m.\ell \;;\; P) \mapsto \mathsf{msg}(c_m^+, 0, c_m.\ell \;;\; c_m^+ \leftarrow c_m) \quad \mathsf{proc}(d_k, w, [c_m^+/c_m]P)} \;\&C_s$$

$$\frac{}{\mathsf{proc}(c_m, w', \mathsf{case}\ c_m\ (l \Rightarrow Q_l)_{l \in L}) \quad \mathsf{msg}(c_m^+, 0, c_m.\ell \;;\; c_m^+ \leftarrow c_m) \mapsto \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m]Q_\ell)} \;\&C_r$$

$$\frac{(c_m^+ \text{ fresh})}{\mathsf{proc}(c_m, w, \mathsf{send}\ c_m\ e_n \;;\; P) \mapsto \mathsf{proc}(c_m^+, w, [c_m^+/c_m]P) \quad \mathsf{msg}(c_m, 0, \mathsf{send}\ c_m\ e_n \;;\; c_m \leftarrow c_m^+)} \;\otimes_n C_s$$

$$\frac{}{\mathsf{msg}(c_m, w, \mathsf{send}\ c_m\ e_n \;;\; c_m \leftarrow c_m^+) \quad \mathsf{proc}(d_k, w', x_n \leftarrow \mathsf{recv}\ c_m \;;\; Q) \mapsto \mathsf{proc}(d_k, w + w', [c_m^+/c_m][e_n/x_n]Q)} \;\otimes_n C_r$$

$$\frac{(c_m^+ \text{ fresh})}{\mathsf{proc}(d_k, w, \mathsf{send}\ c_m\ e_n \;;\; P) \mapsto \mathsf{msg}(c_m^+, 0, \mathsf{send}\ c_m\ e_n \;;\; c_m^+ \leftarrow c_m) \quad \mathsf{proc}(d_k, w, [c_m^+/c_m]P)} \;\multimap_n C_s$$

$$\frac{}{\mathsf{proc}(c_m, w', x_n \leftarrow \mathsf{recv}\ c_m \;;\; Q) \quad \mathsf{msg}(c_m^+, w, \mathsf{send}\ c_m\ e_n \;;\; c_m^+ \leftarrow c_m) \mapsto \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m][e_n/x_n]Q)} \;\multimap_n C_r$$

$$\frac{(c_m^+ \text{ fresh}) \qquad N \text{ val}}{\mathsf{proc}(c_m, w, \mathsf{send}\ c_m\ N \;;\; P) \mapsto \mathsf{proc}(c_m^+, w, [c_m^+/c_m]P) \quad \mathsf{msg}(c_m, 0, \mathsf{send}\ c_m\ N \;;\; c_m \leftarrow c_m^+)} \;\times C_s$$

$$\frac{}{\mathsf{msg}(c_m, w, \mathsf{send}\ c_m\ N \;;\; c_m \leftarrow c_m^+) \quad \mathsf{proc}(d_k, w', x \leftarrow \mathsf{recv}\ c_m \;;\; Q) \mapsto \mathsf{proc}(d_k, w + w', [c_m^+/c_m][N/x]Q)} \;\times C_r$$

$$\frac{(c_m^+ \text{ fresh}) \qquad N \text{ val}}{\mathsf{proc}(d_k, w, \mathsf{send}\ c_m\ N \;;\; P) \mapsto \mathsf{msg}(c_m^+, 0, \mathsf{send}\ c_m\ N \;;\; c_m^+ \leftarrow c_m) \quad \mathsf{proc}(d_k, w, [c_m^+/c_m]P)} \;\rightarrow C_s$$

$$\frac{}{\mathsf{proc}(c_m, w', x \leftarrow \mathsf{recv}\ c_m \;;\; Q) \quad \mathsf{msg}(c_m^+, w, \mathsf{send}\ c_m\ N \;;\; c_m^+ \leftarrow c_m) \mapsto \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m][N/x]Q)} \;\rightarrow C_r$$

$$\frac{}{\mathsf{proc}(c_m, w, \mathsf{close}\ c_m) \mapsto \mathsf{msg}(c_m, w, \mathsf{close}\ c_m)} \;\mathbf{1}C_s$$

$$\frac{}{\mathsf{msg}(c_m, w, \mathsf{close}\ c_m) \quad \mathsf{proc}(d_k, w', \mathsf{wait}\ c_m \;;\; Q) \mapsto \mathsf{proc}(d_k, w + w', Q)} \;\mathbf{1}C_r$$

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

$$\frac{(c_m^+ \text{ fresh})}{\mathsf{proc}(c_m, w, \mathsf{pay}\ c_m\ \{r\} \;;\; P) \mapsto \mathsf{proc}(c_m^+, w, [c_m^+/c_m]P) \quad \mathsf{msg}(c_m, 0, \mathsf{pay}\ c_m\ \{r\} \;;\; c_m \leftarrow c_m^+)} \;\triangleright C_s$$

$$\frac{}{\mathsf{msg}(c_m, w, \mathsf{pay}\ c_m\ \{r\}\ ;\ c_m \leftarrow c_m^+) \quad \mathsf{proc}(d_k, w', \mathsf{get}\ c_m\ \{r\}\ ;\ Q) \mapsto \mathsf{proc}(d_k, w + w', [c_m^+/c_m]Q)} \triangleright C_r$$

$$\frac{(c_m^+ \text{ fresh})}{\mathsf{proc}(d_k, w, \mathsf{pay}\ c_m\ \{r\}\ ;\ P) \mapsto \mathsf{msg}(c_m^+, 0, \mathsf{pay}\ c_m\ \{r\}\ ;\ c_m^+ \leftarrow c) \quad \mathsf{proc}(d_k, w, [c_m^+/c_m]P)} \triangleleft C_s$$

$$\frac{}{\mathsf{proc}(c_m, w', \mathsf{get}\ c_m\ \{r\}\ ;\ Q) \quad \mathsf{msg}(c_m^+, w, \mathsf{pay}\ c_m\ \{r\}\ ;\ c_m^+ \leftarrow c_m) \mapsto \mathsf{proc}(c_m, w + w', [c_m^+/c_m]Q)} \triangleleft C_r$$

$$\frac{(a_{\mathsf{L}} \text{ fresh})}{\mathsf{proc}(a_{\mathsf{S}}, w', x_{\mathsf{L}} \leftarrow \mathsf{accept}\ a_{\mathsf{S}}\ ;\ P_{x_{\mathsf{L}}}) \quad \mathsf{proc}(c_m, w, x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ a_{\mathsf{S}}\ ;\ Q_{x_{\mathsf{L}}}) \mapsto \mathsf{proc}(a_{\mathsf{L}}, w', P_{a_{\mathsf{L}}}) \quad \mathsf{proc}(c_m, w, Q_{a_{\mathsf{L}}})} \uparrow_{\mathsf{L}}^{\mathsf{S}} C$$

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

F CONFIGURATION TYPING

$$\frac{}{\Gamma_0 \overset{0}{\vDash} (\cdot) :: (\cdot\ ;\ \cdot)} \mathsf{emp}$$

$$\frac{\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta'_{\mathsf{R}}) \qquad \cdot\ ;\ \cdot\ ;\ \Delta'_{\mathsf{R}} \overset{q}{\vdash} P :: (x_{\mathsf{R}} : A_{\mathsf{R}})}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{R}}, w, P) :: (\Gamma\ ;\ \Delta, (x_{\mathsf{R}} : A_{\mathsf{R}}))} \mathsf{proc}_{\mathsf{R}}$$

$$\frac{(x_{\mathsf{S}} : A_{\mathsf{S}}) \in \Gamma_0 \qquad (A_{\mathsf{S}}, A_{\mathsf{S}})\ \mathsf{esync} \qquad \Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta'_{\mathsf{R}}) \qquad \cdot\ ;\ \Gamma_0\ ;\ \Delta'_{\mathsf{R}} \overset{q}{\vdash} P :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{S}}, w, P) :: (\Gamma, (x_{\mathsf{S}} : A_{\mathsf{S}})\ ;\ \Delta)} \mathsf{proc}_{\mathsf{S}}$$

$$\frac{(x_{\mathsf{S}} : A_{\mathsf{S}}) \in \Gamma_0 \qquad (A_{\mathsf{L}}, A_{\mathsf{S}})\ \mathsf{esync} \qquad \Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta') \qquad \cdot\ ;\ \Gamma_0\ ;\ \Delta' \overset{q}{\vdash} P :: (x_{\mathsf{L}} : A_{\mathsf{L}})}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{L}}, w, P) :: (\Gamma, (x_{\mathsf{S}} : A_{\mathsf{S}})\ ;\ \Delta, (x_{\mathsf{L}} : A_{\mathsf{L}}))} \mathsf{proc}_{\mathsf{L}}$$

$$\frac{\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta') \qquad \cdot\ ;\ \Gamma_0\ ;\ \Delta' \overset{q}{\vdash} P :: (x_{\mathsf{T}} : A_{\mathsf{T}})}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{T}}, w, P) :: (\Gamma\ ;\ \Delta, (x_{\mathsf{T}} : A_{\mathsf{T}}))} \mathsf{proc}_{\mathsf{T}}$$

$$\frac{\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta, \Delta') \qquad \cdot\ ;\ \cdot\ ;\ \Delta' \overset{q}{\vdash} M :: (x_m : A)}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{msg}(x_m, w, M) :: (\Gamma\ ;\ \Delta, (x_m : A))} \mathsf{msg}$$

In addition, for a well-typed configuration $\Gamma_0 \overset{E}{\models} \Omega :: (\Gamma \;;\; \Delta)$, we need the following well-formedness conditions.

- All channels in Γ_0, Γ and Δ are unique.
- $\Gamma \subseteq \Gamma_0$.

F.1 Equi-Synchronizing

$$\frac{(A_\ell, C_S)\ \mathsf{esync} \quad (\forall \ell \in L)}{(\oplus\{\ell : A_\ell\}_{\ell \in L}, C_S)\ \mathsf{esync}}\ \oplus \qquad \frac{(A_\ell, C_S)\ \mathsf{esync} \quad (\forall \ell \in L)}{(\&\{\ell : A_\ell\}_{\ell \in L}, C_S)\ \mathsf{esync}}\ \&$$

$$\frac{(B, C_S)\ \mathsf{esync}}{(A \otimes B, C_S)\ \mathsf{esync}}\ \otimes \qquad \frac{(B, C_S)\ \mathsf{esync}}{(A \multimap B, C_S)\ \mathsf{esync}}\ \multimap$$

$$\frac{(B, C_S)\ \mathsf{esync}}{(\tau \times B, C_S)\ \mathsf{esync}}\ \times \qquad \frac{(B, C_S)\ \mathsf{esync}}{(\tau \to B, C_S)\ \mathsf{esync}}\ \to$$

$$\frac{(A, C_S)\ \mathsf{esync}}{(\triangleright^r A, C_S)\ \mathsf{esync}}\ \triangleright \qquad \frac{(A, C_S)\ \mathsf{esync}}{(\triangleleft^r A, C_S)\ \mathsf{esync}}\ \triangleleft$$

$$\frac{(A_L, \uparrow_L^S A_L)\ \mathsf{esync}}{(\uparrow_L^S A_L, \uparrow_L^S A_L)\ \mathsf{esync}}\ \uparrow_L^S \qquad \frac{(A_S, A_S)\ \mathsf{esync}}{(\downarrow_L^S A_S, A_S)\ \mathsf{esync}}\ \downarrow_L^S$$

F.2 Purely Linear Context

$$\frac{}{\cdot\ \mathsf{purelin}}\ \mathsf{emp} \qquad \frac{x_R : A_R \qquad \Delta\ \mathsf{purelin}}{x_R : A_R, \Delta\ \mathsf{purelin}}\ \mathsf{step}$$

G TYPE SAFETY

LEMMA 2 (RENAMING). *The following renamings are allowed.*

- *If* $\Psi \;;\; \Gamma, (x_S : A_S) \;;\; \Delta \vdash^g P_{x_S} :: (z_k : C)$ *is well-typed, so is* $\Gamma, (c_S : A_S) \;;\; \Delta \vdash^g P_{c_S} :: (z_k : C)$.
- *If* $\Psi \;;\; \Gamma \;;\; \Delta, (x_m : A) \vdash^g P_{x_m} :: (z_k : C)$ *is well-typed, so is* $\Gamma \;;\; \Delta, (c_m : A) \vdash^g P_{c_m} :: (z_k : C)$.
- *If* $\Psi \;;\; \Gamma \;;\; \Delta \vdash^g P_{z_k} :: (z_k : C)$ *is well-typed, so is* $\Gamma \;;\; \Delta \vdash^g P_{c_k} :: (c_k : C)$.

LEMMA 3 (INVARIANTS). *The process typing judgment* $\Psi \;;\; \Gamma \;;\; \Delta \vdash^g P :: (x_m : A)$ *preserves the following invariants.*

(R) $\Psi \;;\; \cdot \;;\; \Delta_R \vdash^g P :: (x_R : A_R)$
(S/L) $\Psi \;;\; \Gamma \;;\; \Delta_R \vdash^g P :: (x_S : A_S)$ *or* $\Psi \;;\; \Gamma \;;\; \Delta \vdash^g P :: (x_L : A_L)$
(T) $\Psi \;;\; \Gamma \;;\; \Delta \vdash^g P :: (x_T : A_T)$

PROOF. The elimination rules preserve the invariant trivially because they can only be applied when the invariant is maintained and the premise in each rule maintains the same invariant.

- Case (E_{RR}) : This rule can only be applied when the context is purely linear. And then adding x_R to the context will keep it purely linear.
- Case (E_{RS}, E_{RL}) : This rule can only be applied if offering channel is either in S or L mode and the context is purely linear. Hence, adding x_R to the context is allowed.
- Case (E_{RT}) : The context is mixed linear, hence adding a purely linear channel is valid.
- Case (E_{SS}, E_{SL}, E_{ST}) : The context has shared channels in each case, hence adding another shared channel is valid.
- Case (E_{TT}) : Adding a client linear channel to a mixed context is valid.

- Case (fwd) :

(R) : $\Delta_{\mathsf{R}} = (y_{\mathsf{R}} : A_{\mathsf{R}})$ which is valid since Δ_{R} is purely linear and there are no premises.

(S/L) : This rule cannot be applied since the fwd rule applies only when the offering mode is R. Hence, there is a mode mismatch.

(T) : Analogous to (S/L).

- Case ($\oplus R$) :

(R) :

$$\frac{\Psi \ ; \ \cdot \ ; \ \Delta_{\mathsf{R}} \vdash^{q} P :: (x_{\mathsf{R}} : A_k) \qquad (k \in L)}{\Psi \ ; \ \cdot \ ; \ \Delta_{\mathsf{R}} \vdash^{q} (x_{\mathsf{R}}.k \ ; \ P) :: (x_{\mathsf{R}} : \oplus\{\ell : A_\ell\}_{\ell \in L})} \ \oplus R$$

The context doesn't change, and the type of the offered channel remains purely linear.

(S/L) :

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

The context doesn't change, and the type of the offered channel remains shared linear. Also, the mode of x cannot be S because the type doesn't allow that.

(T) :

$$\frac{\Psi \ ; \ \Gamma \ ; \ \Delta \vdash^{q} P :: (x_{\mathsf{T}} : A_k) \qquad (k \in L)}{\Psi \ ; \ \Gamma \ ; \ \Delta \vdash^{q} (x_{\mathsf{T}}.k \ ; \ P) :: (x_{\mathsf{T}} : \oplus\{\ell : A_\ell\}_{\ell \in L})} \ \oplus R$$

The context doesn't change, and the type of the offered channel remains client linear.

- Case ($\oplus L$) :

(R) :

$$\frac{\Psi \ ; \ \cdot \ ; \ \Delta_{\mathsf{R}}, (x_{\mathsf{R}} : A_\ell) \vdash^{q} Q_\ell :: (z_{\mathsf{R}} : C) \qquad (\forall \ell \in L)}{\Psi \ ; \ \cdot \ ; \ \Delta_{\mathsf{R}}, (x_{\mathsf{R}} : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vdash^{q} \mathsf{case}\ x_{\mathsf{R}}\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z_{\mathsf{R}} : C)} \ \oplus L$$

The context remains purely linear, and the offered channel doesn't change.

(S/L) :

$$\frac{\Psi \ ; \ \Gamma \ ; \ \Delta, (x_m : A_\ell) \vdash^{q} Q_\ell :: (z_k : C) \qquad (\forall \ell \in L)}{\Psi \ ; \ \Gamma \ ; \ \Delta, (x_m : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vdash^{q} \mathsf{case}\ x_m\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z_k : C)} \ \oplus L$$

The mode of x_m doesn't change, and the offered channel doesn't change.

(T) :

$$\frac{\Psi \ ; \ \Gamma \ ; \ \Delta, (x_m : A_\ell) \vdash^{q} Q_\ell :: (z_{\mathsf{T}} : C) \qquad (\forall \ell \in L)}{\Psi \ ; \ \Gamma \ ; \ \Delta, (x_m : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vdash^{q} \mathsf{case}\ x_m\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z_{\mathsf{T}} : C)} \ \oplus L$$

The mode of the channel x_m doesn't change, and the offered channel doesn't change.

- Case ($\multimap_n R$) :

(R) :

$$\frac{\Psi \ ; \ \cdot \ ; \ \Delta_{\mathsf{R}}, (y_{\mathsf{R}} : A) \vdash^{q} P :: (x_{\mathsf{R}} : B)}{\Psi \ ; \ \cdot \ ; \ \Delta_{\mathsf{R}} \vdash^{q} y_{\mathsf{R}} \leftarrow \mathsf{recv}\ x_{\mathsf{R}} \ ; \ P :: (x_{\mathsf{R}} : A \multimap_{\mathsf{R}} B)} \ \multimap_{\mathsf{R}} R$$

A process offering a purely linear channel only allows exchanging purely linear channels. This channel gets added to the purely linear context, and the type of the offered channel remains purely linear.

(S/L) :

$$\frac{\Psi \ ; \ \Gamma \ ; \ \Delta, (y_n : A) \vdash^{q} P :: (x_{\mathsf{L}} : B)}{\Psi \ ; \ \Gamma \ ; \ \Delta \vdash^{q} y_n \leftarrow \mathsf{recv}\ x_{\mathsf{L}} \ ; \ P :: (x_{\mathsf{L}} : A \multimap_n B)} \ \multimap_n R$$

A linear channel gets added to the mixed linear context, and the type of the offered channel remains shared linear. Also, the mode of x cannot be S because the type doesn't allow that.

(T) :

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (y_n : A) \vDash^{q} P :: (x_{\mathsf{T}} : B)}{\Psi \; ; \; \Gamma \; ; \; \Delta \vDash^{q} y_n \leftarrow \mathsf{recv}\; x_{\mathsf{T}} \; ; \; P :: (x_{\mathsf{T}} : A \multimap_n B)} \multimap_n R$$

A linear channel gets added to the mixed linear context, and the type of the offered channel remains client linear.

- Case $(\multimap_n L)$:

(R) :

$$\frac{\Psi \; ; \; \cdot \; ; \; \Delta_{\mathsf{R}}, (x_{\mathsf{R}} : B) \vDash^{q} Q :: (z_{\mathsf{R}} : C)}{\Psi \; ; \; \cdot \; ; \; \Delta_{\mathsf{R}}, (w_{\mathsf{R}} : A), (x_{\mathsf{R}} : A \multimap_{\mathsf{R}} B) \vDash^{q} \mathsf{send}\; x_{\mathsf{R}}\; w_{\mathsf{R}} \; ; \; Q :: (z_{\mathsf{R}} : C)} \multimap_{\mathsf{R}} L$$

A purely linear channel is allowed in a purely linear context. The context remains purely linear, and the offered channel doesn't change.

(S/L) :

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : B) \vDash^{q} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (w_n : A), (x_m : A \multimap_n B) \vDash^{q} \mathsf{send}\; x_m\; w_n \; ; \; Q :: (z_k : C)} \multimap_n L$$

A linear channel is allowed in a mixed linear context. The mode of the channel x_m doesn't change, and the offered channel doesn't change.

(T) :

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : B) \vDash^{q} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (w_n : A), (x_m : A \multimap_n B) \vDash^{q} \mathsf{send}\; x_m\; w_n \; ; \; Q :: (z_k : C)} \multimap_n L$$

A linear channel is allowed in a mixed linear context. The mode of the channel x_m doesn't change, and the offered channel doesn't change.

- Case $(\uparrow_{\mathsf{L}}^{\mathsf{S}} R)$:

(R) : This rule cannot be applied since the offered channel in this case should be purely linear, which is not the case for $\uparrow_{\mathsf{L}}^{\mathsf{S}} R$ rule.

(S/L) :

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

The context doesn't change and the offered channel switches its mode from S to L. Moreover, the rule cannot be applied if the offered channel is in L mode, since there will be a mode mismatch.

(T) : This rule cannot be applied since the offered channel should be in T mode, which doesn't match with S.

- Case $(\downarrow_{\mathsf{L}}^{\mathsf{S}} R)$: Analogous to $\uparrow_{\mathsf{L}}^{\mathsf{S}} R$.
- Case $(\uparrow_{\mathsf{L}}^{\mathsf{S}} L)$:

(R) : This rule cannot be applied since the context should be purely linear, which is not the case for $\uparrow_{\mathsf{L}}^{\mathsf{S}} L$ rule.

(S/L) :

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

A shared linear channel is allowed in a mixed linear context. The mode of the offering channel is unchanged. A shared channel is removed from the shared context, but the new context is still shared.

(T) :

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

A shared linear channel gets added to the mixed linear context, which is allowed. A shared channel is removed from the shared context, but the new context is still shared. Moreover, the offered channel remains at the same mode.

- Case ($\downarrow_{\mathsf{L}}^{\mathsf{S}} L$) : Analogous to $\uparrow_{\mathsf{L}}^{\mathsf{S}} L$ rule.

□

Lemma 4 (Configuration Weakening). *If we have a well-typed configuration, $\Gamma_0 \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta)$, then for a shared channel $c_{\mathsf{S}} : B_{\mathsf{S}} \notin \Gamma_0$, we can weaken Γ_0 and get $\Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta)$.*

Proof. We case analyze on the configuration typing judgment.

- Case (emp) : We have $\Gamma_0 \stackrel{0}{\vDash} (\cdot) :: (\cdot \; ; \; \cdot)$. But, since there is no premise, we use the emp rule to get $\Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{0}{\vDash} (\cdot) :: (\cdot \; ; \; \cdot)$.
- Case ($\mathsf{proc}_{\mathsf{R}}$) : We have $\Gamma_0 \stackrel{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{R}}, w, P) :: (\Gamma \; ; \; \Delta, (x_{\mathsf{R}} : A_{\mathsf{R}}))$. Inverting the $\mathsf{proc}_{\mathsf{R}}$ rule,

$$\frac{\Gamma_0 \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}}) \qquad \cdot \; ; \; \cdot \; ; \; \Delta'_{\mathsf{R}} \stackrel{q}{\vdash} P :: (x_{\mathsf{R}} : A_{\mathsf{R}})}{\Gamma_0 \stackrel{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{R}}, w, P) :: (\Gamma \; ; \; \Delta, (x_{\mathsf{R}} : A_{\mathsf{R}}))} \; \mathsf{proc}_{\mathsf{R}}$$

we get $\Gamma_0 \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}})$. By the induction hypothesis, $\Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}})$. Applying the $\mathsf{proc}_{\mathsf{R}}$ rule,

$$\frac{\Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}}) \qquad \cdot \; ; \; \cdot \; ; \; \Delta'_{\mathsf{R}} \stackrel{q}{\vdash} P :: (x_{\mathsf{R}} : A_{\mathsf{R}})}{\Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{R}}, w, P) :: (\Gamma \; ; \; \Delta, (x_{\mathsf{R}} : A_{\mathsf{R}}))} \; \mathsf{proc}_{\mathsf{R}}$$

- Case ($\mathsf{proc}_{\mathsf{S}}$) : We have $\Gamma_0 \stackrel{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{S}}, w, P) :: (\Gamma, (x_{\mathsf{S}} : A_{\mathsf{S}}))$. Inverting the $\mathsf{proc}_{\mathsf{S}}$ rule,

$$\frac{(x_{\mathsf{S}} : A_{\mathsf{S}}) \in \Gamma_0 \qquad (A_{\mathsf{S}}, A_{\mathsf{S}})\;\mathsf{esync} \qquad \Gamma_0 \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}}) \qquad \cdot \; ; \; \Delta'_{\mathsf{R}} \stackrel{q}{\vdash} P :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Gamma_0 \stackrel{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{S}}, w, P) :: (\Gamma, (x_{\mathsf{S}} : A_{\mathsf{S}}) \; ; \; \Delta)} \; \mathsf{proc}_{\mathsf{S}}$$

we get $\Gamma_0 \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}})$. By the induction hypothesis, $\Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}})$. Also, by Lemma 5, we get $\cdot \; ; \; \Gamma_0, (c : B_{\mathsf{S}}) \; ; \; \Delta'_{\mathsf{R}} \stackrel{q}{\vdash} P :: (x_{\mathsf{S}} : A_{\mathsf{S}})$. Applying the $\mathsf{proc}_{\mathsf{S}}$ rule back,

$$\frac{\begin{array}{c}(x_{\mathsf{S}} : A_{\mathsf{S}}) \in \Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \qquad (A_{\mathsf{S}}, A_{\mathsf{S}})\;\mathsf{esync} \qquad \Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta'_{\mathsf{R}}) \\ \cdot \; ; \; \Gamma_0, (c : B_{\mathsf{S}}) \; ; \; \Delta'_{\mathsf{R}} \stackrel{q}{\vdash} P :: (x_{\mathsf{S}} : A_{\mathsf{S}})\end{array}}{\Gamma_0, (c_{\mathsf{S}} : B_{\mathsf{S}}) \stackrel{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_{\mathsf{S}}, w, P) :: (\Gamma, (x_{\mathsf{S}} : A_{\mathsf{S}}) \; ; \; \Delta)} \; \mathsf{proc}_{\mathsf{S}}$$

- Case $(\mathsf{proc}_\mathsf{L})$: We have $\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_\mathsf{L}, w, P) :: (\Gamma, (x_\mathsf{S} : A_\mathsf{S}) \; ; \; \Delta, (x_\mathsf{L} : A_\mathsf{L}))$. Inverting the proc_L rule,

$$\frac{(x_\mathsf{S} : A_\mathsf{S}) \in \Gamma_0 \qquad (A_\mathsf{L}, A_\mathsf{S}) \text{ esync} \qquad \Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta') \qquad \cdot \; ; \; \Gamma_0 \; ; \; \Delta' \vdash^q P :: (x_\mathsf{L} : A_\mathsf{L})}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_\mathsf{L}, w, P) :: (\Gamma, (x_\mathsf{S} : A_\mathsf{S}) \; ; \; \Delta, (x_\mathsf{L} : A_\mathsf{L}))} \; \mathsf{proc}_\mathsf{L}$$

we get $\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta')$. Applying the induction hypothesis, we get $\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta')$. Using Lemma 5, we get $\cdot \; ; \; \Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta' \vdash^q P :: (x_\mathsf{L} : A_\mathsf{L})$. Applying the proc_L rule back,

$$\frac{(x_\mathsf{S} : A_\mathsf{S}) \in \Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \qquad (A_\mathsf{L}, A_\mathsf{S}) \text{ esync} \qquad \Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta') \quad \cdot \; ; \; \Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta' \vdash^q P :: (x_\mathsf{L} : A_\mathsf{L})}{\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_\mathsf{L}, w, P) :: (\Gamma, (x_\mathsf{S} : A_\mathsf{S}) \; ; \; \Delta, (x_\mathsf{L} : A_\mathsf{L}))} \; \mathsf{proc}_\mathsf{L}$$

- Case $(\mathsf{proc}_\mathsf{T})$: We have $\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_\mathsf{T}, w, P) :: (\Gamma \; ; \; \Delta, (x_\mathsf{T} : A_\mathsf{T}))$. Inverting the proc_T rule,

$$\frac{\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta') \qquad \cdot \; ; \; \Gamma_0 \; ; \; \Delta' \vdash^q P :: (x_\mathsf{T} : A_\mathsf{T})}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_\mathsf{T}, w, P) :: (\Gamma \; ; \; \Delta, (x_\mathsf{T} : A_\mathsf{T}))} \; \mathsf{proc}_\mathsf{T}$$

we get $\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta')$. By the induction hypothesis, we get $\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta')$. Also, using Lemma 5, we get $\cdot \; ; \; \Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta' \vdash^q P :: (x_\mathsf{T} : A_\mathsf{T})$ Applying the proc_T rule back,

$$\frac{\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta') \qquad \cdot \; ; \; \Gamma \; ; \; \Delta' \vdash^q P :: (x_\mathsf{T} : A_\mathsf{T})}{\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E+q+w}{\vDash} \Omega, \mathsf{proc}(x_\mathsf{T}, w, P) :: (\Gamma \; ; \; \Delta, (x_\mathsf{T} : A_\mathsf{T}))} \; \mathsf{proc}_\mathsf{T}$$

- Case (msg) : We have $\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{msg}(x_m, w, M) :: (\Gamma \; ; \; \Delta, (x_m : A))$. Inverting the msg rule,

$$\frac{\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta') \qquad \cdot \; ; \; \cdot \; ; \; \Delta' \vdash^q M :: (x_m : A)}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega, \mathsf{msg}(x_m, w, M) :: (\Gamma \; ; \; \Delta, (x_m : A))} \; \mathsf{msg}$$

$\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta')$. By the induction hypothesis, $\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E+q+w}{\vDash} \Omega, \mathsf{msg}(x_m, w, M) :: (\Gamma \; ; \; \Delta, (x_m : A))$. Applying the msg rule back,

$$\frac{\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta, \Delta') \qquad \cdot \; ; \; \cdot \; ; \; \Delta' \vdash^q M :: (x_m : A)}{\Gamma_0, (c_\mathsf{S} : B_\mathsf{S}) \overset{E+q+w}{\vDash} \Omega, \mathsf{msg}(x_m, w, M) :: (\Gamma \; ; \; \Delta, (x_m : A))} \; \mathsf{msg}$$

□

Lemma 5 (Process Weakening). *For a well-typed process* $\Gamma \; ; \; \Delta \vdash^q P :: (x_\mathsf{T} : A)$ *and for a shared channel* $c_\mathsf{S} : A_\mathsf{S} \notin \Gamma$*, we have* $\Gamma, (c_\mathsf{S} : A_\mathsf{S}) \; ; \; \Delta \vdash^q P :: (x_\mathsf{T} : A)$.

Proof. Analogous to Lemma 4. □

Lemma 6 (Permutation-Message). *Consider a well-typed configuration typed by the judgment* $\Gamma_0 \overset{E}{\vDash} \Omega_1,\ \mathsf{msg}(c_m, w, M), \Omega_2, \mathsf{proc}(d_k, w', P(c_m)) :: (\Gamma\ ;\ \Delta)$. *Then, the message can be moved right such that the configuration* $\Gamma_0 \overset{E}{\vDash} \Omega_1, \Omega_2, \mathsf{msg}(c_m, w, M), \mathsf{proc}(d_k, w', P(c_m)) :: (\Gamma\ ;\ \Delta)$ *is well-typed.*

Proof. We case analyze on the structure of the message.

- Case $(\otimes_n)$: We have $\Gamma_0 \overset{E}{\vDash} \Omega_1, \mathsf{msg}(c_m, w, \mathsf{send}\ c_m\ e_n\ ;\ c_m \leftarrow c_m^+), \Omega_2, \mathsf{proc}(d_k, w', P(c_m)) :: (\Gamma\ ;\ \Delta)$. First, we type the message

  $$\cdot\ ;\ \cdot\ ;\ (c_m^+ : B), (e_n : A) \vdash^{q} \mathsf{send}\ c_m\ e_n\ ;\ c_m \leftarrow c_m^+ :: (c_m : A \otimes_n B)$$

  Next, we invert the msg rule,

  $$\frac{\Gamma_0 \overset{E}{\vDash} \Omega_1 :: (\Gamma\ ;\ \Delta, (c_m^+ : B), (e_n : A)) \qquad \cdot\ ;\ \cdot\ ;\ (c_m^+ : B), (e_n : A) \vdash^{q} \mathsf{send}\ c_m\ e_n\ ;\ c_m \leftarrow c_m^+ :: (c_m : A \otimes_n B)}{\Gamma_0 \overset{E+q+w}{\vDash} \Omega_1, \mathsf{msg}(c_m, w, \mathsf{send}\ c_m\ e_n) :: (\Gamma\ ;\ \Delta, (c_m : A \otimes_n B))}\ \mathsf{msg}$$

  Since the channel c_m is only used by $\mathsf{proc}(d_k, w', P(c_m))$, we know that none of the processes or messages in Ω_2 can use it. Hence, we can move the message just left of the process $\mathsf{proc}(d_k, w', P(c_m))$.

□

Lemma 7 (Permutation-Process). *Consider a well-typed configuration typed by the judgment* $\Gamma_0 \overset{E}{\vDash} \Omega_1, \mathsf{proc}(c_m, w, P), \Omega_2, \mathsf{msg}(c_m^+, w', M(c_m)) :: (\Gamma\ ;\ \Delta)$. *Then, the process can be moved right such that the configuration* $\Gamma_0 \overset{E}{\vDash} \Omega_1, \Omega_2, \mathsf{proc}(c_m, w, P), \mathsf{msg}(c_m^+, w', M(c_m)) :: (\Gamma\ ;\ \Delta)$ *is well-typed.*

Proof. We case analyze on the structure of the message.

- Case $(\multimap_n)$: We have $\Gamma_0 \overset{E}{\vDash} \Omega_1, \mathsf{proc}(c_m, w, P), \Omega_2, \mathsf{msg}(c_m^+, w', \mathsf{send}\ c_m\ e_n\ ;\ c_m^+ \leftarrow c_m) :: (\Gamma\ ;\ \Delta)$. First, we type the message

  $$\cdot\ ;\ \cdot\ ;\ (e_n : A), (c_m : A \multimap_n B) \vdash^{q} \mathsf{send}\ c_m\ e_n\ ;\ c_m^+ \leftarrow c_m :: (c_m^+ : B)$$

  Since the message is the only provider of channel c_m offered by $\mathsf{proc}(c_m, w, P)$, we know that none of the processes in Ω_2 can depend on it. Thus, the process can be moved to the without affecting the invariant for any process in Ω_2.

□

Lemma 8 (Permutation-Acquire). *Consider a well-typed configuration typed by the judgment* $\Gamma_0 \overset{E}{\vDash} \Omega_1, \mathsf{proc}(c_m, w', a_{\mathsf{L}} \leftarrow \mathsf{acquire}\ a_{\mathsf{S}}\ ;\ Q), \Omega_2, \mathsf{proc}(a_{\mathsf{S}}, w, a_{\mathsf{L}} \leftarrow \mathsf{accept}\ a_{\mathsf{S}}\ ;\ P), \Omega_3 :: (\Gamma\ ;\ \Delta)$. *Then, the acquiring process can be moved right such that the configuration* $\Gamma_0 \overset{E}{\vDash} \Omega_1, \Omega_2, \mathsf{proc}(a_{\mathsf{S}}, w, a_{\mathsf{L}} \leftarrow \mathsf{accept}\ a_{\mathsf{S}}\ ;\ P), \mathsf{proc}(c_m, w', a_{\mathsf{L}} \leftarrow \mathsf{acquire}\ a_{\mathsf{S}}\ ;\ Q), \Omega_3 :: (\Gamma\ ;\ \Delta)$ *is well-typed.*

Proof. Due to independence, we know that $\mathsf{proc}(a_{\mathsf{S}}, w, a_{\mathsf{L}} \leftarrow \mathsf{accept}\ a_{\mathsf{S}}\ ;\ P)$ can only depend on any channels at mode S or R. On the other hand, m can only be T or L. In particular, the shared process cannot depend on channel c_m, thus the acquiring process can be moved to the right of the shared process. □

Lemma 9 (Permutation-Release). *Consider a well-typed configuration typed by the judgment* $\Gamma_0 \overset{E}{\vDash} \Omega_1, \mathsf{proc}(c_m, w', a_\mathsf{S} \leftarrow \mathsf{release}\ a_\mathsf{L}\ ;\ Q), \Omega_2, \mathsf{proc}(a_\mathsf{L}, w, a_\mathsf{S} \leftarrow \mathsf{detach}\ a_\mathsf{L}\ ;\ P), \Omega_3 :: (\Gamma\ ;\ \Delta)$. *Then, the releasing process can be moved right such that the configuration* $\Gamma_0 \overset{E}{\vDash} \Omega_1, \Omega_2, \mathsf{proc}(a_\mathsf{L}, w, a_\mathsf{S} \leftarrow \mathsf{detach}\ a_\mathsf{L}\ ;\ P), \mathsf{proc}(c_m, w', a_\mathsf{S} \leftarrow \mathsf{release}\ a_\mathsf{L}\ ;\ Q), \Omega_3 :: (\Gamma\ ;\ \Delta)$ *is well-typed.*

Proof. Due to independence, we know that $\mathsf{proc}(a_\mathsf{L}, w, a_\mathsf{S} \leftarrow \mathsf{detach}\ a_\mathsf{L}\ ;\ P)$ can only depend on any channels at mode S or R. On the other hand, m can only be T or L. In particular, the shared process cannot depend on channel c_m, thus the releasing process can be moved to the right of the detaching process. □

Lemma 10 (Shared-Substitution). *If the process* $\Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta \overset{q}{\vdash} P_{x_\mathsf{S}} :: (z_m : C)$ *is well-typed, then* $\Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta \overset{q}{\vdash} P_{b_\mathsf{S}} :: (z_m : C)$ *is also well-typed.*

Proof. We apply induction on the process typing judgment.

- Case $(\{\}E_{\mathsf{TT}})$:

$$\dfrac{r = p + q \qquad \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S}) \supseteq \overline{a_\mathsf{S} : A} \qquad \Delta = \overline{d : D} \quad\quad \Psi \Vdash^{p} M : \{A \leftarrow \overline{A}\ ;\ \overline{D}\}_\mathsf{T} \qquad \Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta', (y_\mathsf{T} : A) \overset{q}{\vdash} Q_{x_\mathsf{S}} :: (z_\mathsf{T} : C)}{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta, \Delta' \overset{r}{\vdash} y_\mathsf{T} \leftarrow M \leftarrow \overline{a_\mathsf{S}}\ ;\ \overline{d}\ ;\ Q_{x_\mathsf{S}} :: (z_\mathsf{T} : C)}\ \{\}E_{\mathsf{TT}}$$

By the induction hypothesis, $\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta', (y_\mathsf{T} : A) \overset{q}{\vdash} Q_{b_\mathsf{S}} :: (z_\mathsf{T} : C)$. We simply substitute b_S for x_S in $\overline{a_\mathsf{S} : A}$. Hence, $\Gamma, (b_\mathsf{S} : B_\mathsf{S}) \supseteq [b_\mathsf{S}/x_\mathsf{S}]\overline{a_\mathsf{S} : A}$. Applying the $\{\}E_{\mathsf{TT}}$ rule back

$$\dfrac{r = p + q \qquad \Gamma, (b_\mathsf{S} : B_\mathsf{S}) \supseteq [b_\mathsf{S}/x_\mathsf{S}]\overline{a_\mathsf{S} : A} \qquad \Delta = \overline{d : D} \quad\quad \Psi \Vdash^{p} M : \{A \leftarrow \overline{A}\ ;\ \overline{D}\}_\mathsf{T} \qquad \Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta', (y_\mathsf{T} : A) \overset{q}{\vdash} Q_{b_\mathsf{S}} :: (z_\mathsf{T} : C)}{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta, \Delta' \overset{r}{\vdash} y_\mathsf{T} \leftarrow M \leftarrow [b_\mathsf{S}/x_\mathsf{S}]\overline{a_\mathsf{S}}\ ;\ \overline{d}\ ;\ Q_{x_\mathsf{S}} :: (z_\mathsf{T} : C)}\ \{\}E_{\mathsf{TT}}$$

- Case (fwd) :

$$\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ (y_k : A) \overset{q}{\vdash} z_m \leftarrow y_k :: (z_m : A)$$

Here, the lemma holds trivially since x_S doesn't occur in P_{x_S}. Therefore, $P_{x_\mathsf{S}} = P_{b_\mathsf{S}}$ and

$$\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ (y_k : A) \overset{q}{\vdash} z_m \leftarrow y_k :: (z_m : A)$$

- Case $(\multimap_n R)$:

$$\dfrac{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta, (y_n : A) \overset{q}{\vdash} P_{x_\mathsf{S}} :: (z_m : B)}{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta \overset{q}{\vdash} y_n \leftarrow \mathsf{recv}\ z_m\ ;\ P_{x_\mathsf{S}} :: (z_m : A \multimap_n B)}\ \multimap_n R$$

By the induction hypothesis, $\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta, (y_n : A) \overset{q}{\vdash} P_{b_\mathsf{S}} :: (z_m : B)$. Applying the $\multimap R$ rule,

$$\dfrac{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta, (y_n : A) \overset{q}{\vdash} P_{b_\mathsf{S}} :: (z_m : B)}{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta \overset{q}{\vdash} y_n \leftarrow \mathsf{recv}\ z_m\ ;\ P_{b_\mathsf{S}} :: (z_m : A \multimap_n B)}\ \multimap_n R$$

- Case $(\multimap_n L)$:

$$\dfrac{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta, (y_k : B) \overset{q}{\vdash} Q_{x_\mathsf{S}} :: (z_m : C)}{\Psi\ ;\ \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S})\ ;\ \Delta, (w_n : A), (y_k : A \multimap B) \overset{q}{\vdash} \mathsf{send}\ y_k\ w_n\ ;\ Q_{x_\mathsf{S}} :: (z_m : C)}\ \multimap L$$

By the induction hypothesis, $\Psi \; ; \; \Gamma, (b_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta, (y_k : B) \vDash^q Q_{b_\mathsf{S}} :: (z_m : C)$. Applying the $\multimap_n L$ rule,

$$\frac{\Psi \; ; \; \Gamma, (b_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta, (y_k : B) \vDash^q Q_{b_\mathsf{S}} :: (z_m : C)}{\Psi \; ; \; \Gamma, (b_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta, (w_n : A), (y_k : A \multimap_n B) \vDash^q \mathsf{send}\; y_k\; w_n \; ; \; Q_{b_\mathsf{S}} :: (z_m : C)} \multimap_n L$$

- Case $(\uparrow^\mathsf{S}_\mathsf{L} L)$:

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

The lemma holds trivially since x_S doesn't occur in Q. Hence, $[b_\mathsf{S}/x_\mathsf{S}]Q = Q$. Applying the $\uparrow^\mathsf{S}_\mathsf{L} L$ rule,

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

- Case $(\downarrow^\mathsf{S}_\mathsf{L} L)$:

$$\frac{\Psi \; ; \; \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S}), (y_\mathsf{S} : A_\mathsf{S}) \; ; \; \Delta \vDash^q Q_{x_\mathsf{S}} :: (z_m : C)}{\Psi \; ; \; \Gamma, (b_\mathsf{S} : B_\mathsf{S}), (x_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta, (y_\mathsf{L} : \downarrow^\mathsf{S}_\mathsf{L} A_\mathsf{S}) \vDash^q y_\mathsf{S} \leftarrow \mathsf{release}\; y_\mathsf{L} \; ; \; Q_{x_\mathsf{S}} :: (z_m : C)} \downarrow^\mathsf{S}_\mathsf{L} L$$

By the induction hypothesis, $\Psi \; ; \; \Gamma, (b_\mathsf{S} : A_\mathsf{S}), (y_\mathsf{S} : A_\mathsf{S}) \; ; \; \Delta \vDash^q Q_{b_\mathsf{S}} :: (z_m : C)$. Applying the $\downarrow^\mathsf{S}_\mathsf{L} L$ rule,

$$\frac{\Psi \; ; \; \Gamma, (b_\mathsf{S} : A_\mathsf{S}), (y_\mathsf{S} : A_\mathsf{S}) \; ; \; \Delta \vDash^q Q_{b_\mathsf{S}} :: (z_m : C)}{\Psi \; ; \; \Gamma, (b_\mathsf{S} : B_\mathsf{S}) \; ; \; \Delta, (y_\mathsf{L} : \downarrow^\mathsf{S}_\mathsf{L} A_\mathsf{S}) \vDash^q y_\mathsf{S} \leftarrow \mathsf{release}\; y_\mathsf{L} \; ; \; Q_{b_\mathsf{S}} :: (z_m : C)} \downarrow^\mathsf{S}_\mathsf{L} L$$

□

Lemma 11 (Variable Substitution). *To substitute value for a variable from the functional context, we need the following two lemmas.*

- *If* V val *and* $\cdot \Vdash^p V : \tau$ *and* $\Psi, (x : \tau) \Vdash^q M : \sigma$*, then* $\Psi \Vdash^{p+q} [V/x]M : \sigma$.
- *If* V val *and* $\cdot \Vdash^p V : \tau$ *and* $\Psi, (x : \tau) \; ; \; \Gamma \; ; \; \Delta \Vdash^q P :: (c : A)$*, then* $\Psi \; ; \; \Gamma \; ; \; \Delta \Vdash^{p+q} [V/x]P :: (c : A)$

Theorem 3 (Expression Preservation). *If a well-typed expression* $\cdot \Vdash^q N : \tau$ *takes a step, i.e.,* $N \Downarrow V \mid \mu$*, then* V val *and* $q \geq \mu$ *and* $\cdot \Vdash^{q-\mu} V : \tau$.

Theorem 4 (Process Preservation). *Consider a closed well-formed and well-typed configuration* Ω *such that* $\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma \; ; \; \Delta)$*. If the configuration takes a step, i.e.* $\Omega \mapsto \Omega'$*, then there exist* Γ_0', Γ' *such that* $\Gamma_0' \overset{E}{\vDash} \Omega' :: (\Gamma' \; ; \; \Delta)$*, i.e., the resulting configuration is well-typed.*

Proof. We case analyze on the semantics.

- Case (internal) : $\Omega = \mathcal{D}, \mathsf{proc}(c_m, w, P[N])$ and $\Omega' = \mathcal{D}, \mathsf{proc}(c_m, w + \mu, P[V])$. We case analyze on $P[N]$.
  - Case ($\rightarrow$ send) : $P[N] = \mathsf{send}\; d_k\; N \; ; \; P$ and $P[V] = \mathsf{send}\; d_k\; V \; ; \; P$, where $N \Downarrow V \mid \mu$. Suppose, $\Gamma_0 \overset{E+r+w}{\vDash} \mathcal{D}, \mathsf{proc}(c_m, w, \mathsf{send}\; d_k\; N \; ; \; P) :: (\Gamma \; ; \; \Delta, (c_m : C))$. Inverting the proc_m

rule,

$$\dfrac{\Gamma_0 \overset{E}{\models} \mathcal{D} :: (\Gamma\ ;\ \Delta_1, (d_k : \tau \to A), \Delta) \qquad \dfrac{r = p + q \quad \cdot \Vdash^{p} N : \tau \quad \cdot\ ;\ \Gamma_0\ ;\ \Delta, (d_k : A) \vdash^{q} P :: (c_m : C)}{\cdot\ ;\ \Gamma_0\ ;\ \Delta_1, (d_k : \tau \to A) \vdash^{r} \mathsf{send}\ d_k\ N\ ;\ P :: (c_m : C)} \to L}{\Gamma_0 \overset{E+r+w}{\models} \mathcal{D}, \mathsf{proc}(c_m, w, \mathsf{send}\ d_k\ N\ ;\ P) :: (\Gamma\ ;\ \Delta, (c_m : C))} \mathsf{proc}_m$$

By Theorem 3, we get that $\cdot \Vdash^{p-\mu} V : \tau$. Finally, we apply the same derivation again to get

$$\dfrac{\Gamma_0 \overset{E}{\models} \mathcal{D} :: (\Gamma\ ;\ \Delta_1, (d_k : \tau \to A), \Delta) \qquad \dfrac{r' = p - \mu + q \quad \cdot \Vdash^{p-\mu} V : \tau \quad \cdot\ ;\ \Gamma_0\ ;\ \Delta, (d_k : A) \vdash^{q} P :: (c_m : C)}{\cdot\ ;\ \Gamma_0\ ;\ \Delta_1, (d_k : \tau \to A) \vdash^{r'} \mathsf{send}\ d_k\ V\ ;\ P :: (c_m : C)} \to L}{\Gamma_0 \overset{E+r'+w+\mu}{\models} \mathsf{proc}(c_m, w+\mu, \mathsf{send}\ d_k\ N\ ;\ P), \mathcal{D} :: (\Gamma\ ;\ \Delta, (c_m : C))} \mathsf{proc}_m$$

and the proof succeeds since $r' + w + \mu = p - \mu + q + w + \mu = p + q + w = r + w$.

- Case ($\times$send) : Analogous to $\to$ send.
- Case ($E_{\mathsf{S}m}$) : $\Omega = \mathcal{D}, \mathcal{D}, \mathsf{proc}(c_m, w, d_{\mathsf{S}} \leftarrow N \leftarrow \overline{a_{\mathsf{S}}}\ ;\ \overline{a_{\mathsf{R}}}\ ;\ Q)$ and $\Omega' = \mathcal{D}, \mathsf{proc}(c_m, w + \mu, d_{\mathsf{S}} \leftarrow V \leftarrow \overline{a_{\mathsf{S}}}\ ;\ \overline{a_{\mathsf{R}}}\ ;\ Q)$ where $N \Downarrow V \mid \mu$. Inverting the $\mathsf{proc}_{\mathsf{m}}$ rule,

$$\dfrac{\Gamma_0 \overset{E}{\models} \mathcal{D} :: (\Gamma\ ;\ \Delta, \Delta_1, \Delta_2) \qquad \dfrac{r = p + q \quad \Gamma_0 \supseteq \overline{a_{\mathsf{S}} : A} \quad \Delta_1 = \overline{a_{\mathsf{R}} : D} \quad \cdot \Vdash^{p} N : \{A_{\mathsf{S}} \leftarrow \overline{A}\ ;\ \overline{D}\}_{\mathsf{S}} \quad \cdot\ ;\ \Gamma_0, (d_{\mathsf{S}} : A_{\mathsf{S}})\ ;\ \Delta_2 \vdash^{q} Q :: (c_m : C)}{\cdot\ ;\ \Gamma_0\ ;\ \Delta_1, \Delta_2 \vdash^{r} d_{\mathsf{S}} \leftarrow N \leftarrow \overline{a_{\mathsf{S}}}\ ;\ \overline{a_{\mathsf{R}}}\ ;\ Q :: (c_m : C)} E_{\mathsf{S}m}}{\Gamma_0 \overset{E+r+w}{\models} \mathcal{D}, \mathsf{proc}(c_m, w, d_{\mathsf{S}} \leftarrow N \leftarrow \overline{a_{\mathsf{S}}}\ ;\ \overline{a_{\mathsf{R}}}\ ;\ Q) :: (\Gamma\ ;\ \Delta, c_m : C)} \mathsf{proc}_m$$

By Theorem 3, $\cdot \Vdash^{p-\mu} V : \{A_{\mathsf{S}} \leftarrow \overline{D}\}_{\mathsf{S}}$. Applying the same derivation back,

$$\dfrac{\Gamma_0 \overset{E}{\models} \mathcal{D} :: (\Gamma\ ;\ \Delta, \Delta_1, \Delta_2) \qquad \dfrac{r' = p - \mu + q \quad \Gamma_0 \supseteq \overline{a_{\mathsf{S}} : A} \quad \Delta_1 = \overline{a_{\mathsf{R}} : D} \quad \cdot \Vdash^{p-\mu} V : \{A_{\mathsf{S}} \leftarrow \overline{A}\ ;\ \overline{D}\}_{\mathsf{S}} \quad \cdot\ ;\ \Gamma_0, (d_{\mathsf{S}} : A_{\mathsf{S}})\ ;\ \Delta_2 \vdash^{q} Q :: (c_m : C)}{\cdot\ ;\ \Gamma_0\ ;\ \Delta_1, \Delta_2 \vdash^{r'} d_{\mathsf{S}} \leftarrow V \leftarrow \overline{a_{\mathsf{S}}}\ ;\ \overline{a_{\mathsf{R}}}\ ;\ Q :: (c_m : C)} E_{\mathsf{S}m}}{\Gamma_0 \overset{E+r'+w+\mu}{\models} \mathcal{D}, \mathsf{proc}(c_m, w + \mu, d_{\mathsf{S}} \leftarrow V \leftarrow \overline{a_{\mathsf{S}}}\ ;\ \overline{a_{\mathsf{R}}}\ ;\ Q) :: (\Gamma\ ;\ \Delta, c_m : C)} \mathsf{proc}_m$$

and the proof succeeds since $r' + w + \mu = p - \mu + q + w + \mu = p + q + w = r + w$.

- Case ($E_{\mathsf{R}m}, E_{\mathsf{TT}}$) : Analogous to $E_{\mathsf{S}m}$.

- Case $(\{\}E_{ST}) : \Omega = \mathcal{D}, \mathsf{proc}(d_T, w, x_S \leftarrow \{x'_S \leftarrow P_{x'_S,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} \leftarrow \overline{a} \;;\; \overline{b} \;;\; Q)$ and $\Omega' = \mathcal{D}, \mathsf{proc}(c_S, 0, P_{c_S,\overline{a},\overline{b}}), \mathsf{proc}(d_T, w, [c_S/x_S]Q)$. Inverting the proc_T rule,

$$
\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \;;\; \Delta, \Delta_1, \Delta_2) \qquad \dfrac{\dfrac{\Gamma_y = \overline{y : A} \qquad \Delta_z = \overline{z : D} \qquad \cdot \;;\; \Gamma_y \;;\; \Delta_z \vdash^{p} P_{x'_S,\overline{y},\overline{z}} :: (x'_S : A_S)}{\cdot \Vdash^{p} \{x'_S \leftarrow P_{x'_S,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} : \{A_S \leftarrow \overline{A} \;;\; \overline{D}\}_S}\{\}I_S \quad r = p + q \quad \Gamma_0 \supseteq \overline{a : A} \quad \Delta_1 = \overline{b : D} \quad (A_S, A_S)\ \mathsf{esync} \quad \cdot \;;\; \Gamma_0, (x_S : A_S) \;;\; \Delta_2 \vdash^{q} Q :: (d_T : A_T)}{\cdot \;;\; \Gamma_0 \;;\; \Delta_1, \Delta_2 \vdash^{r} x_S \leftarrow \{x'_S \leftarrow P_{x'_S,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} \leftarrow \overline{a} \;;\; \overline{b} \;;\; Q :: (d_T : A_T)}\{\}E_{SC}}{\Gamma_0 \overset{E+r+w}{\vDash} \mathcal{D}, \mathsf{proc}(d_T, w, x_S \leftarrow \{x'_S \leftarrow P_{x'_S,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} \leftarrow \overline{a} \;;\; \overline{b} \;;\; Q) :: (\Gamma \;;\; \Delta, (d_T : A_T))}\mathsf{proc}_T
$$

The premise for $\{\}I_S$ gives us $\cdot \;;\; \Gamma_y \;;\; \Delta_z \vdash^{p} P_{x'_S,\overline{y},\overline{z}} :: (x'_S : A_S)$, which by Lemma 2, gives us $\cdot \;;\; \Gamma_0 \;;\; \Delta_1 \vdash^{p} P_{c_S,\overline{a},\overline{b}} :: (c_S : A_S)$. Then, by Lemma 5, we get $\cdot \;;\; \Gamma_0, (c_S : A_S) \;;\; \Delta_1 \vdash^{p} P_{c_S,\overline{a},\overline{b}} :: (c_S : A_S)$ Similarly, we get $\cdot \;;\; \Gamma_0, (c_S : A_S) \;;\; \Delta_2 \vdash^{q} [c_S/x_S]Q :: (d_T : A_T)$. First, using Lemma 4, we get $\Gamma_0, (c_S : A_S) \overset{E}{\vDash} \mathcal{D} :: (\Gamma \;;\; \Delta, \Delta_1, \Delta_2)$. Next, apply the proc_S rule,

$$
\dfrac{\Gamma_0, (c_S : A_S) \overset{E}{\vDash} \mathcal{D} :: (\Gamma \;;\; \Delta, \Delta_1, \Delta_2) \qquad \cdot \;;\; \Gamma_0, (c_S : A_S) \;;\; \Delta_1 \vdash^{p} P_{c_S,\overline{a},\overline{b}} :: (c_S : A_S)}{\Gamma_0, (c_S : A_S) \overset{E+p+0}{\vDash} \mathcal{D}, \mathsf{proc}(c_S, 0, P_{c_S,\overline{a},\overline{b}}) :: (\Gamma, (c_S : A_S) \;;\; \Delta, \Delta_2)}\mathsf{proc}_S
$$

Call this new configuration $\mathcal{D}'$. Now, apply the proc_T rule.

$$
\dfrac{\Gamma_0, (c_S : A_S) \overset{E+p+0}{\vDash} \mathcal{D}' :: (\Gamma, (c_S : A_S) \;;\; \Delta, \Delta_2) \qquad \cdot \;;\; \Gamma, (c_S : A_S) \;;\; \Delta_2 \vdash^{q} [c_S/x_S]Q :: (d_T : A_T)}{\Gamma_0, (c_S : A_S) \overset{E+p+q+w}{\vDash} \mathcal{D}', \mathsf{proc}(d_T, w, [c_S/x_S]Q) :: (\Gamma, (c_S : A_S) \;;\; \Delta, (d_T : A_T))}\mathsf{proc}_T
$$

where $E + p + q + w = E + r + w$ since $r = p + q$. Hence, in this case $\Gamma'_0 = \Gamma_0, (c_S : A_S)$ and $\Gamma' = \Gamma, (c_S : A_S)$.

- Case $(\{\}E_{TT}) : \Omega = \mathcal{D}, \mathsf{proc}(d_T, w, x_T \leftarrow \{x'_T \leftarrow P_{x'_T,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} \leftarrow a_S \;;\; d \;;\; Q)$ and $\Omega' = \mathcal{D}, \mathsf{proc}(c_T, 0, P_{c_T,\overline{a_S},\overline{d}}), \mathsf{proc}(d_T, w, [c_T/x_T]Q)$. Inverting the proc_T rule

$$
\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \;;\; \Delta, \Delta_1, \Delta_2) \qquad \dfrac{\dfrac{\Gamma_y = \overline{y : A} \qquad \Delta_z = \overline{z : D} \qquad \cdot \;;\; \Gamma_y \;;\; \Delta_z \vdash^{p} P_{x'_T,\overline{y},\overline{z}} :: (x'_T : A)}{\cdot \Vdash^{p} \{x'_T \leftarrow P_{x'_T,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} : \{A \leftarrow \overline{A} \;;\; \overline{D}\}_T}\{\}I_T \quad r = p + q \quad \Gamma_0 \supseteq \overline{a_S : A} \quad \Delta_1 = \overline{d : D} \quad \cdot \;;\; \Gamma_0 \;;\; \Delta_2, (x_T : A) \vdash^{q} Q :: (d_T : C)}{\cdot \;;\; \Gamma_0 \;;\; \Delta_1, \Delta_2 \vdash^{r} x_T \leftarrow \{x'_T \leftarrow P_{x'_T,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} \leftarrow \overline{a_S} \;;\; \overline{d} \;;\; Q :: (d_T : C)}\{\}E_{TT}}{\Gamma_0 \overset{E+r+w}{\vDash} \mathcal{D}, \mathsf{proc}(d_T, w, x_T \leftarrow \{x'_T \leftarrow P_{x'_T,\overline{y},\overline{z}} \leftarrow \overline{y} \;;\; \overline{z}\} \leftarrow \overline{a_S} \;;\; \overline{d} \;;\; Q) :: (\Gamma \;;\; \Delta, (d_T : C))}\mathsf{proc}_T
$$

We contract all multiple occurrences of the same channel in $\overline{a_S : A}$. Let the resulting vector be $\Gamma' = \overline{a'_S : A'}$. We know, by Lemma 10 that $\cdot \;;\; \Gamma' \;;\; \Delta' \vdash^{p} P_{x'_T,\overline{a'_S},\overline{z}} :: (x'_T : A)$ is well-typed. Next, by Lemma 2, we get $\Gamma' \;;\; \Delta_1 \vdash^{p} P_{c_T,\overline{a'_S},\overline{d}} :: (c_T : A)$. Finally, we weaken Γ' using Lemma 5 to get $\cdot \;;\; \Gamma_0 \;;\; \Delta_1 \vdash^{p} P_{c_T,\overline{a'_S},\overline{d}} :: (c_T : A)$. Also, note that since $\overline{a'_S}$ is a refinement of $\overline{a_S}$ by eliminating duplicates, $P_{c_T, \overline{a'_S}, \overline{d}} = P_{c_T, \overline{a_S}, \overline{d}}$. Hence, we apply the proc_T rule,

$$\frac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \ ; \ \Delta, \Delta_1, \Delta_2) \qquad \cdot \ ; \ \Gamma_0 \ ; \ \Delta_1 \overset{p}{\vdash} P_{c_T, \overline{a_S}, \overline{d}} :: (c_T : A)}{\Gamma_0 \overset{E+p+0}{\vDash} \mathcal{D}, \mathsf{proc}(c_T, 0, P_{c_T, \overline{a_S}, \overline{d}}) :: (\Gamma \ ; \ \Delta, \Delta_2, (c_T : A))} \ \mathsf{proc}_T$$

Call this new configuration $\mathcal{D}'$. Applying renaming using Lemma 2, we get $\cdot \ ; \ \Gamma_0 \ ; \ \Delta_2, (c_T : A) \overset{q}{\vdash} [c_T/x_T]Q :: (d_T : C)$. Again, applying the proc_T rule, we get

$$\frac{\Gamma_0 \overset{E+p+0}{\vDash} \mathcal{D}' :: (\Gamma \ ; \ \Delta, \Delta_2, (c_T : A)) \qquad \cdot \ ; \ \Gamma_0 \ ; \ \Delta_2, (c_T : A) \overset{q}{\vdash} [c_T/x_T]Q :: (d_T : C)}{\Gamma_0 \overset{E+p+q+w}{\vDash} \mathcal{D}', \mathsf{proc}(d_T, w, [c_T/x_T]Q) :: (\Gamma \ ; \ \Delta, (d_T : C))} \ \mathsf{proc}_T$$

where $E + p + q + w = E + r + w$ since $r = p + q$.

- Case $(\mathsf{fwd}^+) : \Omega = \mathcal{D}, \mathsf{msg}(d_k, w', M), \mathsf{proc}(c_m, w, c_m \leftarrow d_k)$ and $\Omega' = \mathsf{msg}(c_m, w + w', [c_m/d_k]M)$. First, inverting the msg rule,

$$\frac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Omega \ ; \ \Delta, \Delta_1) \qquad \cdot \ ; \ \cdot \ ; \ \Delta_1 \overset{q}{\vdash} M :: (d_k : A)}{\Gamma_0 \overset{E+q+w'}{\vDash} \mathcal{D}, \mathsf{msg}(d_k, w', M) :: (\Gamma \ ; \ \Delta, (d_k : A))} \ \mathsf{msg}$$

Call this new configuration $\mathcal{D}'$. Next, inverting the proc_m rule

$$\frac{\Gamma_0 \overset{E+q+w'}{\vDash} \mathcal{D}' :: (\Gamma \ ; \ \Delta, (d_k : A)) \qquad \cdot \ ; \ \Gamma_0 \ ; \ (d_k : A) \overset{0}{\vdash} c_m \leftarrow d_k :: (c_m : A)}{\Gamma_0 \overset{E+q+w'+0+w}{\vDash} \mathcal{D}', \mathsf{proc}(c_m, w, c_m \leftarrow d_k) :: (\Gamma \ ; \ \Delta, (c_m : A))} \ \mathsf{proc}_m$$

Using Lemma 2, we get $\cdot \ ; \ \cdot \ ; \ \Delta_1 \overset{q}{\vdash} [c_m/d_k]M :: (c_m : A)$. Applying the msg rule,

$$\frac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Omega \ ; \ \Delta, \Delta_1) \qquad \cdot \ ; \ \cdot \ ; \ \Delta_1 \overset{q}{\vdash} [c_m/d_k]M :: (c_m : A)}{\Gamma_0 \overset{E+q+w'+w}{\vDash} \mathcal{D}, \mathsf{msg}(c_m, w', [c_m/d_k]M) :: (\Gamma \ ; \ \Delta, (c_m : A))} \ \mathsf{msg}$$

- Case $(\mathsf{fwd}^-) : \Omega = \mathcal{D}, \mathsf{proc}(c_m, w, c_m \leftarrow d_k), \mathsf{msg}(e_l, w', M(c_m))$ and $\Omega' = \mathsf{msg}(e_l, w + w', M(d_k))$. First, inverting on the proc_m rule

$$\frac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \ ; \ \Delta, \Delta_1, (d_k : A)) \qquad \cdot \ ; \ \Gamma_0 \ ; \ (d_k : A) \overset{0}{\vdash} c_m \leftarrow d_k :: (c_m : A)}{\Gamma_0 \overset{E+0+w}{\vDash} \mathcal{D}, \mathsf{proc}(c_m, w, c_m \leftarrow d_k) :: (\Gamma \ ; \ \Delta, \Delta_1, (c_m : A))} \ \mathsf{proc}_m$$

Call this new configuration $\mathcal{D}'$. Next, inverting on the msg rule,

$$\frac{\Gamma_0 \overset{E+w}{\vDash} \mathcal{D}' :: (\Gamma \ ; \ \Delta, \Delta_1, (c_m : A)) \qquad \cdot \ ; \ \cdot \ ; \ \Delta_1, (c_m : A) \overset{q}{\vdash} M(c_m) :: (e_l : C)}{\Gamma_0 \overset{E+w+q+w'}{\vDash} \mathcal{D}', \mathsf{msg}(e_l, w', M(c_m)) :: (\Gamma \ ; \ \Delta, (e_l : C))} \ \mathsf{msg}$$

Using Lemma 2, we get $\cdot \ ; \ \cdot \ ; \ \Delta_1, (d_k : A) \overset{q}{\vdash} M(d_k) :: (e_l : C)$. Reapplying the msg rule,

$$\frac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \ ; \ \Delta, \Delta_1, (d_k : A)) \qquad \cdot \ ; \ \cdot \ ; \ \Delta_1, (d_k : A) \overset{q}{\vdash} M(d_k) :: (e_l : C)}{\Gamma_0 \overset{E+q+w+w'}{\vDash} \mathcal{D}, \mathsf{msg}(e_l, w', M(d_k)) :: (\Gamma \ ; \ \Delta, (e_l : C))} \ \mathsf{msg}$$

- Case $(\oplus C_s) : \Omega = \mathcal{D}, \mathsf{proc}(c_m, w, c_m.\ell \; ; \; P)$ and $\Omega' = \mathcal{D}, \mathsf{proc}(c_m^+, w, [c_m^+/c_m]P), \mathsf{msg}(c_m, 0, c_m.\ell \; ; \; c_m \leftarrow c_m^+)$. First, inverting on the proc_m rule,

$$\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1) \qquad \dfrac{\cdot \; ; \; \Gamma_0 \; ; \; \Delta_1 \vdash^{q} P :: (c_m : A_\ell)}{\cdot \; ; \; \Gamma_0 \; ; \; \Delta_1 \vdash^{q} c_m.\ell \; ; \; P :: (c_m : \oplus\{l : A_l\}_{l \in L})} \oplus R}{\Gamma_0 \overset{E+q+w}{\vDash} \mathcal{D}, \mathsf{proc}(c_m, w, c_m.\ell \; ; \; P) :: (\Gamma \; ; \; \Delta, (c_m : \oplus\{l : A_l\}_{l \in L}))} \mathsf{proc}_m$$

Using Lemma 2, we get $\cdot \; ; \; \Gamma_0 \; ; \; \Delta_1 \vdash^{q} [c_m^+/c_m]P :: (c_m^+ : A_\ell)$. Now, applying the proc_m rule,

$$\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1) \qquad \cdot \; ; \; \Gamma_0 \; ; \; \Delta_1 \vdash^{q} [c_m^+/c_m]P :: (c_m^+ : A_\ell)}{\Gamma_0 \overset{E+q+w}{\vDash} \mathcal{D}, \mathsf{proc}(c_m, w, c_m.\ell \; ; \; P) :: (\Gamma \; ; \; \Delta, (c_m^+ : A_\ell))} \mathsf{proc}_m$$

Next, typing the message

$$\cdot \; ; \; \cdot \; ; \; (c_m^+ : A_\ell) \vdash^{0} c_m.\ell \; ; \; c_m \leftarrow c_m^+ :: (c_m : \oplus\{l : A_l\}_{l \in L})$$

Call this new configuration $\mathcal{D}'$. Applying the msg rule next

$$\dfrac{\Gamma_0 \overset{E+q+w}{\vDash} \mathcal{D}' :: (\Gamma \; ; \; \Delta, (c_m : A_\ell)) \qquad \cdot \; ; \; \cdot \; ; \; (c_m^+ : A_\ell) \vdash^{0} c_m.\ell \; ; \; c_m \leftarrow c_m^+ :: (c_m : \oplus\{l : A_l\}_{l \in L})}{\Gamma_0 \overset{E+q+w}{\vDash} \mathcal{D}', \mathsf{msg}(c_m, 0, c_m.\ell \; ; \; c_m \leftarrow c_m^+) :: (\Gamma \; ; \; \Delta, (c_m : \oplus\{l : A_l\}_{l \in L}))} \mathsf{msg}$$

- Case $(\oplus C_r) : \Omega = \mathcal{D}, \mathsf{msg}(c_m, w, c_m.\ell \; ; \; c_m \leftarrow c_m^+), \mathsf{proc}(d_k, w', \mathsf{case}\ c_m\ (l \Rightarrow Q_l)_{l \in L})$ and $\Omega' = \mathcal{D}, \mathsf{proc}(d_k, w + w', [c_m^+/c_m]Q_\ell)$. First, inverting the msg rule,

$$\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1, (c_m^+ : A_\ell)) \qquad \cdot \; ; \; \cdot \; ; \; (c_m^+ : A_\ell) \vdash^{0} c_m.\ell \; ; \; c_m \leftarrow c_m^+ :: (c_m : \oplus\{l : A_l\}_{l \in L})}{\Gamma_0 \overset{E+0+w}{\vDash} \mathcal{D}, \mathsf{msg}(c_m, w, c_m.\ell \; ; \; c_m \leftarrow c_m^+) :: (\Gamma \; ; \; \Delta, \Delta_1, (c_m : \oplus\{l : A_l\}_{l \in L}))} \mathsf{msg}$$

Call this new configuration $\mathcal{D}'$. Next, inverting the proc_m rule,

$$\dfrac{\Gamma_0 \overset{E+0+w}{\vDash} \mathcal{D}' :: (\Gamma \; ; \; \Delta, \Delta_1, (c_m : \oplus\{l : A_l\}_{l \in L})) \qquad \dfrac{\cdot \; ; \; \Gamma_0 \; ; \; \Delta_1, (c_m : A_l) \vdash^{q} Q_l :: (d_k : C)}{\cdot \; ; \; \Gamma_0 \; ; \; \Delta_1, (c_m : \oplus\{l : A_l\}_{l \in L}) \vdash^{q} \mathsf{case}\ c_m\ (l \Rightarrow Q_l)_{l \in L} :: (d_k : C)} \oplus R}{\Gamma_0 \overset{E+0+w+q+w'}{\vDash} \mathcal{D}', \mathsf{proc}(d_k, w', \mathsf{case}\ c_m\ (l \Rightarrow Q_l)_{l \in L}) :: (\Gamma \; ; \; \Delta, (d_k : C))} \mathsf{proc}_m$$

Renaming using Lemma 2, we get $\cdot \; ; \; \Gamma_0 \; ; \; \Delta_1, (c_m^+ : A_\ell) \vdash^{q} [c_m^+/c_m]Q_\ell :: (d_k : C)$. Next, we apply the proc_m rule

$$\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1, (c_m^+ : A_\ell)) \qquad \cdot \; ; \; \Gamma_0 \; ; \; \Delta_1, (c_m^+ : A_\ell) \vdash^{q} [c_m^+/c_m]Q_\ell :: (d_k : C)}{\Gamma_0 \overset{E+q+w+w'}{\vDash} \mathcal{D}', \mathsf{proc}(d_k, w + w', [c_m^+/c_m]Q_\ell) :: (\Gamma \; ; \; \Delta, (d_k : C))} \mathsf{proc}_m$$

- Case $(\multimap_n C_s) : \Omega = \mathcal{D}, \mathsf{proc}(d_k, w, \mathsf{send}\ c_m\ e_n \; ; \; P)$ and $\Omega' = \mathcal{D}, \mathsf{msg}(c_m^+, 0, \mathsf{send}\ c_m\ e_n \; ; \; c_m^+ \leftarrow c_m), \mathsf{proc}(d_k, w, [c_m^+/c_m]P)$. First, we invert the

proc_m rule,

$$\dfrac{\dfrac{\begin{array}{c}\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1, (e_{\mathsf{R}} : A), (c_m : A \multimap B)) \\ \cdot \; ; \; \Gamma \; ; \; \Delta_1, (c_m : B) \overset{q}{\vdash} P :: (d_k : C)\end{array}}{\cdot \; ; \; \Gamma \; ; \; \Delta_1, (e_{\mathsf{R}} : A), (c_m : A \multimap B) \overset{q}{\vdash} \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; P :: (d_k : C)} \multimap L}{\Gamma_0 \overset{E+q+w}{\vDash} \mathcal{D}, \mathsf{proc}(d_k, w, \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; P) :: (\Gamma \; ; \; \Delta, (d_k : C))} \mathsf{proc}_m$$

Using renaming (Lemma 2), we get $\Gamma \; ; \; \Delta_1, (c_m^+ : B) \overset{q}{\vdash} [c_m^+/c_m]P :: (d_k : C)$. Next, we type the message

$$\cdot \; ; \; \Gamma \; ; \; (e_{\mathsf{R}} : A), (c_m : A \multimap B) \overset{0}{\vdash} \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; c_m^+ \leftarrow c_m :: (c_m^+ : B)$$

Next, we apply the msg rule,

$$\dfrac{\begin{array}{c}\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1, (e_{\mathsf{R}} : A), (c_m : A \multimap B)) \\ \cdot \; ; \; \Gamma \; ; \; (e_{\mathsf{R}} : A), (c_m : A \multimap B) \overset{0}{\vdash} \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; c_m^+ \leftarrow c_m :: (c_m^+ : B)\end{array}}{\Gamma_0 \overset{E}{\vDash} \mathcal{D}, \mathsf{msg}(c_m^+, 0, \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; c_m^+ \leftarrow c_m) :: (\Gamma \; ; \; \Delta, \Delta_1, (c_m^+ : B))} \mathsf{msg}$$

Call this new configuration $\mathcal{D}'$. Next, we apply the proc_m rule

$$\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D}' :: (\Gamma \; ; \; \Delta, \Delta_1, (c_m^+ : B)) \qquad \cdot \; ; \; \Gamma \; ; \; \Delta_1, (c_m^+ : B) \overset{q}{\vdash} [c_m^+/c_m]P :: (d_k : C)}{\Gamma_0 \overset{E+q+w}{\vDash} \mathcal{D}', \mathsf{proc}(d_k, w, [c_m^+/c_m]P) :: (\Gamma \; ; \; \Delta, (d_k : C))} \mathsf{proc}_m$$

- Case $(\multimap C_r) : \Omega = \mathcal{D}, \mathsf{proc}(c_m, w', x_{\mathsf{R}} \leftarrow \mathsf{recv}\; c_m \; ; \; Q), \mathsf{msg}(c_m^+, w, \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; c_m^+ \leftarrow c_m)$ and $\Omega' = \mathcal{D}, \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m][e_{\mathsf{R}}/x_{\mathsf{R}}]Q)$. First, inverting the proc_m rule,

$$\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1, (e_{\mathsf{R}} : A)) \qquad \dfrac{\cdot \; ; \; \Gamma \; ; \; \Delta_1, (x_{\mathsf{R}} : A) \overset{q}{\vdash} Q :: (c_m : B)}{\cdot \; ; \; \Gamma \; ; \; \Delta_1 \overset{q}{\vdash} x_{\mathsf{R}} \leftarrow \mathsf{recv}\; c_m \; ; \; Q :: (c_m : A \multimap B)} \multimap R}{\Gamma_0 \overset{E+q+w'}{\vDash} \mathcal{D}, \mathsf{proc}(c_m, w', x_{\mathsf{R}} \leftarrow \mathsf{recv}\; c_m \; ; \; Q) :: (\Gamma \; ; \; \Delta, (e_{\mathsf{R}} : A), (c_m : A \multimap B))} \mathsf{proc}_m$$

Call this new configuration $\mathcal{D}'$. Next, we type the message.

$$\cdot \; ; \; \Gamma \; ; \; (e_{\mathsf{R}} : A), (c_m : A \multimap B) \overset{0}{\vdash} \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; c_m^+ \leftarrow c_m :: (c_m^+ : B)$$

Inverting the msg rule,

$$\dfrac{\begin{array}{c}\Gamma_0 \overset{E+q+w'}{\vDash} \mathcal{D}' :: (\Gamma \; ; \; \Delta, (e_{\mathsf{R}} : A), (c_m : A \multimap B)) \\ \cdot \; ; \; \Gamma \; ; \; (e_{\mathsf{R}} : A), (c_m : A \multimap B) \overset{0}{\vdash} \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; c_m^+ \leftarrow c_m :: (c_m^+ : B)\end{array}}{\Gamma_0 \overset{E+q+w'+0+w'}{\vDash} \mathcal{D}', \mathsf{msg}(c_m^+, w, \mathsf{send}\; c_m\; e_{\mathsf{R}} \; ; \; c_m^+ \leftarrow c_m) :: (\Gamma \; ; \; \Delta, (c_m^+ : B))} \mathsf{msg}$$

By renaming using Lemma 2, $\cdot \; ; \; \Gamma \; ; \; \Delta_1, (e_{\mathsf{R}} : A) \overset{q}{\vdash} [c_m^+/c_m][e_{\mathsf{R}}/x_{\mathsf{R}}]Q :: (c_m^+ : B)$. Now, applying the proc_m rule,

$$\dfrac{\Gamma_0 \overset{E}{\vDash} \mathcal{D} :: (\Gamma \; ; \; \Delta, \Delta_1, (e_{\mathsf{R}} : A)) \qquad \cdot \; ; \; \Gamma \; ; \; \Delta_1, (e_{\mathsf{R}} : A) \overset{q}{\vdash} [c_m^+/c_m][e_{\mathsf{R}}/x_{\mathsf{R}}]Q :: (c_m^+ : B)}{\Gamma_0 \overset{E+q+w'}{\vDash} \mathcal{D}, \mathsf{proc}(c_m^+, w + w', [c_m^+/c_m][e_{\mathsf{R}}/x_{\mathsf{R}}]Q) :: (\Gamma \; ; \; \Delta, (c_m^+ : B))} \mathsf{proc}_m$$

- Case $(\uparrow_{L}^{S} C) : \Omega = \mathcal{D}_1, \mathsf{proc}(a_S, w', x_L \leftarrow \mathsf{accept}\ a_S\ ;\ P_{x_L}), \mathsf{proc}(c_m, w, x_L \leftarrow \mathsf{acquire}\ a_S\ ;\ Q_{x_L})$ and $\Omega' = \mathcal{D}_1, \mathsf{proc}(a_L, w', P_{a_L}), \mathsf{proc}(c_m, w, Q_{a_L})$. Applying the proc_S rule first,

$$\dfrac{\begin{array}{c}(a_S : \uparrow_{L}^{S} A_L) \in \Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \\ (\uparrow_{L}^{S} A_L, \uparrow_{L}^{S} A_L)\ \mathsf{esync} \quad \Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \overset{E}{\vDash} \mathcal{D}_1 :: (\Gamma\ ;\ \Delta, \Delta_1, \Delta_2) \quad \mathcal{E}\end{array}}{\Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \overset{E+p+w'}{\vDash} \mathcal{D}_1, \mathsf{proc}(a_S, w', x_L \leftarrow \mathsf{accept}\ a_S\ ;\ P_{x_L}) :: (\Gamma, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta, \Delta_2)}\ \mathsf{proc}_S$$

where $\mathcal{E}$ is

$$\dfrac{\cdot\ ;\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta_1 \vdash^{p} P_{x_L} :: (x_L : A_L)}{\cdot\ ;\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta_1 \vdash^{p} x_L \leftarrow \mathsf{accept}\ a_S\ ;\ P_{x_L} :: (a_S : \uparrow_{L}^{S} A_L)}\ \uparrow_{L}^{S} R$$

Call this new configuration $\mathcal{D}_1'$. Applying the proc_m rule next,

$$\dfrac{\begin{array}{c}\Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \overset{E'}{\vDash} \mathcal{D}_1' :: (\Gamma, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta, \Delta_2) \\ \dfrac{\cdot\ ;\ \Gamma_0\ ;\ \Delta_2, (x_L : A_L) \vdash^{q} Q_{x_L} :: (c_m : C)}{\cdot\ ;\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta_2 \vdash^{q} x_L \leftarrow \mathsf{acquire}\ a_S\ ;\ Q_{x_L} :: (c_m : C)}\ \uparrow_{L}^{S} L\end{array}}{\Gamma_0 \overset{E'+q+w}{\vDash} \mathcal{D}_1', \mathsf{proc}(c_m, w, x_L \leftarrow \mathsf{acquire}\ a_S\ ;\ Q_{x_L}) :: (\Gamma, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta, (c_m : C))}\ \mathsf{proc}_m$$

From the first premise, we get by Lemma 2, $\cdot\ ;\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta_1 \vdash^{p} P_{a_L} :: (a_L : A_L)$ while from the second premise, we get by Lemma 2 and Lemma 5, $\cdot\ ;\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta_2, (a_L : A_L) \vdash^{q} Q_{a_L} :: (c_m : C)$. Reapplying the proc_L rule,

$$\dfrac{\begin{array}{c}(a_S : \uparrow_{L}^{S} A_L) \in \Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \qquad (A_L, \uparrow_{L}^{S} A_L)\ \mathsf{esync} \\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \overset{E}{\vDash} \mathcal{D}_1 :: (\Gamma\ ;\ \Delta, \Delta_1, \Delta_2) \quad \cdot\ ;\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta_1 \vdash^{p} P_{a_L} :: (a_L : A_L)\end{array}}{\Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \overset{E+p+w'}{\vDash} \mathcal{D}_1, \mathsf{proc}(a_L, w', P_{a_L}) :: (\Gamma, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta, \Delta_2, (a_L : A_L))}\ \mathsf{proc}_L$$

Call this new configuration $\mathcal{D}_1''$. Reapplying the proc_m rule,

$$\dfrac{\begin{array}{c}\Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \overset{E'}{\vDash} \mathcal{D}_1'' :: (\Gamma, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta, \Delta_2, (a_L : A_L)) \\ \cdot\ ;\ \Gamma_0, (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta_2, (a_L : A_L) \vdash^{q} Q_{a_L} :: (c_m : C)\end{array}}{\Gamma_0, (a_S : \uparrow_{L}^{S} A_L) \overset{E'+q+w}{\vDash} \mathcal{D}_1'', \mathsf{proc}(c_m, w, Q_{a_L}) :: (\Gamma', (a_S : \uparrow_{L}^{S} A_L)\ ;\ \Delta', (c_m : C))}\ \mathsf{proc}_m$$

- Case $(\downarrow_{L}^{S} C) : \Omega = \mathcal{D}_1, \mathsf{proc}(a_L, w', x_S \leftarrow \mathsf{detach}\ a_L\ ;\ P_{x_S}), \mathsf{proc}(c_T, w, x_S \leftarrow \mathsf{release}\ a_L\ ;\ Q_{x_S})$ and $\Omega' = \mathcal{D}_1, \mathsf{proc}(a_S, w', P_{a_S}), \mathsf{proc}(c_L, w, Q_{a_S})$. Applying the proc_L rule first,

$$\dfrac{(a_S : A_S) \in \Gamma_0 \qquad (\downarrow_{L}^{S} A_S, A_S)\ \mathsf{esync} \qquad \Gamma_0 \overset{E}{\vDash} \mathcal{D}_1 :: (\Gamma\ ;\ \Delta, \Delta_1, \Delta_2) \qquad \mathcal{E}}{\Gamma_0 \overset{E+p+w'}{\vDash} \mathcal{D}_1, \mathsf{proc}(a_L, w', x_S \leftarrow \mathsf{detach}\ a_L\ ;\ P_{x_S}) :: (\Gamma, (a_S : A_S)\ ;\ \Delta, \Delta_2, (a_L : \downarrow_{L}^{S} A_S))}\ \mathsf{proc}_L$$

where $\mathcal{E}$ is

$$\dfrac{\cdot\ ;\ \Gamma_0\ ;\ \Delta_1 \vdash^{p} P_{x_S} :: (x_S : A_S)}{\cdot\ ;\ \Gamma_0\ ;\ \Delta_1 \vdash^{p} x_S \leftarrow \mathsf{detach}\ a_L\ ;\ P_{x_S} :: (a_L : \downarrow_{L}^{S} A_S)}\ \downarrow_{L}^{S} R$$

Call this configuration $\mathcal{D}_1'$. Applying the proc_m rule,

$$\dfrac{\dfrac{\cdot \; ; \; \Gamma_0, (x_S : A_S) \; ; \; \Delta_2 \vDash^q Q_{x_S} :: (c_m : C)}{\cdot \; ; \; \Gamma_0 \; ; \; \Delta_2, (a_L : \downarrow_L^S A_S) \vDash^q x_S \leftarrow \text{release } a_L \; ; \; Q_{x_S} :: (c_m : C)} \downarrow_L^S L \qquad \Gamma_0 \overset{E'}{\vDash} \mathcal{D}_1' :: (\Gamma, (a_S : A_S) \; ; \; \Delta, \Delta_2, (a_L : \downarrow_L^S A_S))}{\Gamma_0 \overset{E'+q+w}{\vDash} \mathcal{D}_1', \text{proc}(c_T, w, x_S \leftarrow \text{release } a_L \; ; \; Q_{x_S}) :: (\Gamma, (a_S : A_S) \; ; \; \Delta, (c_m : C))} \text{proc}_m$$

From the first premise, we get by Lemma 2, $\cdot \; ; \; \Gamma_0 \; ; \; \Delta_1 \vDash^p P_{a_S} :: (a_S : A_S)$. From the second premise, by Lemma 10 (contracting $a_S : A_S$ and $x_S : A_S$), we get $\cdot \; ; \; \Gamma_0 \; ; \; \Delta_2 \vDash^q Q_{a_S} :: (c_m : C)$. Finally, applying the proc_S rule,

$$\dfrac{(a_S : A_S) \in \Gamma_0 \qquad (A_S, A_S) \text{ esync} \qquad \Gamma_0 \overset{E}{\vDash} \mathcal{D}_1 :: (\Gamma \; ; \; \Delta, \Delta_1, \Delta_2) \qquad \cdot \; ; \; \Gamma_0 \; ; \; \Delta_1 \vDash^p P_{a_S} :: (a_S : A_S)}{\Gamma_0 \overset{E+p+w'}{\vDash} \mathcal{D}_1, \text{proc}(a_S, w', P_{a_S}) :: (\Gamma, (a_S : A_S) \; ; \; \Delta, \Delta_2)} \text{proc}_S$$

Call this new configuration $\mathcal{D}_1''$. Applying the proc_m rule,

$$\dfrac{\Gamma_0 \overset{E'}{\vDash} \mathcal{D}_1'' :: (\Gamma, (a_S : A_S) \; ; \; \Delta, \Delta_2) \qquad \cdot \; ; \; \Gamma_0 \; ; \; \Delta_2 \vDash^q Q_{a_S} :: (c_m : C)}{\Gamma_0 \overset{E'+q+w}{\vDash} \mathcal{D}_1'', \text{proc}(c_m, w, Q_{a_S}) :: (\Gamma, (a_S : A_S) \; ; \; \Delta, (c_m : C))} \text{proc}_T$$

□

Definition 2. *A process* $\text{proc}(c_m, w, P)$ *is said to be poised if it is trying to receive a message on* c_m. *A message* $\text{msg}(c_m, w, M)$ *is said to be poised if it is trying to send a message along* c_m. *A configuration* Ω *is said to be poised if all the processes and messages in* Ω *are poised. Concretely, the following processes are poised.*

- $\text{proc}(c_m, w, c_m \xleftarrow{-} d_m)$
- $\text{proc}(c_m, w, \text{case } c_m \; (l_i \Rightarrow P_i)_{i \in I})$
- $\text{proc}(c_m, w, x_R \leftarrow \text{recv } c_m \; ; \; P)$
- $\text{proc}(c_m, w, x \leftarrow \text{recv } c_m \; ; \; P)$
- $\text{proc}(c_S, w, c_L \leftarrow \text{accept } c_S \; ; \; P)$
- $\text{proc}(c_L, w, c_S \leftarrow \text{detach } c_L \; ; \; P)$
- $\text{proc}(c_m, w, \text{get } c_m \; \{r\} \; ; \; P)$

Similarly, the following messages are poised.

- $\text{msg}(c_m, w, c_m.l_k \; ; \; P)$
- $\text{msg}(c_m, w, \text{send } c_m \; e_n \; ; \; P)$
- $\text{msg}(c_m, w, \text{send } c_m \; N \; ; \; P)$
- $\text{msg}(c_m, w, \text{close } c_m)$
- $\text{msg}(c_m, w, \text{pay } c_m \; \{r\} \; ; \; P)$

Theorem 5 (Process 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* $\text{proc}(a_L, w, P) \in \Omega$.

Proof. Either $\Omega = \Omega_1, \text{proc}(c_m, w, P)$ or $\Omega = \Omega_1, \text{msg}(c_m, w, M)$. In either case, either $\Omega_1 \mapsto \Omega_1'$, in which case we are done. Or there is a process in Ω_1 blocked along a_S in which case, we are also done. Hence, in the final case, we get Ω_1 is poised and there is no process in Ω_1 blocked along a_S. Now, we case analyze on the structure of the process or message. We start with processes.

- Case $(\{\}E_{mn})$: In each case, the process spontaneously steps by spawning another process.
- Case (fwd^+ : $\mathsf{proc}(c_m, w, c_m \xleftarrow{+} d_k)$) :

$$\cdot \; ; \; \Gamma \; ; \; (d_k : A) \overset{0}{\vdash} c_m \xleftarrow{+} d_k :: (c_m : A)$$

  Since Ω_1 is poised, there must be a message in Ω_1 offering along $d_m : A$. We use Lemma 6 to move the message just left of the process, and then apply the fwd^+ rule. Hence, Ω can step.
- Case (fwd^- : $\mathsf{proc}(c_m, w, c_m \xleftarrow{-} d_m)$) : This process is poised, hence Ω is poised.
- Case ($\oplus R$: $\mathsf{proc}(c_m, w, c_m.k \; ; \; P)$) : Ω steps using $\oplus C_s$ rule.
- Case ($\oplus L$: $\mathsf{proc}(d_k, w, \mathsf{case}\; c_m\; (l \Rightarrow Q_l)_{l \in L})$) :

$$\cdot \; ; \; \Gamma \; ; \; (c_m : \oplus\{l : A_l\}_{l \in L}) \overset{q}{\vdash} \mathsf{case}\; c_m\; (l \Rightarrow Q_l)_{l \in L} :: (d_k : C)$$

  Since Ω_1 is poised, there must be a message in Ω_1 offering along $c_m : \oplus\{l : A_l\}_{l \in L}$. We use Lemma 6 to move the message just left of the process, and then apply the $\oplus C_r$ rule. Hence, Ω can step.
- Case ($\multimap R$: $\mathsf{proc}(c_m, w, x_n \leftarrow \mathsf{recv}\; c_m \; ; \; P)$) : This process is poised, hence Ω is poised.
- Case ($\multimap L$: $\mathsf{proc}(c_m, w, \mathsf{send}\; c_m\; e_n \; ; \; Q)$) : Ω steps using $\multimap C_s$ rule.
- Case ($\uparrow_\mathsf{L}^\mathsf{S} R$: $\mathsf{proc}(c_\mathsf{S}, c_\mathsf{L} \leftarrow \mathsf{accept}\; c_\mathsf{S} \; ; \; P)$) : This process is poised, hence Ω is poised.
- Case ($\uparrow_\mathsf{L}^\mathsf{S} L$: $\mathsf{proc}(c_m, w, a_\mathsf{L} \leftarrow \mathsf{acquire}\; a_\mathsf{S} \; ; \; Q)$) :

$$\cdot \; ; \; \Gamma, (a_\mathsf{S} : \uparrow_\mathsf{L}^\mathsf{S} A_\mathsf{L}) \; ; \; \Delta \overset{q}{\vdash} a_\mathsf{L} \leftarrow \mathsf{acquire}\; a_\mathsf{S} \; ; \; Q :: (c_m : C)$$

  There must be some process in Ω_1 that offers on a_S. Either this process is in shared mode or linear mode. If the process is in shared mode, and since Ω_1 is poised, the process must be $\mathsf{proc}(a_\mathsf{S}, w', a_\mathsf{L} \leftarrow \mathsf{accept}\; a_\mathsf{S} \; ; \; P)$ in which case, we can use Lemma 8 to move the two processes next to each other and Ω can step using $\uparrow_\mathsf{L}^\mathsf{S} C$ rule. Or the process is in linear mode in which case the acquiring process is blocked and there is some $\mathsf{proc}(a_\mathsf{L}, w', P)$ in Ω.
- Case ($\downarrow_\mathsf{L}^\mathsf{S} R$: $\mathsf{proc}(c_\mathsf{S}, c_\mathsf{L} \leftarrow \mathsf{detach}\; c_\mathsf{S} \; ; \; P)$: This process is poised, hence Ω is poised.
- Case ($\downarrow_\mathsf{L}^\mathsf{S} L$: $\mathsf{proc}(c_\mathsf{T}, w, a_\mathsf{L} \leftarrow \mathsf{release}\; a_\mathsf{S} \; ; \; Q)$) :

$$\cdot \; ; \; \Gamma \; ; \; \Delta, (a_\mathsf{L} : \downarrow_\mathsf{L}^\mathsf{S} A_\mathsf{S}) \overset{q}{\vdash} a_\mathsf{L} \leftarrow \mathsf{release}\; a_\mathsf{S} \; ; \; Q :: (c_m : C)$$

  There must be some process in Ω_1 that offers along a_L. Since Ω_1 is poised, this process must be $\mathsf{proc}(a_\mathsf{L}, w', a_\mathsf{S} \leftarrow \mathsf{detach}\; a_\mathsf{L} \; ; \; P)$ in which case we use Lemma 9 to move the releasing process next to the detaching process and Ω can step using $\downarrow_\mathsf{L}^\mathsf{S} C$ rule.

That completes the cases where the last predicate is a process. Now, we consider the cases where the last predicate is a message.

- Case (fwd^- : $\mathsf{msg}(e_k, w, M(c_m))$) : There must be some process in Ω_1 that offers along d_m. Since Ω_1 is poised, if there is a forwarding process $\mathsf{proc}(c_m, w', c_m \xleftarrow{-} d_m)$ in Ω_1, then Ω steps using fwd^- rule. Hence, in the following cases, we assume that the offering process used by the message will not be a forwarding process.
- Case ($\oplus$: $\mathsf{msg}(c_m, c_m.k \; ; \; M)$) : This message is poised, hence Ω is poised.
- Case ($\multimap$: $\mathsf{msg}(c_m^+, \mathsf{send}\; c_m\; e_\mathsf{R} \; ; \; c_m^+ \leftarrow c_m)$) : There must be a process in Ω_1 that offers along c_m. Since Ω_1 is poised, this process must be $\mathsf{proc}(c_m, x_n \leftarrow \mathsf{recv}\; c_m \; ; \; P)$. We move the process to the left of this message using Lemma 7. And then, Ω can step using $\multimap C_r$ rule.

□

REFERENCES

2016. Solidity by Example. https://solidity.readthedocs.io/en/v0.3.2/solidity-by-example.html. Accessed: 2018-11-04.

2017. Ether.Camp's HKG Token Has A Bug And Needs To Be Reissued. https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and- Accessed: 2019-02-25.

2018. Bamboo. https://github.com/cornellblockchain/bamboo. Accessed: 2018-11-04.

2018. ERC20 Token Standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard. Accessed: 2018-02-027.

2018. The Michelson Language. https://www.michelson-lang.com/. Accessed: 2018-11-04.

2018. Rholang. https://github.com/rchain/Rholang. Accessed: 2018-11-04.

2018. Vyper. https://vyper.readthedocs.io/en/latest/index.html. Accessed: 2018-11-04.

2018. Welcome to Liquidity's documentation! http://www.liquidity-lang.org/doc/index.html. Accessed: 2018-11-04.

2019. Nomos Implementation. link to repository removed for double blind review. Accessed: 2019-11-11.

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017*. 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. 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)*. ACM, New York, NY, USA, 152–164. https://doi.org/10.1145/2784731.2784753

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP (2017), 37:1–37:29.

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. (2019). 28th European Symposium on Programming (to appear).

P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models. In *8th International Workshop on Computer Science Logic (CSL) (Lecture Notes in Computer Science)*, Vol. 933. Springer, 121–135. An extended version appeared as Technical Report UCAM-CL-TR-352, University of Cambridge.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

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

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *21st International Conference on Concurrency Theory (CONCUR)*. Springer, 222–236.

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

Iliano Cervesato and Andre Scedrov. 2009. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation* 207, 10 (2009), 1044 – 1077. https://doi.org/10.1016/j.ic.2008.11.006 Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

K. Mani Chandy, Jayadev Misra, and Laura M. Haas. 1983. Distributed Deadlock Detection. *ACM Trans. Comput. Syst.* 1, 2 (May 1983), 144–156. https://doi.org/10.1145/357360.357365

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

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 50–63.

Karl Crary and Michael J. Sullivan. 2015. Peer-to-peer Affine Commitment Using Bitcoin. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 479–488. https://doi.org/10.1145/2737924.2737997

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 140–151. https://doi.org/10.1145/2784731.2784749

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *33rd ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*.

Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding Concurrency to Smart Contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. ACM, New York, NY, USA, 303–312. https://doi.org/10.1145/3087801.3087835

Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (01 Nov 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

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

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 116 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276486

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158136

Maurice Herlihy. 2019. Blockchains from a Distributed Computing Perspective. *Commun. ACM* 62, 2 (Jan. 2019), 78–85. https://doi.org/10.1145/3209623

Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Stefănescu, and Grigore Rosu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 204–217.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*.

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

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

Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR)*. Springer, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming (ESOP)*. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 273–284.

Blockchain Insurance Industry Initiative. 2008. B3i. (2008).

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

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

Shuvendu K. Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. 2018. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *CoRR* abs/1812.08829 (2018). arXiv:1812.08829 http://arxiv.org/abs/1812.08829

Angwei Law. 2017. *Smart contracts and their application in supply chain management.* Ph.D. Dissertation. Massachusetts Institute of Technology.

Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse Record and Replay with Controlled Scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 576–593. https://doi.org/10.1145/3314221.3314635

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

Lucius Gregory Meredith. 2015. Linear Types Can Change the Blockchain. *arXiv preprint arXiv:1506.01001* (2015).

Don P. Mitchell and Michael J. Merritt. 1984. A Distributed Algorithm for Deadlock Detection and Resolution. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (PODC '84)*. ACM, New York, NY, USA, 282–284. https://doi.org/10.1145/800222.806755

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

Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf.

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. Springer, 3–22.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

Francois Pottier and Yann Régis-Gianas. 2019. *Menhir Reference Manual*.

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. 2018. *Adjoint Logic*. Technical Report. Carnegie Mellon University.

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

Jason Reed. 2009. A Judgmental Deconstruction of Modal Logic. (January 2009). http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf Unpublished manuscript.

Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.* 17, 2 (May 1999), 133–152. https://doi.org/10.1145/312203.312214

Vikram Saraph and Maurice Herlihy. 2019. An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts. *CoRR* abs/1901.01376 (2019). arXiv:1901.01376 http://arxiv.org/abs/1901.01376

Alceste Scalas and Nobuko Yoshida. 2019. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 30 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290343

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 185 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360611

David Siegel. 2016. Understanding The DAO Hack for Journalists. https://medium.com/@pullnews/understanding-the-dao-hack-for-jour

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: a Monadic Integration. In *22nd European Symposium on Programming (ESOP)*. Springer, 350–369.

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

Gavin Wood. 2014. Ethereum: A secure decentralized transaction ledger. http://gavwood.com/paper.pdf .