Resource-Aware Session Types for Digital Contracts

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 attacks and out-of-gas vulnerabilities.

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.

Many of today's prominent smart contract languages suffer from security vulnerabilities, which have severe financial consequences. A well-known example is the attack on The DAO [Siegel 2016], resulting in a $60 million theft by exploiting a contract re-entrancy vulnerability. Smart contract languages have been typically derived from existing general-purpose languages [Auc 2016; Liq 2018; Cachin 2016] and 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 deletion of a contract's assets.

This article presents the design, type-theoretic foundation, and implementation of Nomos, a language for digital contracts accommodating these requirements by construction.

To express and enforce the protocols underlying a contract, Nomos is based on *session types* [Caires and Pfenning 2010; Honda 1993; Honda et al. 1998, 2008; Pfenning and Griffith 2015; Toninho et al. 2013; Wadler 2012]. Session types capture the protocols of interactions in the type, rather

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/

than the implementation code, and type-checking statically guarantees protocol adherence at run-time. Delimiting the sequences of actions that must be executed atomically, session types also prevent *re-entrance* into a contract in an inconsistent state. To control resource usage, Nomos employs *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, allowing instantiation to track gas usage. As a result, Nomos contracts mitigate denial-of-service attacks without being vulnerable to out-of-gas exceptions. Moreover, resource bounds are integrated with session-typed protocols and enable precise path-sensitive descriptions of cost that avoid gaps between worst-case and average-case cost. To prevent duplication or deletion of assets, Nomos uses a *linear type system* [Girard 1987]. The effectful session-typed language, which implements contract interfaces and contract-to-contract communication, is integrated with a strict, general-purpose functional language using a contextual monad.

Integrating these 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 bound 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.

Nomos is completely formalized, including the type system, a cost semantics, and a transactional semantics to instantiate Nomos contracts on a blockchain. A type soundness proof ensures that protocols are followed at run-time and that types establish sound upper bounds on the resource consumption. Type checking is linear in the size of the program and resource bounds can be 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. To the best of our knowledge, this is the first implementation to integrate shared binary session types into a functional language with support for resource analysis.

To simplify programming and make Nomos accessible to digital contract developers, we incorporated the following design decisions: *(i)* we developed an intuitive surface syntax particularly related to the contextual monad integrating session types into a functional core; *(ii)* we used a bi-directional type checker with a particular focus on improving the quality of error messages whenever a Nomos program fails to typecheck to guide the programmer to locate the source of the error; *(iii)* we used an off-the-shelf LP solver to automatically infer channel modes and potential annotations so that the burden of inference does not fall on the programmer.

Our main technical 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 store 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);
- an implementation and case study of prominent blockchain applications (Section 8);
- a transactional semantics to deploy and execute Nomos contracts and transactions on a blockchain (Section 9).

In addition, the supplementary material details the technical development, provides additional explanations and provides the full implementation of the blockchain applications.

2 NOMOS BY EXAMPLE

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

Explicit Protocols of Interaction. Digital contracts, like traditional contracts, follow a *predefined protocol*. For instance, an auction contract distinguishes a bidding phase, where bidders submit their bids, possibly multiple times, from a subsequent collection phase, where the highest bidder receives the lot while all other bidders receive their bids back. In Solidity [Auc 2016], the bidding phase of an auction is typically implemented as the bid function below. This function receives a bid (msg.value) from a bidder (msg.sender) and adds it to the bidder's total previous bids (bidValue).

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

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

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

$$
\begin{aligned}
\text{auction} = \uparrow^{S}_{L} \triangleleft^{22} \oplus \{ & \textbf{running} : \&\{\textbf{bid} : \text{id} \to \text{money} \multimap \downarrow^{S}_{L}\text{auction}, && \%\ \text{recv bid from client} \\
& \qquad\quad \textbf{cancel} : \triangleright^{21}\downarrow^{S}_{L}\text{auction}\}, && \%\ \text{client canceled} \\
& \textbf{ended} : \&\{\textbf{collect} : \text{id} \to \oplus\{\textbf{won} : \text{lot} \otimes \downarrow^{S}_{L}\text{auction}, && \%\ \text{client won} \\
& \qquad\quad \textbf{lost} : \text{money} \otimes \triangleright^{7}\downarrow^{S}_{L}\text{auction}\}, && \%\ \text{client lost} \\
& \qquad\quad \textbf{cancel} : \triangleright^{21}\downarrow^{S}_{L}\text{auction}\}\} && \%\ \text{client canceled}
\end{aligned}
$$

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

In the former case, the auction will send the lot, in the latter case, it will return the client's bid. The linear product ($\otimes$) is dual to $\multimap$ and denotes the transfer of a linear value from the contract to the client. The auction type guarantees that a client cannot collect during the running phase, while they cannot bid during the ended phase.

Nomos uses *shared* session types [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 $\uparrow^S_L$ and $\downarrow^S_L$ modalities. The $\uparrow^S_L$ modality denotes the beginning of a critical section, the $\downarrow^S_L$ modality denotes its end. Programmatically, $\uparrow^S_L$ translates into an *acquire* of the auction session and $\downarrow^S_L$ into its *release*, which is only sound if the protocol behaves like an auction afterwards (*equi-synchronizing* type).

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

```
1:  stype bid = &{addr : id ∧ bid, val : money},   stype bids = hashmap_{id,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 ← check ← b' l                %  check if threshold reached
11:             | cancel ⇒ sa ← detach la ;               %  detach from client
12:                        sa ← run ← b l)               %  recurse
```

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

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

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

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

A bidder can now cause re-entrancy by creating a dummy contract with an unnamed *fallback* function (on the right) that calls the auction's collect function. This call is triggered when collect calls send (last line on the left), leading to an infinite recursive call to collect, depleting all funds from the auction. The message-passing framework of session types eliminates this vulnerability. While session types provide multiple clients access to a contract, the acquire-release discipline ensures that clients interact with the contract in mutual exclusion. To attempt re-entrancy, a bidder will need to acquire the auction contract twice without releasing it.

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

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

Nomos uses resource-aware session types [Das et al. 2018b] to statically analyze the resource cost of a transaction. They operate by assigning an initial *potential* to each process. This potential is consumed by each operation that the process executes or can be transferred between processes to share and amortize cost. The cost of each operation is defined by a cost model. If the cost model assigns a cost to each operation as equivalent to their gas cost during execution, the potential consumed during a transaction reflects upper bound on the gas usage.

Resource-aware session types express the potential as part of the session type using the operators $\triangleleft$ and $\triangleright$. The $\triangleleft$ operator prescribes that the client must send potential to the contract, with the amount of potential indicated as a superscript. Dually, $\triangleright$ prescribes that the contract must send potential to the client. In case of the auction contract, we require the client to pay potential for the operations that the contract must execute, both while placing and collecting their bids. If the cost model assigns a cost of 1 to each contract operation, then the maximum cost of an auction session is 22 (taking the max number of operations in all branches). Thus, we require the client to send 22 units of potential at the start of a session using $\triangleleft^{22}$. In the lost branch of the auction type, on the other hand, the contract returns 7 units of potential to the client using $\triangleright^{7}$. This simulates gas usage in smart contracts, where the sender initiates a transaction with some initial gas, and the leftover gas at the end of the transaction is returned to the sender. In contrast to existing smart contract languages like Solidity, which provide no support for analyzing the cost of a transaction, Nomos' type checker has automatically inferred these potential annotations and guarantees that well-typed transactions cannot run out of gas.

Bringing It All Together. Combining all these features soundly in one language is challenging. In Nomos, we achieve this by using different *typing judgments* and *modes*, identifying the role of the process offered along that channel. The mode R denotes *purely linear processes* for linear assets or private data structures, such as b and l in the auction. The modes S and L denote *sharable processes*, i.e., contracts, that are either in their shared or linear phase such as sa and la, respectively. The mode T denotes a *transaction process* that can refer to shared and linear processes and is issued by a user, such as bidder in the auction. The mode assignment carries over into the process typing judgments imposing invariants (Definition 1) that are key to type safety. The mode annotations are automatically inferred by the type checker relieving programmers from this burden.

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 $A_1, A_2, \ldots, A_n$

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

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

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

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. The complete formalization is presented in the supplement.

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

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

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

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

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

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

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

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

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:

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

The message $\mathsf{msg}(c_m, w, c_m.l\ ;\ c_m \leftarrow c_m^+)$ 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 $(y_n \leftarrow \mathsf{recv}\ x_m\ ;\ P)$, where P is the continuation. The corresponding client sends this channel using $(\mathsf{send}\ x_m\ w_n\ ;\ Q)$. The corresponding typing rules are presented in Figure 1. Operationally, the client creates a message containing the channel:

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

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*.

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

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.

Process and Type Definitions. Process definitions have the form $\Psi\ ;\ \Gamma\ ;\ \Delta \Vdash^g f = P :: (x_m : A)$ where f is the name of the process and P its definition. All definitions are collected in a fixed global signature Σ. We require well-typedness, i.e., $\Psi\ ;\ \Gamma\ ;\ \Delta \Vdash^g f = P :: (x_m : A)$ for every definition, which allows the definitions to be mutually recursive. For readability of the examples, we break a definition into two declarations, one providing the type (on the left) and the other the process definition (on the right) binding the variables x_m and those in Ψ, Γ and Δ (omitting their types):

$$\Psi\ ;\ \Gamma\ ;\ \Delta \Vdash^g f = P :: (x_m : A) \qquad\qquad x_m \leftarrow f\ \Psi \leftarrow \Gamma\ ;\ \Delta = P$$

A new instance of a defined process f can be spawned with the expression $x_m \leftarrow f\ \overline{y_1} \leftarrow \overline{y_2}\ ;\ Q$ where $\overline{y_1}$ is a sequence of functional variables matching the antecedents Ψ and $\overline{y_2}$ is a sequence of channels matching the antecedents $\Gamma\ ;\ \Delta$. The newly spawned process will use all variables in $\overline{y_1}$ and channels in $\overline{y_2}$ and provide x_m to the continuation Q. The operational semantics is defined by

$$(\mathsf{def}C): \mathsf{proc}(c_k, w, x_m \leftarrow f\ \overline{d} \leftarrow \overline{e}\ ;\ Q) \mapsto \mathsf{proc}(a_m, 0, [a_m/x_m, \overline{d}/\Psi, \overline{e}/\Gamma\ \Delta]P), \mathsf{proc}(c_k, w, [a_m/x_m]Q)$$

where a_m is a fresh channel. Here we write $[\overline{d}/\Psi]$ and $[\overline{e}/\Gamma\ \Delta]$ to denote substitution of the variables in $\overline{d}$ and $\overline{e}$ for the corresponding variables in Ψ and $\Gamma\ ;\ \Delta$ respectively in that order.

Sometimes a process invocation is a *tail call*, written without a continuation as $x_m \leftarrow f\ \overline{y_1} \leftarrow \overline{y_2}$. This is a short-hand for $x'_m \leftarrow f\ \overline{y_1} \leftarrow \overline{y_2}\ ;\ x_m \leftarrow x'_m$ for a fresh variable x'_m, that is, we create a fresh channel and immediately identify it with x_m (although it is implemented more efficiently).

Session types can be naturally extended to include recursive types. For this purpose we allow (possibly mutually recursive) type definitions $X = A$ in the signature, where we require A to be *contractive* [Gay and Hole 2005]. This means here that A should not itself be a type name. Our type definitions are *equi-recursive* so we can silently replace X by A during type checking, and no explicit rules for recursive types are needed.

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:

$$\begin{array}{lcll} A_S & ::= & \uparrow^S_L A_L & \text{shared session type} \\ A_L & ::= & \ldots \mid \downarrow^S_L A_S & \text{linear session types} \end{array}$$

The $\uparrow^S_L$ type modality translates into an *acquire*, while the dual $\downarrow^S_L$ type modality into a *release*. Whereas mutual exclusion is one key ingredient to guarantee session fidelity (a.k.a. type preservation) for shared session types, the other key ingredient is the requirement that a session type is *equi-synchronizing*. A session type is equi-synchronizing if it imposes the invariant on a process to be released back to the same type at which the process was previously acquired. 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.

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

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

Fig. 2. Grammar for shared session types

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 (i) for all* $d_k \in \Delta \implies k = \mathsf{R} \vee k = \mathsf{L}$*, and (ii)* $A \in \mathbf{L}(A_{\mathsf{L}})$*.*
(4) If $m = \mathsf{T}$*, then* $A \in \mathbf{L}(A_{\mathsf{T}})$*.*

Figure 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 modes R or L. Finally, the fourth typing judgment types a linear process, corresponding to a *transaction* holding access to shared channels Γ and linear channels Δ, and offering at mode T.

This novel factorization upholds preservation while allowing shared contract processes to rely on linear resources. The modes impose the ordering $\mathsf{R} < \mathsf{S} < \mathsf{L} < \mathsf{T}$ among the linear channels in the configuration. A process (offering a channel) at a certain mode is allowed to rely only on processes at the same or lower mode. These are exactly the conditions imposed by Definition 1. This introduces an implicit ordering among the linear processes depending on their mode, thus eliminating cycles in the process tree. Relatedly, shared processes can only refer to shared channels (at mode S) or purely linear channels (at mode R) as exemplified by the judgment Δ purelin in Figure 3. Formally, Δ purelin denotes that for all $d_k \in \Delta \implies k = \mathsf{R}$. This ensures that a shared contract must release all processes it has acquired before itself being released. This further enforces an ordering in which the channels are acquired and released, thus *allowing contracts to interact with other contracts without compromising type safety.*

Shared session types introduce new typing rules into our system, concerning the *acquire-release* constructs (see Figure 3). In rule $\uparrow^{\mathsf{S}}_{\mathsf{L}} L$, an acquire is applied to the shared channel $x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$ in Γ and yields a linear channel x_{L} added to Δ when successful. A contract process can *accept* an acquire request along its offering shared channel x_{S}. After the accept is successful, the shared contract process transitions to its linear phase, now offering along the linear channel x_{L} (rule $\uparrow^{\mathsf{S}}_{\mathsf{L}} R$).

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

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

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

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

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

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

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

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

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

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

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.

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

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 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

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

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

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

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

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

have the expected term formers such as function abstraction and application, type constructors, and pattern matching. We also define a standard type judgment for the functional part of the language.

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

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

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

The monadic *bind* operation implements process composition and acts as the elimination form for values of type $\{A_{\mathsf{R}} \leftarrow \overline{A_{\mathsf{R}}}\}_{\mathsf{R}}$. The bind operation, written as $c_{\mathsf{R}} \leftarrow M \leftarrow \overline{d_{\mathsf{R}}}\ ;\ Q_c$, composes the process underlying the monadic term M, which offers along channel c_{R} and uses channels $\overline{d_{\mathsf{R}}}$, with Q_c, which uses c_{R}. The typing rule for the monadic bind is rule $\{\}E_{\mathsf{RR}}$ in Figure 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 x is R. The effect of executing a bind is the spawn of the purely linear process corresponding to the monad M, and the parent process continuing with Q. The corresponding operational semantics rule (named $\mathsf{spawn}_{\mathsf{RR}}$) is given as follows:

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

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

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

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

The type $\tau \to A$ prescribes receiving a value of type τ with continuation type A, while its dual $\tau \wedge A$ prescribes sending a value of type τ with continuation A. The corresponding typing rules for arrow ($\to R, \to L$) are given in Figure 4 (rules for $\wedge$ 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:

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

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.

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

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)

```
1:  (n : int) ; (l_R : list_coin) ⊢ wallet :: (m_R : money)
2:    m ← wallet n ← l =
3:      case m                                  %  case analyze on label received on m
4:        (value ⇒ send m n ;                   %  receive value, send n
5:                  m ← wallet n ← l
6:        | add ⇒ m' ← recv m ;                 %  receive m' : money to add
7:                 m'.value ;                   %  query value of m'
8:                 v ← recv m' ;
9:                 m'.coins ;                   %  extract list of coins stored in m'
10:                k ← append ← l m' ;          %  append list received to internal list
11:                m ← wallet (n + v) ← k
12:       | subtract ⇒ n' ← recv m ;            %  receive int to subtract
13:                    if (n' > n) then
14:                      m.insufficient ;       %  funds insufficient
15:                      m ← wallet n ← l
16:                    else
17:                      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: $m \leftarrow \mathit{wallet}\ (n - n') \leftarrow l'$
23: | **coins** $\Rightarrow m \leftarrow l$)

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 bear the risk of high losses if the contract has flaws or vulnerabilities.

It is not trivial to decide on the right amount for the fee since the gas cost of the contract does not only depend on the requested transaction but also on the (a priori unknown) state of the blockchain. Thus, precise and static estimation of gas cost facilitates transactions and reduces risks. We discuss our approach of tracking resource usage, both at the functional and process layer.

Functional Layer. Numerous techniques have been proposed to statically derive resource bounds for functional programs [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. 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.

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

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 supplementary material.

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

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

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. 2018b], our version is simpler and more streamlined. Instead of requiring every message to carry a potential (and potentially tagging several messages with 0 potential), we introduce two new type constructors for exchanging potential.

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

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

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

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

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

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

Fig. 5. Selected typing rules corresponding to potential.

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

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

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

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 $x_m : \triangleleft^r A$ using the *get* construct and the continuation executes with $p = q + r$ units of potential. In the dual rule $\triangleleft L$, a process storing potential $q = p + r$ sends r units along the channel $x_m : \triangleleft^r A$ in Δ using the *pay* construct, and the continuation remains with p units of potential. The typing rules for the dual constructor $\triangleright^r A$ are the exact inverse. Finally, executing the tick (r) construct consumes r potential from the stored process potential q, and the continuation remains with $p = q - r$ units, as described in the tick rule.

The tick construct is used to simulate a cost model in Nomos. If an operation (e.g., sending a message, calling a function, etc.) has a cost of r, this cost is simulated by inserting tick (r) just before the operation. Then, the tick operations are the only ones that cost potential, thus simplifying the type system. These tick operations are automatically inserted by the Nomos type checker, using a predefined cost model that assigns a constant cost to each operation. In addition, our implementation provides some standard cost models, for instance, that assign a unit cost to each internal operation and sending a message.

Integration. Since both AARA for functional programs and resource-aware session types are based on the integration of the potential method into their type systems, their combination is natural. The two points of integration of the functional and process layer are (i) spawning a process, and (ii) sending/receiving a value from the functional layer. Recall the spawn rule $\{\}E_{\mathsf{RR}}$ from Figure 4. A process storing potential $r = p + q$ can spawn a process corresponding to the monadic expression M, if M needs p units of potential to evaluate, while the continuation needs q units of potential to execute. Moreover, the functional context Ψ is shared in the two premises as Ψ_1 and Ψ_2 using the judgment $\Psi \curlyvee (\Psi_1, \Psi_2)$. This judgment, already explored in prior work [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 $\longrightarrow L$ in Figure 4 follows a similar pattern. A process Q storing $r = p + q$ potential sends a monadic expression M needing p units of potential to evaluate, and the continuation remains with q units of potential to execute. The p units of potential are consumed to evaluate M to a value before sending since only values are exchanged at runtime. Thus, the combination of the two type systems is smooth, assigning a uniform meaning to potential, both for the functional and process layer. Remarkably, this technical device of exchanging functional values can be used to exchange non-constant potential with messages. For instance, exchanging a list $l : L^q(\tau)$ will exchange $q \cdot n$ units of potential where n is the size of the list l.

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

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

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* $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{g} (x_m : A)$ *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 impose a partial order among the processes and messages, requiring the provider of a channel to appear before its client. We stipulate that no two distinct processes or messages in a well-formed configuration provide the same channel c.

The typing judgment for configurations has the form $\Sigma\ ;\ \Gamma_0 \overset{E}{\models} \Omega :: (\Gamma\ ;\ \Delta)$ defining a configuration Ω providing shared channels in Γ and linear channels in Δ. Additionally, we need to track the mapping between the shared channels and their linear counterparts offered by a contract process, switching back and forth between them when the channel is acquired or released respectively. This mapping, along with the type of the shared channels, is stored in Γ_0. E is a natural number and

stores the sum of the total potential and work as recorded in each process and message. We call E the energy of the configuration. The supplement details the configuration typing rules.

Finally, Σ denotes a signature storing the type and function definitions. A signature is well-formed if *(i)* every type definition $V = A_V$ is *contractive* [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* $\Sigma\ ;\ \Gamma_0 \overset{E}{\models} \Omega :: (\Gamma\ ;\ \Delta)$. *If the configuration takes a step, i.e.* $\Omega \mapsto \Omega'$, *then there exist* Γ_0', Γ' *such that* $\Sigma\ ;\ \Gamma_0' \overset{E}{\models} \Omega' :: (\Gamma'\ ;\ \Delta)$, *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* $\Gamma_0 \overset{E}{\models} \Omega :: (\Gamma\ ;\ \Delta)$. *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* $\mathsf{proc}(a_L, w, P) \in \Omega$.

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 (929 lines of code), a type checker (2388 lines of code), a pretty printer (451 lines of code), an LP solver interface (915 lines of code) and an interpreter (1286 lines of code) for implementing, type checking and executing Nomos programs. We also describe our efforts to simplify programming and improve accessiblity of Nomos to developers.

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

```
type 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 restricted to reflexivity (constant time), although we have also implemented the standard co-inductive algorithm [Gay and Hole 2005] which is quadratic in the size of type definitions. For all our examples, the reflexive notion of equality was sufficient. *Type checking is linear time in the size of the program*, which is important in the blockchain domain where type checking can be 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 automatic inference algorithm for both these annotations by relying on an off-the-shelf LP solver.

Using ideas from existing techniques for type inference for AARA [Hoffmann et al. 2017; Hofmann and Jost 2003], we reduce the reconstruction of potential annotations to linear optimization. To this end, Nomos' inference engine uses the Coin-Or LP solver. In a Nomos program, the programmer can indicate unknown potential using $*$. Thus, resource-aware session types can be marked with $\triangleright^*$ and $\triangleleft^*$, list types can be marked as $L^*(\tau)$ and process definitions can be marked with $|\{*\}-$ on the turnstile. The mode of all the channels is marked as 'unknown' while parsing.

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 minimizes the value of the potential annotations to achieve tight bounds. The LP solver either returns that the constraints are infeasible, or returns a satisfying assignment, which is then substituted into the program. 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 $\uparrow^S_L$, ii) \/ for $\downarrow^S_L$, iii) <{q}| for $\triangleleft^q$, iv) |{q}> for $\triangleright^q$, v) ^ for $\wedge$, vi) *[m] for $\otimes_m$, vii) -o[m] for $\multimap_m$.

ERC-20 Token Standard. 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 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. In our implementation, we store two hash maps, one for the balance of each account, and one for the allowance between each pair of accounts. The contract relies on custom linear coins that are used exclusively for exchanges among the private accounts. These coins can be minted by a special transaction that can only be issued by the owner of the contract and that creates coins out of thin air (consuming gas to create coins). We use a built-in type to represent a single coin, providing custom functions to *mint* and *burn* a coin. The type for the two hash maps is described below.

```
type balance-map = &{ get-balance : id -> int ^ balance-map,
                      transfer : id -> id -> int -> balance-map}
type allowance-map = &{ get : id -> id -> int ^ allowance-map,
                        set : id -> id -> int -> allowance-map}
```

The type balance−map supports two functionalities: querying the balance value of an account by receiving an id and responding with an int; and allowing a transfer by receiving the sender and receiver ids and the transfer amount. In each case, the type recurses back to balance−map allowing other users to interact with the hash map. The allowance−map type stores the allowances for each pair of accounts, which can be queried and updated using the get and set functionalities. They have a similar communication protocol as the balance−map.

Another implementation can use a different linear type with its own introduction and elimination forms for minting and burning, respectively. Nomos' linear type system enforces that the coins are treated linearly modulo minting and burning.

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]. 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, acquire-release discipline along with the guarded session type guarantees that the user submitting the solution receives their expected winnings.

Voting. The voting contract provides a ballot type in an election.

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

This contract allows voting when the election is **open** by receiving the candidate's *id*. To only allow legitimate voters to cast a ballot and prevent double voting by the same voter, the contract then checks if the voter is eligible to vote. It then replies with **vote** or **novote** depending on their eligibility. Once the election closes (the **closed** label), the contract can be acquired to check the winner of the election. We use two implementations for the contract: the first stores a counter for each candidate that is updated after each vote is cast (voting in Table 2); the second does not use a counter but stores potential inside the vote list that is consumed for counting the votes at the end (voting-aa in Table 2). This stored potential is provided by the voter to amortize the cost of counting. The type above shows the potential annotations corresponding to the latter.

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

```
type insurer = /\ <{6}| &{
  submit : claim -> +{ success : money *[R] |{0}> \/ insurer,
                       failure : |> \/ insurer } }
type verifier = /\ <{3}| &{
  verify : claim -> +{ valid : |{0}> \/ verifier,
                       invalid : |{0}> \/ verifier } }
```

The insurer type provides the option to **submit** a claim by receiving it and responds with **success** or **failure** depending upon verification of the claim. If the claim is successful, the insurer sends

Contract	LOC	Defs	Procs	T (ms)	Vars	Cons	I (ms)	Gap
auction	176	5	10	0.558	229	730	5.225	3
ERC 20	136	4	2	0.579	161	561	4.317	6
puzzle	108	3	7	0.410	126	389	8.994	8
voting	101	3	6	0.324	109	351	3.664	0
voting-aa	101	3	7	0.346	140	457	3.926	0
insurance	56	3	2	0.299	76	224	8.289	0
escrow	85	2	2	0.404	95	321	3.816	3
bank	147	4	5	0.663	173	561	4.549	0
wallet	30	3	2	0.231	32	102	3.224	0

Table 2. Evaluation of Nomos with Case Studies. LOC = lines of code; Defs = #type definitions; Procs = #process definitions; T (ms) = type checking time in ms; Vars = #potential and mode variables generated during type checking; Cons = #constraints generated during type checking; I (ms) = potential and mode inference time in ms; Gap = maximal gas bound gap.

over the reimbursement in the form of money. The verifier type provides the option to **verify** a claim by receiving it and responding with **valid** or **invalid** depending on the validity of the claim.

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

8.2 Experimental Evaluation

We implemented 8 case studies in Nomos. We have already discussed the auction (Section 2), ERC 20, puzzle, voting and insurance contracts. The other case studies are:

- An escrow to exchange bonds between two parties.
- A bank account that allows users to create accounts, make deposits and withdrawals and check their balance relying on custom linear coins.
- A wallet allowing users to store money on the blockchain.

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 number of type (Defs) and process definitions (Procs), 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. Moreover, Nomos' type system can be easily modified to only allow contracts with tight bounds.

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

9 BLOCKCHAIN INTEGRATION

To integrate Nomos 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.

Nomos on a Blockchain. We assume a blockchain like Ethereum that contains a set of Nomos contracts $C_1, \ldots, C_n$ together with their type information $\cdot\ ;\ \Gamma^i\ ;\ \Delta_{\mathsf{R}}^i \vdash^{q_i} C_i :: (x_{\mathsf{S}}^i : A_{\mathsf{S}}^i)$. The shared context Γ^i types the shared contracts that C_i refers to, and the linear context Δ_{R}^i types the contract's linear assets. The channel name x_{S}^i of a contract is its address and has to be globally unique. We allow contracts to carry potential given by the annotation q_i and the potential defined by the annotations in Δ_{R}^i but the type system could easily be altered to suppress the stored potential.

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

$$\Sigma\ ;\ \Gamma \overset{E}{\vDash} \mathsf{proc}(x_{\mathsf{S}}^1, w_1, C_1) \ldots \mathsf{proc}(x_{\mathsf{S}}^n, w_n, C_n) :: (\Gamma\ ;\ \cdot)$$

where $\Gamma = (x_{\mathsf{S}}^1 : A_{\mathsf{S}}^1), \ldots, (x_{\mathsf{S}}^n : A_{\mathsf{S}}^n)$ and $E = \Sigma_{i=1}^n q_i + w_i$ is the total energy, that is, the sum of the stored potential and previously performed work. To perform a transaction with a contract, a user submits a transaction script Q (a process) that is well-typed with respect to the existing contracts:

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

We mandate that the transaction offers along a channel of type $\mathbf{1}$ and terminates by sending a close message on its offered channel. This approach enables dynamic deadlock detection (explained later) and allows abortion of a transaction if a deadlock is detected. This script process is added to the set of contracts and the new (closed) configuration is typed as

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

This configuration then steps according to the Nomos semantics, ending with the termination of the script Q, leaving the configuration in a stuck state again to start a new transaction. If type checking were too costly here, that can lead to yet another source of denial-of-service attacks. In Nomos however, type checking is linear time in the size of the script.

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 $\{x_\mathsf{T} \leftarrow Q\} : \{\mathbf{1}\} = \mathsf{tx\_code}$. 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 create new contracts or update the state of existing ones. In the former case, new contracts are added to the blockchain state, making them visible in the type of the configuration for subsequent transactions to access. In the latter 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.

Deterministic Execution. Since blockchains rely on consensus among the miners, it is important to ensure deterministic execution of transactions. However, Nomos semantics has one source of non-determinism: the *acquire-accept* rule where an accepting contract latches on to any acquiring transaction. One simple approach to resolve this non-determinism is to determinize the resource scheduler based on some heuristics. Another promising approach 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.

Deadlocks. The only language specific reason a transaction can fail is a deadlock in the transaction code. Our progress theorem accounts for this possibility of deadlocks. Since a valid blockchain state represents a stuck configuration of a particular form (only shared contracts in the configuration), we verify at the end of the transaction execution if the new configuration has this form. If not, we conclude that a deadlock occurred during the execution, and we simply abort the whole transaction. We maintain snapshots of the configuration after every transaction execution, so we simply revert to the previous valid blockchain state. It is the user's responsibility to issue a new transaction that does not deadlock. In the future, we also plan to employ deadlock prevention techniques [Balzer et al. 2019] to statically rule out deadlocks.

10 OTHER 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. 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. However, none of these languages describe and enforce communication protocols statically. 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. Scilla does not use session types or linearity but features static gas bounds. A difference is that Nomos' bounds are not asymptotic and are proved sound with respect to a cost semantics. The Move programming language from Facebook [Blackshear et al. 2019] is a flexible language based on Rust [Klabnik and Nichols 2018] to implement contracts on the Libra blockchain. Similar to Nomos, it provides the ability to define custom linear types to represent assets. However, it does not provide support to express contract protocols or gas usage.

Static Analysis. Analysis of smart contracts has received substantial attention [Grishchenko et al. 2018; Tikhomirov et al. 2018] recently due to their security vulnerabilities [Atzei et al. 2017; Tsankov et al. 2018]. 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. 2018a,b], but was 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 [Das and Pfenning 2020] for expressing and verifying functional correctness of contracts against their specifications. We also plan 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] and evaluate the corresponding speed-up.

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-needs-to-be-reissued. 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

Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources.

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).

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 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. 2018a. Parallel Complexity Analysis with Temporal Session Types. In *23rd International Conference on Functional Programming (ICFP'18)*.

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

Ankush Das and Frank Pfenning. 2020. Session Types with Arithmetic Refinements and Their Application to Work Analysis. arXiv:cs.PL/2001.04439

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

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 51–78.

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)*.

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

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

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-journalists-2312dd43e993.

S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16.

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.

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

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.