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

Programming digital contracts comes with unique challenges, which include expressing and enforcing protocols of interaction, controlling resource usage, and tracking linear assets. This article presents the type-theoretic foundation of Nomos, a programming language for digital contracts whose type system addresses the aforementioned domain-specific requirements. To express and enforce protocols, Nomos is based on shared binary session types rooted in linear logic. To control resource usage, Nomos uses resource-aware session types and automatic amortized resource analysis, a type-based technique for inferring resource bounds. To track linear assets, Nomos employs a linear type system that prevents assets from being duplicated or discarded. The technical contribution is the design and soundness proof of Nomos' type system, which integrates shared session types and resource-aware session types with a functional type system that supports automatic amortized resource analysis. To evaluate Nomos' viability for digital contract development, we include a case study of several blockchain applications and demonstrate how their vulnerabilities do not arise in corresponding Nomos applications. In particular, Nomos ensures that synthetic tokens implement their technical standard and prevents vulnerabilities due to violations of linearity, out-of-gas exceptions, or dependence on transaction ordering. To evaluate Nomos' session-based design, we report on the performance of three digital contracts implemented in an existing prototype language.

1 INTRODUCTION

Digital contracts are computer programs that describe and enforce 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 an enforcement mechanism that does not rely on third parties. 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]. Digital contracts hold the promise to lower cost, increase fairness, and popularize access to the financial infrastructure.

Smart contracts have not only shed light on the benefits of digital contracts but also on their potential risks. Like all software, smart contracts can contain bugs and security vulnerabilities [Atzei et al. 2017], which can have direct financial consequences. A well-known example, is the attack on The DAO [Siegel 2016], resulting in a multi-million dollar theft by exploiting a contract vulnerability. Maybe even more important than the direct financial consequences is the potential erosion of trust in cryptocurrencies as a result of such failures.

Contract languages today are derived from existing general-purpose languages like JavaScript (Ethereum's Solidity [Auc 2016]), Go (in the Hyperledger project [Cachin 2016]), or OCaml (Tezos' Liquidity [Liq 2018]). While this makes contract languages look familiar to software developers, it is inadequate to accommodate the domain-specific requirements of digital contracts.

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.

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

- Instead of centering contracts on their interactions with users, the high-level protocol of the intended interactions with a contract is buried in the implementation code, hampering understanding, formal reasoning, and trust.
- Resource (or *gas*) usage of digital contracts is of particular importance for transparency and consensus. However, obliviousness of resource usage in existing contract languages makes it hard to predict the cost of executing a contract and prevent denial-of-service vulnerabilities.
- Existing languages fail to enforce linearity of assets, endangering the validity of a contract when assets get duplicated or deleted, accidentally or maliciously [Meredith 2015].

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

In this article, we present the type-theoretic foundation of Nomos, a programming language for digital contracts whose type system makes explicit the protocols governing a contract, provides static bounds on the resource cost of a transaction, and enforces a linear treatment of a contract's assets.

To express and enforce the protocols underlying a contract, we base Nomos on *session types* [Honda 1993; Honda et al. 1998, 2008] and in particular on the works that are grounded in linear logic [Balzer and Pfenning 2017; Caires and Pfenning 2010; Pfenning and Griffith 2015; Toninho et al. 2013; Wadler 2012]. Session types have been introduced to moderate bidirectional communication between concurrent message-passing processes. They can describe complex protocols of interactions between users and contracts and serve as a high-level description of the functionality of the contract. Type checking can be automated and guarantees that Nomos programs follow the given protocol. In this way, the key functionality of the contract is visible in the type, and contract development is based on the interaction of the contract with the world.

To make transaction cost in Nomos predictable and transparent, and to prevent bugs and vulnerabilities in contracts based on excessive resource usage, we apply and develop *automatic amortized resource analysis (AARA)*, a type-based technique for automatically inferring symbolic resource bounds [Carbonneaux et al. 2017; Hoffmann et al. 2011, 2017; Hofmann and Jost 2003; Jost et al. 2010]. AARA is parametric in the cost model making it directly applicable to track gas cost of Nomos contracts. A unique feature of Nomos' resource-aware type system is that it allows contracts to store gas in internal data structures to amortize the cost of resource intensive transactions. Other advantages of AARA include natural compositionality, a formal soundness proof with respect to a cost semantics, and reduction of (non-linear) bound inference to off-the-shelf LP solving.

To eliminate a class of bugs in which the internal state of a contract loses track of its assets or performs unintended transactions, Nomos integrates a linear type system [Wadler 1990] into a functional language. Linear type systems use the ideas of Girard's linear logic [Girard 1987] to ensure that certain data is neither duplicated nor discarded by a program (since they do not exhibit contraction and weakening). Programming languages such as Rust [Rus 2018] have demonstrated that substructural type systems are practical in industrial-strength languages.

We evaluate Nomos' viability for digital contract development with a case study of three typical blockchain applications and their vulnerabilities (see Section 8). The results show that the same vulnerabilities do not arise in corresponding Nomos applications. For example, session types readily express the ERC-20 standard on synthetic tokens and identify a linearity violation in one implementation of the standard. Moreover, resource-aware types prevent vulnerabilities due to out-of-gas exceptions and shared types avert dependence on transaction ordering.

Like any language, Nomos has some limitations. First, incorporating sharing into binary session types introduces the possibility of deadlocks (already present in prior work [Balzer and Pfenning 2017]). We plan to explore static [Balzer et al. 2019] and dynamic techniques for deadlock avoidance

as part of future work. Nomos also does not support dependent types [Griffith and Gunter 2013] which could be helpful in ruling out other classes of contract errors. We plan to explore adding simple arithmetic refinements in future work. We also plan to design an implementation of Nomos and address challenges regarding integration with a blockchain (see Section 9 for details).

In addition to the design of the Nomos language, we make the following technical contributions.

(1) We integrate linear session types that support controlled sharing [Balzer and Pfenning 2017; Balzer et al. 2018] into a conventional functional type system. To leave the logical foundation intact, the integration is achieved by a *contextual monad* [Toninho et al. 2013] (Section 4) that gives process expressions first-class status in the functional language. Moreover, we recast shared session types [Balzer and Pfenning 2017] to allow shared sessions to depend on linear assets (Section 6).
(2) We smoothly integrate AARA for functional programs with session types for work analysis [Das et al. 2018] to track transaction cost (Section 5).
(3) We prove the type safety of Nomos with respect to a novel asynchronous cost semantics ensuring soundness of resource bounds and that linearity is enforced. (Section 7).
(4) We provide a case study establishing the importance of each design feature in identifying security vulnerabilities of real world digital contracts (Section 8).
(5) We translated several examples used in this paper into Concurrent C0 [Willsey et al. 2016], which serves as proof-of-concept for evaluating the viability of digital contract languages based on session types. Our preliminary results indicate that the underlying ideas are robust and valuable broadly, not just specifically in the context of Nomos (Section 10).

This article introduces the key concepts of Nomos. The supplementary material formalizes the complete language with typing rules, cost semantics, and the type soundness theorem and proof.

2 NOMOS BY EXAMPLE

Nomos is a programming language based on resource-aware [Das et al. 2018] and shared [Balzer and Pfenning 2017] session types for writing digital contracts. This section introduces the main features of Nomos using a simple auction contract as an example. The subsequent sections explain each feature in technical detail.

Explicit Protocols of Interaction. Digital contracts, like ordinary contracts, follow a predefined protocol. For instance, an auction contract follows the protocol that the bidders first submit their bids to the auctioneer, and then the highest bidder receives the lot while all other bidders receive their bids back. In existing smart contract languages like Solidity [Auc 2016], this protocol is neither made explicit in the contract program nor enforced statically. Without such an explicit protocol, there is no guarantee that the parties involved in the contract will follow the protocol. As a result, contracts in these languages have to resort to explicit runtime checks to prevent undesirable behavior (e.g., bids must be placed before collecting them back). This is a common source of bugs in contracts as accounting for all possible unwanted behavior is challenging, especially in a distributed system with distrusting parties.

Contracts in Nomos, on the other hand, are typed with a *session type*, which specifies the contract's protocol of interaction. Type-checking then makes sure that the program implements the protocol defined by the session type. For instance, consider the following protocol prescribed by the auction session type (ignore the $\triangleleft$ and $\triangleright$ operators which are discussed later).

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

$$\textbf{cancel} : \triangleright^{8} \downarrow^{S}_{L} \text{auction}\}\} \qquad \% \text{ client cancelled}$$

Since there exist multiple bidders in an auction, we use a *shared* session type [Balzer and Pfenning 2017] to define the auction protocol. To guarantee that bidders interact with the auction in mutual exclusion, the session type demarcates the parts of the protocol that become a *critical section*. The $\uparrow^{S}_{L}$ type 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 the *release* of the session. Shared session types guarantee that inside a critical section there exists exactly one client, whose interaction is described by a *linear* session type.

Once a client has acquired the auction session, the auction will indicate whether it is still running (label running) or not (ended). This protocol is expressed by the internal choice type constructor ($\oplus$), describing the provider's (aka contract's) choice. An external choice ($\&$), on the other hand, leaves the choice to the client. For example, in case the auction is still running, the client can choose between placing a bid (label bid) or backing out (cancel). If the client chooses to place a bid, they have to indicate their identifier (type id), followed by a payment (type money), after which they release the session. Nomos session types allow transfer of both non-linear values that can be copied (e.g. id), using the arrow ($\rightarrow$) constructor, and linear assets, using the lolli ($\multimap$) constructor. Using a linear type to represent digital money (money) makes sure that such a value can neither be duplicated nor lost. Should the auction have ended, the client can choose to check their outcome (label collect) or back out (cancel). In the case of collect, the auction will answer with either won or lost. In the former case, the auction will send the lot (commodity being auctioned, represented as a linear type), in the latter case, it will return the client's bid. The tensor ($\otimes$) constructor is dual to $\multimap$ and denotes the transfer of a linear value from the contract to the client. The auction type additionally guarantees that a client cannot collect during the running phase, while they cannot bid during the ended phase. Thus, the contract programmer need not programmatically enforce that bids must be placed before they are collected.

In Nomos, session types are implemented by *processes*, revealing the concurrent, message-passing nature of session-typed languages. The implementation below shows the process *run* representing the running phase of the auction. It internally stores a linear hash map of bids $b : \text{hashmap}_{\text{bid}}$ and a linear lot l and offers along a shared channel. The bid type (line 1) supports querying for the stored identifier and bid value, and is offered by a process that stores this identifier and money.

```
1:  bid = &{addr : id × bid, val : money},    bids = hashmap_bid
2:  (b : bids), (l : lot) ⊢ run :: (sa : auction)
3:    sa ← run ← b l =                                  % auction contract in running phase
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 ← run ← b l                  % recurse
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. It then waits for the bidder's choice. Should the bidder choose to make a bid, the process waits to receive the bidder's identifier (line 6) followed by money equivalent to the bidder's bid (line 7). After this linear exchange, the process leaves the critical section by issuing a *detach* (line 8), matching the bidder's release request. Internally, the

process stores the pair of the bidder's identifier and bid in the data structure bids (line 9). The ended protocol of the contract is governed by a different process, 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).

Re-Entrancy Vulnerabilities. This condition is created when a client can call a contract function and potentially re-enter before the previous call is completed. In existing languages, transferring funds from the contract to the client also transfers execution control over to the client, who can then call into the same transfer function recursively, eventually leading to all funds being transferred from the contract to the client. This vulnerability was exposed by the infamous DAO attack [Siegel 2016], where $60 million worth of funds were stolen. The message passing framework of session types eliminates this vulnerability. While session types provide multiple clients access to a contract, the acquire-release discipline ensures that clients interact with the contract in mutual exclusion.

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

Resource-aware session types [Das et al. 2018] are well-suited for statically analyzing the resource cost of a process. They operate by assigning an initial potential to each process. This potential is consumed by each operation that the process executes or can be transferred between processes to share and amortize cost. The cost of each operation is defined by a cost model. Resource-aware session types express the potential as part of the session type, making the resource analysis static. For instance, in the auction contract, 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 a session is 11 (taking the max of all branches). Thus, we require the client to send 11 units of potential at the start of a session.

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

Linear Assets. Nomos integrates a linear type system that tracks the assets stored in a process. The type system enforces that these assets are never duplicated or discarded, but only exchanged between processes. The type system forbids a process to terminate while it stores any linear asset. For instance, in the auction contract, money and lot are treated as linear assets.

Bringing It All Together. A main contribution of this paper is to combine all these features in a single language while retaining type safety. To this end, we introduce four different *modes* of a channel, identifying the role of the process offering along that channel. The mode P denotes *purely*

Session Type	Continuation	Process Term	Continuation	Description
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$c.k\ ;\ P$	P	provider sends label k along c
		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

linear processes, typically amounting to linear assets or private data structures, such as b and l in the auction. The modes S and L denote *sharable processes* that are either in their shared phase or linear phase, respectively, and are typically used for contracts, such as sa and la, respectively, in the auction. The mode U, finally, denotes a *linear process that can access shared processes* and is typically used for a user, such as bidder in the auction. The mode assignment carries over into the process typing judgments (see Section 6) ascertaining certain well-formedness conditions (Section 7) on their type. This is crucial in preserving the tree structure of linear processes at run-time, establishing type safety.

3 BASE SYSTEM OF SESSION TYPES

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

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

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:

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

We will gradually introduce each concept in the remainder of this article. For now, they can be viewed as constant, and are simply threaded through by the rules.

The Curry-Howard correspondence gives each connective of linear logic an interpretation as a session type:

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

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

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

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

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

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

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

Fig. 1. Selected typing rules for process communication

Each type prescribes the kind of message that must be sent or received along a channel of that type and at which type the session continues after the exchange. Types are defined mutually recursively in a global signature, where type definitions are constrained to be *contractive* [Gay and Hole 2005]. This allows us to treat them equi-recursively [Crary et al. 1999], meaning we can silently replace a type variable by its definition for type-checking.

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

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

Table 1 provides an overview of the types along with their operational meaning. Because we adopt the intuitionistic version of linear logic, session types are expressed from the point of view of the provider. Table 1 provides the viewpoint of the provider in the first line, and that of the client in the second line for each connective. Columns 1 and 3 describe the session type and process term before the interaction respectively. 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 lolli ($\multimap$) connectives.

Internal Choice. The linear logic connective $A \oplus B$ has been generalized to n-ary labeled sum $\oplus\{\ell : A_\ell\}_{\ell \in L}$. A process that provides $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ can send any label $l \in L$ 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 $x\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$. 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 $\mathsf{proc}(c_m, w, P)$ and $\mathsf{msg}(c_m, w, M)$ denoting process P and message M, 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 5. Communication is *asynchronous*, allowing the sender $(c.l\ ;\ P)$ 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^+ so that the message l is actually represented as $(c.l\ ;\ c \leftarrow c^+)$ (read: send l along c and continue as c^+):

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

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

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

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* explained later. Therefore, no separate typing rules for messages are needed; they can be typed as processes [Balzer and Pfenning 2017].

Delegation. Nomos allows the exchange of channels over channels, sometimes called *delegation*. 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 $x \leftarrow y$ (which provides channel x) identifies channels x and y 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*.

$$(\mathsf{id}^+ C): \quad \mathsf{msg}(d_m, w', M), \mathsf{proc}(c_m, w, c_m \leftarrow d_m) \mapsto \mathsf{msg}(c_m, w + w', [c_m/d_m]M)$$
$$(\mathsf{id}^- C): \quad \mathsf{proc}(c_m, w, c_m \leftarrow d_m), \mathsf{msg}(e_l, w', M(c_m)) \mapsto \mathsf{msg}(e_l, w + w', M(d_m))$$

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

4 ADDING A FUNCTIONAL LAYER

Digital contracts combine linear channels with conventional data structures, such as integers, lists, or dictionaries to enable contracts to maintain state. To reflect and track different classes of data in the type system, we take inspiration from prior work [Pfenning and Griffith 2015; Toninho et al. 2013] and incorporate processes into a functional core via a *linear contextual monad* that isolates session-based concurrency. To this end, we introduce a separate functional context to the typing of a process. The linear contextual monad encapsulates open concurrent computations, which can be passed in functional computations but also transferred between processes in the form of *higher-order processes*, providing a uniform integration of higher-order functions and processes.

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

$$\begin{array}{rcl} \tau & ::= & \tau \to \tau \mid \tau + \tau \mid \tau \times \tau \mid \mathsf{int} \mid \mathsf{bool} \mid L^q(\tau) \\ & \mid & \{A_P \leftarrow \overline{A_P}\}_P \mid \{A_S \leftarrow \overline{A_S}\ ;\ \overline{A_P}\}_S \mid \{A_U \leftarrow \overline{A_S}\ ;\ \overline{A}\}_U \end{array}$$

The types are standard, except for the potential annotation $q \in \mathbb{N}$ in list type $L^q(\tau)$, which we explain in Section 5, 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

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

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

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

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

Fig. 2. 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.

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

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

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

The monadic *bind* operation implements process composition and acts as the elimination form for values of type $\{A_P \leftarrow \overline{A_P}\}_P$. The bind operation, written as $c_{\mathsf{P}} \leftarrow M \leftarrow \overline{d_{\mathsf{P}}}\ ;\ Q_c$, composes the process underlying the monadic term M, which offers along channel c_{P} and uses channels $\overline{d_{\mathsf{P}}}$, with Q_c, which uses c_{P}. The typing rule for the monadic bind is rule $\{\}E_{PP}$ in Figure 2. The linear context is split between the monad M and continuation Q, enforcing linearity. Similarly, the potential in the functional context is split using the sharing judgment ($\curlyvee$), explained in Section 5. The shared context Γ is empty and is discussed in Section 6. 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 is given as follows:

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

The above rule spawns the process P offering along a globally fresh channel c_{P}, and using channels $\overline{a}$. The continuation process Q acts as a client for this fresh channel c_{P}. The other two monadic types correspond to spawning a shared process $\{A_S \leftarrow \overline{A_S}\ ;\ \overline{A_P}\}_S$ and a user process $\{A_U \leftarrow \overline{A_S}\ ;\ \overline{A}\}_U$ at mode S and U, respectively. Their rules are analogous to $\{\}I_P$ and $\{\}E_{PP}$.

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

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

The type $\tau \to A$ prescribes receiving a value of type τ with continuation type A, while its dual $\tau \times A$ prescribes sending a value of type τ with continuation A. The corresponding typing rules

for arrow ($\rightarrow R, \rightarrow L$) are given in Figure 2 (rules for $\times$ are inverse). Receiving a value adds it to the functional context Ψ, while sending it requires proving that the value has type τ. Semantically, sending a value $N : \tau$ creates a message predicate along a fresh channel c_m^+ containing the value:

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

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

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

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

```
money = &{value : int × money,                          % send value
          add : money ⊸ money,                          % receive money and add it
          subtract : int → ⊕{sufficient : money ⊗ 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

```
1:  (n : int) ; (l : list_coin) ⊢ wallet :: (m : 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 ← wallet (n − n') ← l'
23:       | coins ⇒ m ← l)
```

If the *wallet* process receives the message value, it sends back the integer n, and recurses (lines 4 and 5). If it receives the message add followed by a channel of type money (line 6), it queries the value of the received money m' (line 7), stores it in v (line 8), extracts the coins stored in m' (line 9), and appends them to its internal list of coins (line 10). Similarly, if the *wallet* process receives the message subtract followed by an integer, it compares the requested amount against the stored funds. If the balance is insufficient, it send 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 over (line 21) and recurses. Finally, if the *wallet* receives the message coins, it simply sends its internal list along the offered channel.

5 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 to the global distributed ledger, i.e., blockchain. This validation is performed by *miners*, who charge fees based on the gas consumption of the transaction. This fee has to be estimated and provided by the sender prior to the transaction. If the provided amount does not cover the gas cost, the money falls to the miner, the transaction fails, and the state of the contract is reverted back. Overestimates bare the risk of high losses if the contract has flaws or vulnerabilities.

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

Functional Layer. Numerous techniques have been proposed to statically derive resource bounds for functional programs [Avanzini et al. 2015; Cicek et al. 2017; Danner et al. 2015; Lago and Gaboardi 2011; Radiček et al. 2017]. In Nomos, we adapt the work on automatic amortized resource analysis (AARA) [Hoffmann et al. 2011; Hofmann and Jost 2003] that has been implemented in Resource Aware ML (RaML) [Hoffmann et al. 2017]. RaML can automatically derive worst-case resource bounds for higher-order polymorphic programs with user-defined inductive types. The derived bounds are multivariate resource polynomials of the size parameters of the arguments. AARA is parametric in the resource metric and can deal with non-monotone resources like memory that can become available during the evaluation.

As an illustration, consider the function *applyInterest* that iterates over a list of balances and applies interest on each element, multiplying them by a constant c. An imperative version of the same function in Solidity is implemented in Section 8. We use *tick* annotations to define the resource usage of an expression in this article. We have annotated the code to count the number of multiplications. The resource usage of an evaluation of *applyInterest* b is $|b|$.

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

The idea of AARA is to decorate base types with potential annotations that define a potential function as in amortized analysis. The typing rules ensure that the potential before evaluating an

expression is sufficient to cover the cost of the evaluation and the potential defined by the return type. This posterior potential can then be used to pay for resource usage in the continuation of the program. For example, we can derive the following resource-annotated type.

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

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 resource usage of a process, Nomos features recently introduced resource-aware session types [Das et al. 2018] for work analysis. Resource-aware session types describe resource contracts for inter-process communication. The type system supports amortized analysis by assigning potential to both messages and processes. The derived resource bounds are functions of interactions between processes. As an illustration, consider the following resource-aware list interface from prior work [Das et al. 2018].

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

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

While resource-aware session types in Nomos are equivalent to the existing formulation [Das et al. 2018], our version is simpler and more streamlined. Instead of requiring every message to carry a potential (and potentially tagging several messages with 0 potential), we introduce two new type constructors for exchanging potential.

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

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

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

$\boxed{\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{g} P :: (x : A)}$ 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. 3. Selected typing rules corresponding to potential.

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

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

for Nomos processes. The non-negative number q in the judgment denotes the potential that is stored in the process. Figure 3 shows the rules that interact with the potential annotations. In the rule $\triangleleft R$, process P storing potential q receives r units along the offered channel x_m using the *get* construct and the continuation executes with $p = q + r$ units of potential. In the dual rule $\triangleleft L$, a process storing potential $q = p + r$ sends r units along the channel x_m in its context using the *pay* construct, and the continuation remains with p units of potential. The typing rules for the dual constructor $\triangleright^{r} A$ are the exact inverse. Finally, executing the tick (r) construct consumes r potential from the stored process potential q, and the continuation remains with $p = q - r$ units, as described in the tick rule in Figure 3.

Integration. Since both AARA for functional programs and resource-aware session types are based on the integration of the potential method into their type systems, their combination is natural. The two points of integration of the functional and process layer are (i) spawning a process, and (ii) sending/receiving a value from the functional layer. Recall the spawn rule $\{\}E_{PP}$ from Figure 2. A process storing potential $r = p + q$ can spawn a process corresponding to the monadic value M, if M needs p units of potential to evaluate, while the continuation needs q units of potential to execute. Moreover, the functional context Ψ is shared in the two premises as Ψ_1 and Ψ_2 using the judgment $\Psi \curlyvee (\Psi_1, \Psi_2)$. This judgment, already explored in prior work [Hoffmann et al. 2017] describes that the base types in Ψ are copied to both Ψ_1 and Ψ_2, but the potential is split up. For instance, $L^{q_1+q_2}(\tau) \curlyvee (L^{q_1}(\tau), L^{q_2}(\tau))$. The rule $\multimap L$ follows a similar pattern. Thus, the combination of the two type systems is smooth, assigning a uniform meaning to potential, both for the functional and process layer.

Remarkably, this technical device of exchanging functional values can be used to exchange non-constant potential with messages. As an illustration, we revisit the auction protocol introduced in Section 2. Suppose the bids were stored in a list, instead of a hash map, thus making the cost of collection of winnings linear in the worst case, instead of a constant. A user would then be required to send a linear potential after acquiring the contract. This can be done by sending a natural number $n : \mathsf{nat}^{q}$, storing potential $q \cdot |n|$ (like a unary list), where q is the cost of iterating over an element in the list of bids. The contract would then iterate over the first n elements of the list and refund the remaining gas if n exceeds the length. Since the auction has ended, a user can view the size of the list of bids, compute the required potential, store it in a natural number, and transfer it. It would still be possible that a user does not provide enough fuel to reach the sought-after element in the

list. However, this behavior is clearly visible in the protocol and code and out-of-gas exceptions are not possible.

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

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

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

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

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

6 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:

A_S	::=	$\uparrow^{\mathsf{S}}_{\mathsf{L}} A_L$	shared session type
A_L	::=	$\ldots \mid \downarrow^{\mathsf{S}}_{\mathsf{L}} A_S$	linear session types

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

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

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

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

This denotes a process P providing a service of type A along channel x and using functional variables in Ψ, shared channels in Γ, and linear channels in Δ. The stratification of channels into layers arises from a difference in structural properties that exist for types at a mode. Shared propositions

$$
\begin{array}{lcl}
A_P & ::= & \oplus\{\ell : A_P\}_{\ell \in L} \mid \&\{\ell : A_P\}_{\ell \in L} \mid \mathbf{1} \mid A_m \multimap_m A_P \mid A_m \otimes_m A_P \mid \tau \to A_P \mid \tau \times A_P \\
A_L & ::= & \oplus\{\ell : A_L\}_{\ell \in L} \mid \&\{\ell : A_L\}_{\ell \in L} \mid \mathbf{1} \mid A_m \multimap_m A_L \mid A_m \otimes_m A_L \mid \tau \to A_L \mid \tau \times A_L \mid \downarrow^{\mathsf{S}}_{\mathsf{L}} A_S \\
A_S & ::= & \uparrow^{\mathsf{S}}_{\mathsf{L}} A_L \\
A_U & ::= & A_P
\end{array}
$$

Fig. 4. Grammar for shared session types

exhibit weakening, contraction and exchange, thus can be discarded or duplicated, while linear propositions only exhibit exchange.

Allowing Contracts and Users 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) 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 is 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 type preservation is to prevent a process from acquiring its client, which would result in a cycle in the linear process tree. To this end, we factorize the above typing judgment according to the *three roles* that arise in digital contract programs: *contracts*, *users*, and *linear assets*. Since contracts are shared and thus can oscillate between shared and linear, we get the following four typing judgments:

$$
\begin{array}{llll}
\Psi\ ;\ \cdot\ ;\ \Delta_{\mathsf{P}} & \vdash^{g} & P :: (x_{\mathsf{P}} : A_P) & \% \quad \text{linear asset} \\
\Psi\ ;\ \Gamma\ ;\ \Delta_{\mathsf{P}} & \vdash^{g} & P :: (x_{\mathsf{S}} : A_S) & \% \quad \text{contracts in shared mode} \\
\Psi\ ;\ \Gamma\ ;\ \Delta & \vdash^{g} & P :: (x_{\mathsf{L}} : A_L) & \% \quad \text{contracts in linear mode} \\
\Psi\ ;\ \Gamma\ ;\ \Delta & \vdash^{g} & P :: (x_{\mathsf{U}} : A_U) & \% \quad \text{user process}
\end{array}
$$

The first typing judgment is for typing linear assets. These type a purely linear process P using a purely linear context Δ_{P} (without access to shared channels) and offering type A_P along channel x_{P}. The mode P of the channel indicates that a purely linear session is offered. The second and third typing judgment are for typing contracts. The second judgment shows the type of a contract process P using a shared context Γ and a purely linear channel context Δ_{P} (judgment purelin) and offering type A_S 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 judgment. The offered channel transitions to linear mode L, while the linear context may now contain channels at arbitrary modes (L, U or P). *This allows contracts to interact with other contracts without compromising type safety.* Finally, the fourth typing judgment types a user process, corresponding to an *external account* holding access to shared channels Γ and linear channels Δ, and offering at mode U.

This factorization and the fact that contracts, as the only shared processes, can only access linear channels at mode P, upholds preservation while allowing shared contract processes to rely on linear resources. Figure 4 shows the abstract syntax of types in Nomos.

Shared session types introduce new typing rules into our system, concerning the *acquire-release* constructs (see Figure 5). An acquire is applied to the shared channel x_{S} along which the shared process offers and yields a linear channel x_{L} when successful. A contract process can *accept* an acquire request along its offering shared channel x_{S}. After the accept is successful, the shared contract process transitions to its linear phase, now offering along the linear channel x_{L}.

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

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

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

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

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

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

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

Fig. 5. 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 5. The effect of releasing the linear channel x_{L} is that the continuation Q loses access to x_{L}, while a new reference to x_{S} is made available in the shared context Γ. The contract, on the other hand, detaches from the client by transitioning its offering channel from linear mode x_{L} back to the shared mode x_{S}. Operationally, the release-detach rule is inverse to the acquire-accept rule.

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

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, we formalize the process typing judgment, $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{g} P :: (x_m : A)$ which is factorized into 4 categories, depending on the mode m. This mode asserts certain well-formedness conditions on the judgment (Lemma 1). Remarkably, our process typing rules, despite being parametric in the mode, preserve these conditions.

Lemma 1 (Invariants). *The typing rules on the judgment $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{g} (x_m : A)$ preserve the following invariants i.e., if the conclusion satisfies the invariant, so do all the premises.* $\mathbf{L}(A)$ *denotes the language generated by the grammar of* A.

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

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

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

A key question then is how to type these configurations. A configuration both uses and provides a collection of channels. The typing imposes a partial order among the processes and messages, requiring the provider of a channel to appear 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 $\Sigma\ ;\ \Gamma_S \overset{E}{\models} \Omega :: (\Gamma\ ;\ \Delta)$ 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 linear channels 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_S. 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 supplementary material details the configuration typing rules.

Finally, Σ denotes a signature storing the type and function definitions. A signature is well-formed if a) every type definition $V = A_V$ is *contractive* [Gay and Hole 2005] and b) every function definition $f = M : \tau$ is well-typed according to the expression typing judgment $\Sigma\ ;\ \cdot \Vdash^p M : \tau$. The signature does not contain process definitions; any process is encapsulated inside a function using the contextual monad.

Theorem 1 (Type Preservation).

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

The preservation theorem is standard for expressions [Hoffmann et al. 2017]. For processes, we proceed by induction on the operational cost semantics and inversion on the configuration and process typing judgment.

A process $\mathrm{proc}(c_m, w, P)$ is poised if it is trying to receive a message on c_m. Dually, a message $\mathrm{msg}(c_m, w, M)$ is poised if it is sending along c_m. A configuration is poised if every message or process in the configuration is poised. Intuitively, this means that the configuration is trying to interact with the outside world along a channel in Γ or Δ. Additionally, a process can be blocked 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* $\Gamma_S \overset{E}{\models} \Omega :: (\Gamma\ ;\ \Delta)$*. 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* $\mathrm{proc}(a_L, w, P) \in \Omega$.

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

8 CASE STUDY: BLOCKCHAIN APPLICATIONS

This section evaluates the design of Nomos by discussing the implementation of several blockchain applications and typical issues that arise. In particular, we show *(i)* that session types can enforce the protocol underlying the ERC-20 token standard, *(ii)* that resource-aware types prevent out-of-gas exceptions, and *(iii)* that controlled sharing can prevent common vulnerabilities that are based on transaction-ordering.

8.1 ERC-20 Token Standard

Tokens are a representation of a particular asset or utility, that resides on top of an existing blockchain. ERC-20 [ERC 2018] is a technical standard for smart contracts on the Ethereum blockchain that defines a common list of standard functions that a token contract has to implement. The majority of tokens on the Ethereum blockchain are ERC-20 compliant. The standard requires the following functions to be implemented:

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

An ERC-20 token can be declared with the following session type in Nomos:

$$\begin{aligned}\mathsf{erc20token} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \&\{&\mathbf{totalSupply} : \mathsf{int} \times \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{erc20token},\\ &\mathbf{balanceOf} : \mathsf{id} \to \mathsf{int} \times \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{erc20token},\\ &\mathbf{transfer} : \mathsf{id} \to \mathsf{id} \to \mathsf{int} \to \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{erc20token},\\ &\mathbf{transferFrom} : \mathsf{id} \to \mathsf{id} \to \mathsf{int} \to \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{erc20token},\\ &\mathbf{approve} : \mathsf{id} \to \mathsf{id} \to \mathsf{int} \to \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{erc20token},\\ &\mathbf{allowance} : \mathsf{id} \to \mathsf{id} \to \mathsf{int} \times \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{erc20token}\}\end{aligned}$$

The type ensures that the token follows the protocol underlying the 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. The transferFrom label has the same functionality. If the contract receives approve, it receives the two identifiers and the value, and updates the allowance internally. Finally, this allowance can be checked by issuing the allowance label, and sending the owner's and spender's identifier.

A programmer can design their own implementation (contract) of the erc20token session type. Internally, the contract relies on custom coins created and named by its owner and used exclusively for exchanges among private accounts. These coins can be minted by a special transaction that can only be issued by the owner, and creates coins out of thin air (consuming gas to create coin).

Depending on the functionality intended by the owner, they can employ different types to represent their coins. For instance, choosing type $\mathbf{1}$, the multiplicative unit from linear logic will allow both creation and destruction of coins "for free". A *mint-one* process, typed as $\cdot \vdash \mathit{mint\text{-}one} :: (c : \mathbf{1})$ can create coin c out of thin air, and a *burn-one* process, typed as $(c : \mathbf{1}) \vdash \mathit{burn\text{-}one} :: (d : \mathbf{1})$ will destroy the coin c. Nomos' linear type system enforces that the coins are treated linearly modulo minting and burning. Any transaction that does not involve minting or burning ensures linearity of these coins.

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

```
account = &{addr : id × account,                    %  send identifier
            add : plcoins ⊸ account,                %  receive pl coins and add internally
            subtract : int → plcoins ⊗ account}     %  receive integer, send pl coins
```

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

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

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

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

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

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

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

Hacker Gold (HKG) Token. The HKG token is one particular implementation of the ERC-20 token specification. Recently, a vulnerability was discovered in the HKG token smart contract based on a typographical error leading to a re-issuance of the entire token [HKG 2017].

The typographical error in the contract came about when updating the receiver's balance during a transfer. Instead of writing balance += value, the programmer mistakenly wrote balance =+ value (semantically meaning balance = value). Moreover, while testing this error was missed, because the first transfer always succeeds (since the two statements are semantically equivalent when balance = 0). Nomos' type system would have caught the linearity violation in the latter statement that drops the existing balance in the recipient's account.

8.2 Out-of-Gas Vulnerabilities

A broad family of contract vulnerabilities concerns *out-of-gas* behavior. This section describes some common patterns causing such vulnerabilities with examples and evaluates Nomos' ability to detect them. As mentioned before, gas is the fuel of computation in several blockchains such as Ethereum that users provide together with a transaction. If the computation exceed its allotted amount of gas, an out-of-gas exception is raised and the transaction is aborted. A contract that does not correctly handle the possible abortion of a transaction is at risk of a gas-focused vulnerability. The examples presented have been inspired from prior work [Grech et al. 2018].

Unbounded Mass Operations. This can be caused by a loop whose iterations cannot be bounded at design time because they depend on the user input or the state of the contract. Such a loop can become economically too expensive to perform or even exceed the *block gas limit*, which would make the contract unusable in Ethereum. Consider the contract snippet below in Solidity.

```
function applyInterest() returns (uint) {
  for (uint i = 0; i < accounts.length; i++)
    // apply 5 percent interest
    accounts[i].balance = accounts[i].balance * 105 / 100;
  return accounts.length; }
```

As the number of accounts increases, the gas requirement for applyInterest rises. If an attacker can control the number of accounts then this loop can be used for a *denial-of-service* attack.

Nomos can infer the linear gas cost for applyInterest. Suppose the gas cost for executing one loop iteration is c (ignoring other constant costs). Nomos will assign the resource-aware type $L^c(\mathsf{int})$ to account, assigning a potential of c to each element in the list, which is consumed to perform each loop iteration. Such *gas sharing schemes* would be sufficient to reimburse all unsuccessful participants in an auction and proved sound by a resource-aware type derivation. The onus of supplying the gas cost for executing the loop does not fall on one user, and is amortized over all users of the contract. This avoid the denial-of-service vulnerability of this contract.

Nomos also supports more complex gas sharing schemes that would allow a function like applyInterest to be executed an unlimited number of times. One possibility would be to internally store the accounts in two lists. The list of *active accounts* would have type $L^c(\mathsf{int})$ and carry potential c per element to cover the cost of adding interest. The list of *inactive accounts* would have type $L^0(\mathsf{int})$ and not carry potential. The function applyInterest would operate only on the active accounts and move all accounts to the inactive list during its iteration. The contract would allow users to move their account back to the active list, charging the additional gas cost c to pay for the potential of the additional list element. Again, the type system would prove that the gas cost of each transaction is indeed constant.

Non-Isolated External Calls. A contract may also have out-of-gas vulnerabilities because its invoking an external function that may throw an out-of-gas exception. This occurs when a programmer has not considered an external call extensively, as is often the case with implicit calls. For instance, on the Ethereum blockchain, sending Ether involves calling a fallback function on the recipient's side. All it takes for an attacker is to provide a fallback function that runs out of gas.

All calls in Nomos, whether internal within the same account or external, are checked for sufficient potential. The type of a function (on the turnstile, and the potential of the arguments) indicates precisely the gas cost of the function call. The caller must have enough potential for both the call and the continuation that handles the result of the call. This is expressed, for instance, in the $\{\}E_{PP}$ rule in Figure 2, where p is the potential for the call, and q is the potential for the

continuation. Type checking guarantees that any function implementation does not run out of gas during execution, thus protecting all external calls against this out-of-gas vulnerability.

8.3 Transaction-Ordering Dependence

A common source of security bugs in smart contracts is when they depend on the order in which the transactions are executed in the contract. Consider a scenario where a blockchain is in some state and a new added block contains two transactions invoking the same contract. In such a scenario, users have uncertain knowledge of which state the contract is at when their individual invocation is executed. Only the miner who mines the new block can decide the order of the transactions. If the outcome of either of the two transactions depends on the order in which they are executed, then the contract suffers from a transaction-order dependent vulnerability.

Consider a puzzle contract [Luu et al. 2016] which rewards users who solve a computational puzzle and submit the solution to the contract. 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 winnings. The following code presents the relevant aspects of the implementation in Solidity.

```
contract Puzzle {
  ...
  address public owner;      // contract owner
  uint public reward;        // reward value
  ...
  function update() {
    if (msg.sender == owner){
      owner.send(reward);            // owner sends reward to contract
      reward = msg.value; }}
  ...
  function submit(){
    if (verify(msg.data)) {          // verify the solution
      msg.sender.send(reward); }}}   // send reward to user
```

A malicious owner of the contract can listen on the network to check for users submitting solutions to the contract. If they encounter such a transaction, they can issue their own update transaction to update the reward and make it 0 (or a smaller amount). With a certain chance, the update transaction can execute before the submit, and the owner can enjoy a free solution to the puzzle. Thus, the transaction-order dependency can be exploited.

In Nomos, the contract can be implemented to offer the following type

$$\begin{aligned}\text{puzzle} = \uparrow^{S}_{L} \,\&\{\, &\textbf{update} : \text{id} \to \text{money} \multimap \downarrow^{S}_{L} \text{puzzle},\\ &\textbf{submit} : \text{int} \times \&\{\, \textbf{success} : \text{solution} \to \text{money} \otimes \downarrow^{S}_{L} \text{puzzle},\\ &\qquad\qquad \textbf{failure} : \downarrow^{S}_{L} \text{puzzle}\}\}\end{aligned}$$

The contract still supports the two transactions, one to update the reward, where it receives the identifier, verifies that the sender is the owner, receives money from the sender, and terminates the session. 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 the guard 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.

The acquire-release discipline along with the guarded transaction prevents the exploitation of the ordering dependent vulnerability. Since the user *acquires* the contract before submitting the solution, the user holds exclusive access to the contract data. This data cannot be modified by any

other user, even the owner of the contract during this *critical section*. Thus, if the guard condition succeeds, the user is guaranteed to receive their expected winnings.

9 COMPUTATION MODEL AND LIMITATIONS

Although Nomos has been designed to be applicable for implementing general digital contracts, we provide a high-level outline of how Nomos could be implemented on a blockchain. We also highlight the main limitations of the language.

Nomos on a Blockchain. To describe a possible implementation of Nomos, we assume a blockchain like Ethereum that contains a list of Nomos contracts $C_1, \ldots, C_n$ together with their type information $\Psi^i \;;\; \Gamma^i \;;\; \Delta^i_{\mathrm{P}} \vDash^{q_i} C_i :: (x^i_{\mathrm{S}} : A^i_S)$. The functional context Ψ^i types the contract data, while the shared context Γ^i types the shared contracts that C_i refers to, and the linear context Δ^i_{P} types the contract's linear assets. We allow contracts to carry potential given by the annotations q_i and the potential defined by the annotations in Ψ^i and Δ^i_{P}. This potential is useful to amortize gas cost over different transactions but might be challenging to implement if the gas price fluctuates. The channel names x^i_{P} of the contracts have to be globally unique and we assume the existence of a mechanism that produces fresh names.

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

$$\Psi \;;\; \Gamma \;;\; \Delta \vDash^{g} P :: (x_{\mathrm{U}} : A_U)$$

Here, $\Gamma = x^1_{\mathrm{S}} : A^1_S, \ldots, x^n_{\mathrm{S}} : A^n_S$ contains references to the shared channels offered by Nomos contracts. The functional context Ψ and the linear context Δ store the state of the client and specify the types of the arguments that have to be provided by the user as part of the transaction.

Like in Ethereum or Bitcoin, we assume a mechanism that would sequentialize and queue transaction requests. When selecting a request, a miner first creates and type-checks a configuration using the submitted type information, user process P, contracts $C_1, \ldots, C_n$, and the submitted arguments. The gas cost of the transaction is statically bounded by the potential given by q, Ψ, Δ, and the submitted arguments. If we allow amortization then the potential in the contracts C_i's is also available to cover the gas cost. This internal potential is not up for grabs for the user but can only be accessed according to the protocol that is given in the contract session type.

A successful transaction will lead to the contract detaching from the user, thereby terminating the session. If the user received linear resources during the transaction, they must store it in their *wallet* process (which stores the user's assets) before terminating (required by the typechecker). The type system ensures that the contract channels are equi-synchronizing. In this way, it is guaranteed that the next user transaction finds the shared data in a well-formed state. 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.

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

Deadlocks. The only language specific reason a transaction can fail is a deadlock in the user code. Our progress theorem accounts for the possibility of deadlocks. Deadlocks may arise due to

cyclic interdependencies on the contracts users attempt to acquire. While it is of course desirable to rule out deadlocks, we felt that this is orthogonal to the design of Nomos. Any extensions for shared session types that prevent deadlocks (e.g., [Balzer et al. 2019]) will be readily transferable to our setting. Another possibility is to employ dynamic deadlock detection.

10 PRELIMINARY EVALUATION

To evaluate the viability of our approach, we implemented several digital contracts in Concurrent C0 [Balzer and Pfenning 2017; Willsey et al. 2016], a type-safe C-like imperative language with support for session types. The implementation uses an asynchronous semantics, and supports functional variables, shared and linear channels. Although Concurrent C0 does not support resource-aware types, the language is similar enough to Nomos to draw conclusions about the efficiency of its session-type based approach, especially since the implemented contracts are not imperative.

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

Contracts. In the evaluation we use three different contract implementations, namely *auction*, *voting* and *bank account*. The auction contract was already introduced in Section 2.

The voting contract provides a ballot type.

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

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

The banking contract provides a baccount type, as follows.

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

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

Evaluation. We evaluate the execution time of each contract, varying number of transactions. Figure 6 plots the execution time in seconds (y-axis) vs the size of the input (x-axis). The experiments were performed on an Intel Core i5-5250U processor with 16 GB of main memory.

- Auction : We use the total number of bidders in the auction as input, varying them from 100 to 5000 in steps of 100. Each client first places a bid, followed by the winner of the auction being declared, and finally each client collects either the lot or their money.

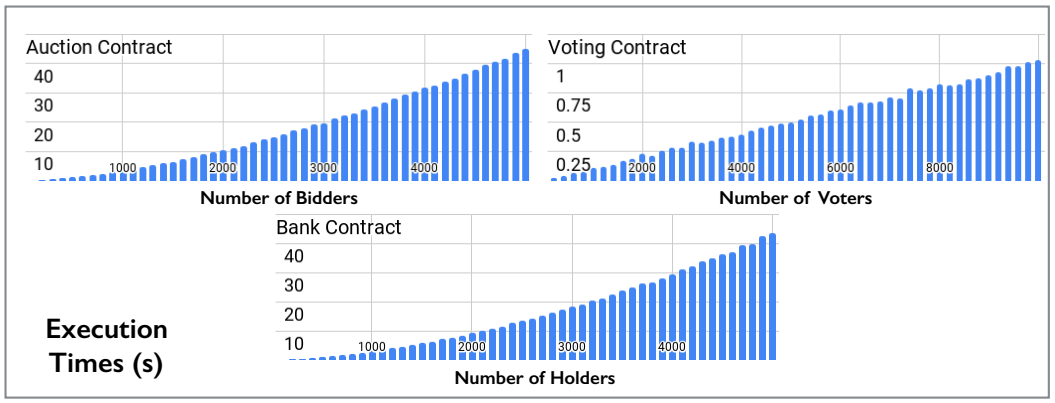

Fig. 6. Contract execution times

- Voting : We use the total number of voters as the input size, varying them from 200 to 10000 in steps of 200. Each voter sends their vote to the contract, and at the end, the winner of the election is determined.
- Bank : We use the total number of account holders as input size, varying them from 100 to 5000 in steps of 100. Each client first creates their account, then makes a deposit followed by a withdrawal, and finally checks their balance.

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

11 RELATED WORK

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

Ethereum Blockchain. Existing smart contracts on Ethereum are predominantly implemented in Solidity [Auc 2016], a statically typed object-oriented language influenced by Python and Javascript. Contracts in Solidity are similar to classes containing state variables and function declarations. However, the language provides no information about the resource usage of a contract. Languages like Vyper [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.

Other Blockchains. 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. Liquidity [Liq 2018] is a high-level language that complies with the security restrictions of Michelson. Scilla [Sergey et al. 2018] is an intermediate-level language where contracts are structured as communicating automata providing a continuation-passing style computational model to the language semantics. However, none of these languages describe and enforce communication protocols statically.

Static Analysis. Analysis of smart contracts has received substantial attention recently due to the security vulnerabilities that can be exploited by malicious users to steal funds, lock the contract or produce other unintended behaviors. KEVM [Hildenbrandt et al. 2018] provides a fully executable formal semantics of EVM (Ethereum Virtual Machine) bytecode in $\mathbb{K}$ framework [Rosu 2017]. It also 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. However, in contrast with Nomos, where guarantees are proved by a soundness proof of the type system, static analysis techniques are unsound, as they 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]. Type systems for static resource bound analysis for sequential programs have been extensively studied. Automatic amortized resource analysis (AARA) has been introduced as a type system to derive linear [Hofmann and Jost 2003] and polynomial bounds [Hoffmann et al. 2017] for functional programming languages. Resource usage has also previously been explored separately for the purely linear process layer [Das et al. 2018], but were never combined with shared session types or integrated with the functional layer.

12 CONCLUSION

We have described the programming language Nomos and its type-theoretic foundation. Nomos builds on linear logic, shared session types, and automatic amortized resource analysis to address the additional challenges that programmers are faced with when implementing digital contracts. Our main contributions are the design of Nomos' multi-layered resource-aware type system and its type soundness proof. The new type system may find applications beyond digital contracts, for example, for analyzing the complexity of distributed and concurrent systems.

We plan to design an efficient implementation of Nomos, with a focus on the practical aspects. Currently, Nomos does not support automatic inference of resource bounds or enjoy deadlock

freedom. We plan to alleviate these limitations while maintaining type safety. We also plan to explore the scope of refinement session types [Gommerstadt et al. 2018] in expressing and verifying correctness of contracts against their specifications. There are also questions about integrating Nomos into a blockchain. These include the exact cost model, fluctuation of gas prices and potential compilation to a lower-level language. We plan to answer these questions together with the blockchain community, and design features to assist programmers in implementing smart contracts.

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. The Rust Programming Language. https://doc.rust-lang.org/book.

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.

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, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *29th International Conference on Concurrency Theory (CONCUR)*. To apper.

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.

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

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

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

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

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.

Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2018. Session-Typed Concurrent Contracts. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 771–798.

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

Dennis Griffith and Elsa L. Gunter. 2013. LiquidPi: Inferrable Dependent Session Types. In *NASA Formal Methods*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–197.

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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.

Grigore Rosu. 2017. K - A Semantic Framework for Programming Languages and Formal Analysis Tools. In *Dependable Software Systems Engineering (NATO Science for Peace and Security)*, Doron Peled and Alexander Pretschner (Eds.). IOS Press.

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, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR* abs/1801.00687 (2018). arXiv:1801.00687 http://arxiv.org/abs/1801.00687

David Siegel. 2016. Understanding The DAO Hack for Journalists. https://medium.com/@pullnews/understanding-the-dao-hack-for-journalists-2312dd43e993.

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

Philip Wadler. 1990. Linear Types Can Change the World!. In *IFIP TC 2 Working Conference on Programming Concepts and Methods*. North, 546–566.

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

Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2016. Design and Implementation of Concurrent C0. In *Fourth International Workshop on Linearity*.

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