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

Abstract

Programming digital contracts comes with unique challenges, which include describing 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 strong static guarantees match the domain-specific requirements and facilitate contract development. To describe 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 demonstrate the practicability of Nomos' session-type–based design, we implemented three digital contracts in an existing prototype language. Experiments indicate that the performance of the contracts is adequate for scaling to thousands of users and interactions.

1 Introduction

Digital contracts are computer protocols that describe and enforce the execution of a contract. With the rise of blockchains and cryptocurrencies such as Bitcoin [38], Ethereum [48], and Tezos [24], digital contracts have become popular in the form of smart contracts, which provide potentially distrusting parties with programmable money and an enforcement mechanism that does not rely on third parties. Smart contracts have been used to implement auctions [1], investment instruments [37], insurance agreements [31], supply chain management [34], and mortgage loans [36]. In general, digital contracts hold the promise to reduce friction, lower cost, and broaden access to financial infrastructure.

Smart contracts have not only shed light on the benefits of digital contracts but also on their potential risks. Like all software, smart contracts can contain bugs and security vulnerabilities [8], which can have direct financial consequences. A well-known example, is the attack on The DAO [37], resulting in a multi-million dollar theft by exploiting a contract

vulnerability. Maybe even more important than the direct financial consequences is the potential erosion of trust as a result of such failures.

Contract languages today are derived from existing general-purpose languages like JavaScript (Ethereum's Solidity [1]), Go (in the Hyperledger project [13]), or OCaml (Tezos' Liquidity [7]). While this makes contract languages look familiar to software developers, it is inadequate to accommodate the domain-specific requirements of digital contracts.

- Instead of centering contracts on their interactions with users, the high-level protocol of the intended interactions with a contract is buried in the implementation code, hampering understanding, formal reasoning, and trust.
- Resource (or *gas*) usage of digital contracts is of particular importance for transparency and consensus. However, obliviousness of resource usage in existing contract languages makes it hard to predict the cost of executing a contract and prevent denial-of-service vulnerabilities.
- Existing languages fail to enforce linearity of assets, endangering the validity of a contract when assets get duplicated or deleted, accidentally or maliciously [35].

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

To express and enforce the protocols underlying a contract, we base Nomos on *session types* [28–30] and in particular on the works that are grounded in linear logic [10, 14, 39, 44, 46]. 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 centered on the interaction of the contract with the world.

In the future, typed Nomos code could safely interact with untyped or untrusted code through monitors, which can be automatically synthesized from the session type [23].

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 further develop *automatic amortized resource analysis (AARA)*, a type-based technique for automatically inferring symbolic resource bounds [15, 25–27, 32]. AARA is parametric in the cost model, which makes it directly applicable to track gas consumption of Nomos contracts. Other advantages of the technique 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 [45] into a functional language. Linear type systems use the ideas of Girard's linear logic [22] to ensure that certain data is neither duplicated nor discarded by a program. Programming languages such as Rust [5] have demonstrated that substructural type systems are practical in industrial-strength languages. Moreover, linear types are compatible with session types, which are themselves based on linear logic [10, 14, 39, 44, 46].

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 [10, 11] into a conventional functional type system. To leave the logical foundation intact, the integration is achieved by a *contextual monad* [44] (Section 4) that gives process expressions first-class status in the functional language. Moreover, we recast shared session types [10] to accommodate the explicit notions of *contracts* and *clients* (Section 5).

2. We smoothly integrate AARA for functional programs with session types for work analysis [20] (Section 6).

3. We prove the type soundness of Nomos with respect to a novel asynchronous cost semantics using progress and preservation (Section 7).

4. We translated all examples used in this paper into Concurrent C0 [47], which serves as proof-of-concept for evaluating the performance of digital contract languages based on session types. Our preliminary results indicate that the performance of language based on session types is adequate for implementation of digital contracts (Section 9).

This article illustrates the key concepts and functionality 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 [20] and shared [10] session types for writing safe digital contracts. This section uses a simple auction contract to showcase the most significant features of the language. 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 [1], this protocol is neither made explicit in the contract program nor enforced statically. Without such an explicit protocol, there is no guarantee that the parties involved in the contract will follow the protocol. As a result, contracts in these languages have to resort to explicit runtime checks to prevent undesirable behavior. This is a common source of bugs in contracts as accounting for all possible unwanted behavior is challenging, especially in a distributed system with distrusting parties.

Contracts in Nomos, on the other hand, are typed with a *session type* [10, 14, 28–30, 39, 44, 46], which specifies the contract's protocol of interaction. Type-checking then makes sure that the program implements the protocol defined by the session type correctly. For instance, consider the following protocol prescribed by the auction session type (ignore the annotations on ◁ and ▷ , discussed later).

$$\uparrow_L^S \triangleleft^{11} \oplus \{ \text{running} : \&\{ \text{bid} : \text{id} \supset \text{money} \multimap \triangleright^1 \downarrow_L^S \text{ auction},$$
$$\text{cancel} : \triangleright^8 \downarrow_L^S \text{ auction} \},$$
$$\text{ended} : \&\{ \text{collect} : \text{id} \supset$$
$$\oplus\{ \text{won} : \text{lot} \otimes \triangleright^3 \downarrow_L^S \text{ auction},$$
$$\text{lost} : \text{money} \otimes \downarrow_L^S \text{ auction} \},$$
$$\text{cancel} : \triangleright^8 \downarrow_L^S \text{ auction} \}\}$$

Since there exist multiple bidders in an auction, we use a *shared* session type [10] 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 $\uparrow_L^S$ type modality denotes the beginning of a critical section, the $\downarrow_L^S$ modality its end. Programmatically, $\uparrow_L^S$ translates into an *acquire* of the auction session and $\downarrow_L^S$ into the *release* of the session. Shared session types guarantee that inside a critical section there exists exactly one client, whose interaction is described by a *linear* session type.

Once a client has acquired the auction session, the auction will indicate whether it is still running (running) or not (ended). This protocol is expressed by the internal choice type constructor (⊕), 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 lottery is still running, the client can choose between placing a bid (bid) or backing out (cancel). If the client chooses to place a bid, they have to indicate their identifier, followed by a payment, after which they release the session. Nomos session types allow exchange of both functional values (e.g. id),

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

Figure 1. Auction process in running and ended mode

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

Figure 1 implements the contract providing session type auction. In Nomos, session types are implemented by *processes*, revealing the concurrent, message-passing nature of session-typed languages. The implementation shows the process *run* representing the running auction. It first *accepts* an acquire request by a client (line 3) and then sends the message running (line 4) indicating the auction status. The process then waits for the clients choice. Should the client choose to make a bid, the process waits to receive the client's identifier (line 6) followed by money equivalent to the client's bid (line 7). After this linear exchange, the process leaves the critical section by issuing a *detach* (line 8), matching the client's release request. Internally, the process stores the pair of the client's identifier and their bid in the data structure bids (lines 9 - 11), and the total funds of the auction as a linear resource provided by channel M of type money (lines 12 - 13). The ended protocol of the contract is governed by a different process, responsible for distributing the bids back to the clients. The contract transitions to the ended state when the number of bidders reaches a threshold (stored in auction). The supplementary material provides the complete implementation of the auction contract.

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 [37], where $60 million worth of funds were stolen. The message passing framework of session types eliminates this vulnerability. While session types provide multiple clients access to a contract, the acquire-release discipline ensures that clients interact with the contract in mutual exclusion and according to the protocol defined by the type.

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

Resource-aware session types [20] are adept for statically analyzing the resource cost of a process. They operate by assigning an initial potential to each process. This potential is consumed by each operation that the process executes or can be transferred between processes to share and amortize cost. The cost of each operation is defined by a cost model. Resource-aware session types express the potential as part of the session type, making the resource analysis static. For instance, in the auction contract, we can require the client to pay potential for the operations that the contract must execute, both while placing and collecting their bids. If the cost model assigns a cost of 1 to each contract operation, then the maximum cost of a session is 11 (taking the max of all branches). Thus, we require the client to send 11 units of potential at the start of a session.

The ◃ type constructor prescribes that the client must send potential to the client. The amount of potential is marked as a superscript to ◃. Thus, $◃^{11}$ 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 ▹ 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 $▹^8$ 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

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 $w : A$ along c
		$y \leftarrow$ recv $c \; ; \; Q_y$	$[w/y]Q_y$	client receives channel $w : A$ along c
$c : A \multimap B$	$c : B$	$y \leftarrow$ recv $c \; ; \; P_y$	$[w/y]P_y$	provider receives channel $w : A$ along c
		send $c \; w \; ; \; Q$	Q	client sends channel $w : S$ along c
$c : \mathbf{1}$	–	close c	–	provider sends end along c
		wait $c \; ; \; Q$	Q	client receives end along c

Table 1. Binary session types

returned to the sender. If the cost model assigns a cost to each operation as equivalent to their gas cost, the total initial potential of a process plus the potential it receives during a session reflects the upper bound on the gas usage.

Linear Resources   Nomos integrates a linear type system that tracks the resources stored in a process. The type system enforces that these resources are never duplicated or discarded, but only exchanged between processes. The type system forbids a process from terminating while it stores any linear resource. For instance, in the auction contract, money and lot are treated as linear resources. Technically, such resources are handled in the same way as channels.

Bringing It All Together   A main contribution of this paper is combining these features in a single language while retaining type safety. We also introduce *modes* for each channel; i) P for purely linear channels (mode for M and l in auction), ii) S for a shared session type (mode for sa), iii) l for a shared session type in linear mode (mode for la), and iv) C for a channel offered by a client process. The mode of a channel assigns a specific typing judgment (described in Section 5) to the process offering it. This prevents the linear channels from forming a cycle (which would break linearity), and is crucial while proving preservation.

3   Base System of Session Types

Nomos builds on linear session-types for message-passing concurrency [14, 28–30, 46] and, in particular, on the line of works that have a logical foundation [10, 14, 39, 44] due to the existence of a Curry-Howard correspondence between linear logic and the session-typed π-calculus [14, 46]. Linear logic [22] 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:

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

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:

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

Each type prescribes the kind of message that must be sent or received along a channel of this 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* [21]. This allows us to treat them equi-recursively [18], meaning we can silently replace a type variable by its definition for type-checking.

Following previous work on session types [39, 44], the process expressions of Nomos are defined as follows.

$$P \quad ::= x.l_k \; ; \; P \mid \text{case } x \; (l_i \Rightarrow P_i)_{i \in I} \mid x \leftarrow y$$
$$\mid \text{close } x \mid \text{wait } x \; ; \; P \mid \text{send } x \; w \; ; \; P \mid y \leftarrow \text{recv } x \; ; \; P$$

Table 1 provides an overview of the types along with their operational meaning. Because we adopt the intuitionistic version of linear logic, session types are expressed from the point of view of the provider. Table 1 provides the viewpoint of the provider in the first line, and that of the client in the second line for each connective. Columns 1 and 3 describe the session type and process term before the interaction respectively. Similarly, columns 2 and 4 describe the type and term after the interaction. Finally, the last column describes

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

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{g} P :: (x_m : A_n) \qquad (k \in L)}{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{g} x_m.n \; ; \; P :: (x_m : \oplus\{\ell : A_\ell\}_{\ell \in L})} \oplus R \qquad \frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : A_\ell) \vdash^{g} Q_\ell :: (z_k : C) \qquad (\forall \ell \in L)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vdash^{g} \text{case } x_m \; (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z_k : C)} \oplus L$$

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (y_P : A) \vdash^{g} P :: (x_m : B)}{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{g} y_P \leftarrow \text{recv } x_m \; ; \; P :: (x_m : A \multimap B)} \multimap R \qquad \frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : B) \vdash^{g} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (w_P : A), (x_m : A \multimap B) \vdash^{g} \text{send } x_m \; w_P \; ; \; Q :: (z_k : C)} \multimap L$$

$$\frac{q = 0}{\Psi \; ; \; \Gamma \; ; \; \cdot \vdash^{g} \text{close } x_m :: (x_m : \mathbf{1})} 1R \quad \frac{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{g} Q :: (z_k : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_m : \mathbf{1}) \vdash^{g} \text{wait } x_m; Q :: (z_k : C)} 1L \quad \frac{q = 0}{\Psi \; ; \; \Gamma \; ; \; (y_k : A) \vdash^{g} x_m \leftarrow y_k :: (x_m : A)} \text{fwd}$$

Figure 2. Selected typing rules for process communication

the provider and client action. Figure 2 provides the corresponding typing rules. As illustrations of the statics and semantics, we explain internal choice and forwarding.

Internal Choice   A process that provides $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ can send any label $n \in L$ along x and then continues by providing $x : A_n$. The corresponding process term is written as $(x.n \; ; \; P)$, 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 2 (parametric in mode m).

The operational semantics is formalized as a system of *multiset rewriting rules* [16]. We introduce semantic objects $\text{proc}(c_m, w, P)$ and $\text{msg}(c_m, w, M)$ 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 6. Communication is *asynchronous*, allowing the sender $(c.n \; ; \; P)$ to continue with P, without waiting for n 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 n is actually represented as $(c.n \; ; \; c \leftarrow c^+)$ (read: send n along c and continue as c^+):

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

Receiving the message n corresponds to selecting branch Q_n and substituting continuation c^+ for c:

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

The message $\text{msg}(c, w, c.k \; ; \; c \leftarrow c^+)$ is just a particular form of process, where $c \leftarrow c^+$ is *forwarding* explained below. Therefore, no separate typing rules for messages are needed; they can be typed as processes [10].

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 2 and corresponds to the logical rule of *identity*.

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 and coins with conventional data structures, such as integers, lists, or dictionaries to enable contracts to maintain state. For instance, the auction contract introduced in Section 2 contains a list of bids that is not treated linearly.

To reflect and track different classes of data in the type system, we take inspiration from prior work [39, 44] 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.

$$\tau \quad ::= \quad \tau \rightarrow \tau \mid \tau + \tau \mid \tau \times \tau \mid \text{int} \mid \text{bool} \mid \text{list}_\tau^q$$
$$\mid \quad \{A_P \leftarrow \overline{A_P}\}_P \mid \{A_S \leftarrow \overline{A_P}\}_S \mid \{A_C \leftarrow \overline{A_S} \; ; \; \overline{A}\}_C$$

The types are standard, except for the potential annotation $q \in \mathbb{N}$ in list types, which we explain in Section 6, and the contextual monadic types in the last line, which are the topic of this section. The expressivity of the types and and terms in the functional layer are not important for the development in this paper. Thus, we do not formally define functional terms M but assume that they have the expected term formers such as function abstraction and application, type constructors, and pattern matching. We also define a standard type judgment for the functional part of the language.

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

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

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

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

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

Figure 3. Typing rules corresponding to the functional layer.

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

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

The monadic *bind* operation implements process composition and acts as the elimination form for values of type $\{A_P \leftarrow \overline{A_P}\}_P$. The bind operation, written as $c_\mathsf{P} \leftarrow M \leftarrow \overline{d_\mathsf{P}} \; ; \; Q_c$, composes the process underlying the monadic value M, which offers along channel c_P and uses channels $\overline{d_\mathsf{P}}$, with Q_c, which uses c_P. The typing rule for the monadic bind is rule $\{\}E_{PP}$ in Figure 3. The linear context is split between the monad M and continuation Q, enforcing linearity. Similarly, the potential in the functional context is split using the sharing judgment ($\curlyvee$), explained in Section 6. The shared context Γ is empty and will be discussed in Section 5. 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 other two monadic types correspond to spawning a shared process ($\{A_S \leftarrow \overline{A_P}\}_S$) and a client process ($\{A_C \leftarrow \overline{A_S} \; ; \; \overline{A}\}_C$) at mode S and C respectively. Their rules are analogous to $\{\}I_P$ and $\{\}E_{PP}$.

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

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

The type $\tau \supset A$ prescribes receiving a value of type τ with continuation type A, while its dual $\tau \wedge A$ prescribes sending a value of type τ with continuation A. The corresponding

typing rules for arrow ($\supset R, \supset L$) are given in Figure 3 (rules for $\wedge$ are inverse). Receiving a value adds it to the functional context Ψ, while sending it requires proving that the value has type τ. Again, we defer the discussion of the resource annotation on the turnstile to later.

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

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

The type supports addition and subtraction, and querying for value. The abstraction is provided by a *wallet* process that internally stores a list of coins and an integer representing its value. The process is typed as

$$(n : \mathsf{int}) \; ; \; (l : \mathsf{list}_{\mathsf{coin}}) \vdash wallet :: (M : \mathsf{money})$$

The *wallet* treats $l : \mathsf{list}_{\mathsf{coin}}$ linearly, while the integer n is a functional value, and can be duplicated or discarded. The type coin stands for a unit of currency and can be added to the type system as an abstract channel type that does not allow for interactions.

5 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 [10]. 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* [12, 39, 42], session types are stratified into a linear and shared layer with two *adjoint modalities* going back and forth between them:

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

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

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

$$A_P \quad ::= \quad V \mid \oplus\{\ell : A_P\}_{\ell \in L} \mid \&\{l_i : A_P\}_{i \in I} \mid \mathbf{1}$$
$$\mid \quad A_P \multimap A_P \mid A_P \otimes A_P \mid \tau \supset A_P \mid \tau \wedge A_P$$
$$A_L \quad ::= \quad V \mid \oplus\{\ell : A_L\}_{\ell \in L} \mid \&\{l_i : A_L\}_{i \in I} \mid \mathbf{1}$$
$$\mid \quad A_P \multimap A_L \mid A_P \otimes A_L \mid \tau \supset A_L \mid \tau \wedge A_L \mid\downarrow_{\mathsf{L}}^{\mathsf{S}} A_S$$
$$A_S \quad ::= \quad \uparrow_{\mathsf{L}}^{\mathsf{S}} A_L$$
$$A_C \quad ::= \quad A_P$$

Figure 4. Grammar for shared session types

to be released back to the same type at which the process was previously acquired, should it ever have been acquired.

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

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

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

Allowing Shared Contracts to Rely on Linear Resources
As exemplified by the auction contract, a digital contract typically amounts to a process that is shared at the outset, but oscillates between shared and linear to interact with clients, one at a time. Crucial for this pattern is the ability of a contract to maintain its linear resources (e.g., money) regardless of its mode. Unfortunately, current shared session types [10] do not allow a shared process to rely on any linear resources, requiring any linear resources to be consumed before becoming shared. This precaution is logically motivated [40] 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*, *clients*, and *linear assets*. Since contracts are shared and thus can oscillate between shared and linear, we get the following four typing judgments:

$$\Psi \; ; \; \cdot \; ; \; \Delta_{\mathsf{P}} \quad \vdash^{g} \quad P :: (x_P : A_P)$$
$$\Psi \; ; \; \cdot \; ; \; \Delta_{\mathsf{P}} \quad \vdash^{g} \quad P :: (x_S : A_S)$$
$$\Psi \; ; \; \cdot \; ; \; \Delta_{\mathsf{P}} \quad \vdash^{g} \quad P :: (x_L : A_L)$$
$$\Psi \; ; \; \Gamma \; ; \; \Delta \quad \vdash^{g} \quad P :: (x_C : A_C)$$

The first typing judgment is for typing linear assets. These type a purely linear process P using a purely linear context Δ_{P} and offering type A_P along channel x_P. The mode P of the channel indicates that a purely linear session is offered. The second and third typing judgment are for typing contracts. The second judgment shows the type of a contract

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

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_L : A_L) \vdash^{g} Q :: (z_C : C)}{\Psi \; ; \; \Gamma, (x_S : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_L) \; ; \; \Delta \vdash^{g} x_L \leftarrow \text{acquire } x_S \; ; \; Q :: (z_C : C)} \; \uparrow_{\mathsf{L}}^{\mathsf{S}} L$$

$$\frac{\Psi \; ; \; \cdot \; ; \; \Delta \vdash^{g} P :: (x_L : A_L)}{\Psi \; ; \; \cdot \; ; \; \Delta \vdash^{g} x_L \leftarrow \text{accept } x_S \; ; \; P :: (x_S : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_L)} \; \uparrow_{\mathsf{L}}^{\mathsf{S}} R$$

$$\frac{\Psi \; ; \; \Gamma, (x_S : A_S) \; ; \; \Delta \vdash^{g} Q :: (z_C : C)}{\Psi \; ; \; \Gamma \; ; \; \Delta, (x_L : \downarrow_{\mathsf{L}}^{\mathsf{S}} A_S) \vdash^{g} x_S \leftarrow \text{release } x_L \; ; \; Q :: (z_C : C)} \; \downarrow_{\mathsf{L}}^{\mathsf{S}} L$$

$$\frac{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{g} P :: (x_S : A_S)}{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^{g} x_S \leftarrow \text{detach } x_L \; ; \; P :: (x_L : \downarrow_{\mathsf{L}}^{\mathsf{S}} A_S)} \; \downarrow_{\mathsf{L}}^{\mathsf{S}} R$$

Figure 5. Typing rules corresponding to the shared layer.

process P using a purely linear channel context Δ_P and offering type A_S on channel x_S. Once this shared channel is acquired by a client, the shared process transitions to its linear phase, whose typing is governed by the third judgment. The context remains purely linear Δ_P, while the offered channel transitions to shared linear mode L. Finally, the fourth typing judgment types a client process. This is the only judgment with a shared channel context Γ, allowing only a client process to acquire a contract process.

This factorization and the fact that contracts, as the only shared processes, do not have a shared channel context Γ, upholds preservation while allowing shared contract processes to rely on linear resources. Figure 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

$$\text{proc}(a_S, w', x_L \leftarrow \text{accept } a_S \; ; \; P_{x_L}),$$
$$\text{proc}(c_C, w, x_L \leftarrow \text{acquire } a_S \; ; \; Q_{x_L}) \mapsto$$
$$\text{proc}(a_L, w', P_{a_L}), \quad \text{proc}(c_C, w, Q_{a_L})$$

This rule exploits the invariant that a contract process' providing channel a can come at two different modes, a linear one a_L, and a shared one a_S. The linear channel a_L is substituted for the channel variable x_L occurring in the process terms P and Q.

The dual to acquire-accept is *release-detach*. A client can *release* linear access to a contract process, while the contract process *detaches* from the client. The corresponding typing rules are presented in Figure 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.

$$\text{proc}(a_L, w', x_S \leftarrow \text{detach } a_L \; ; \; P_{x_S}),$$
$$\text{proc}(c_C, w, x_S \leftarrow \text{release } a_L \; ; \; Q_{x_S}) \longmapsto$$
$$\text{proc}(a_S, w', P_{a_S}), \quad \text{proc}(c_C, w, Q_{a_S})$$

6 Tracking Resource Usage

Resource usage is particularly important in digital contracts: Since multiple parties need to agree on the result of the execution of a contract, the computation is potentially performed multiple times or by a trusted third party. This immediately introduces the need to prevent denial of service attacks and to distribute the cost of the computation among the participating parties.

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

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

Functional Layer   Numerous techniques have been proposed to statically derive resource bounds for functional programs [9, 17, 19, 33, 41]. In Nomos, we adapt the work on automatic amortized resource analysis (AARA) [25, 27] that has been implemented in Resource Aware ML (RaML) [26]. 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 *findbid* in Figure 6. The argument b : bids can store the bids as a list of pairs of integers. The first component of a pair stores the identifier and the second component stores the bid. The function *findbid* searches the list for the identifier r and returns the

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

Figure 6. A linear-time implementation of the function *findbid* from the auction contract in Figure 1

corresponding bid value v and the list of bits without the pair (r, v) (to prevent bidders to withdraw twice).

We use *tick* annotations to define the resource usage of an expression in this article. We have annotated the code in Figure 6 to count the number of function calls. So the resource usage of an evaluation of *findbid b r* is $|b| + 1$.

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

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

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

In a larger program, we might want to call the function *findbid* again on the result of a call to the function. In this case, we would need to assign the type $L^1(\text{int} \times \text{int})$ to the resulting list and require $L^2(\text{int} \times \text{int})$ 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. [25, 26].

In the auction process from Figure 1, we would like to have a constant worst-case bound for finding a bit. We can achieve that by stopping the list traversal after checking 50 elements (capacity of the auction). We can then derive the following type expressing the constant worst-case bound.

$$\textit{findbid} : L^0(\text{int} \times \text{int}) \times \text{int} \xrightarrow{51/0} L^0(\text{int} \times \text{int}) \times \text{int option}$$

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 $\cdot \Vdash^g M : \tau$ and $M \Downarrow V \mid \mu$ then $q \geq \mu$.

More details about AARA can be found in the literature [26, 27] and the supplementary material.

Process Layer   To bound resource usage of a process, Nomos features recently introduced resource-aware session types [20] 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 [20].

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

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 [20], 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.

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

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

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

The reformulation is more compact since we need to account for potential in only the typing rules corresponding to $\triangleright^r A$ and $\triangleleft^r A$. Note that the distribution of the potential is a theoretical construct to prove resource bounds but do not introduce transmission of messages at runtime.

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 7 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

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

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

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

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

Figure 7. Selected typing rules corresponding to potential.

of potential. In the dual rule $\triangleleft L$, a process storing potential $q = p + r$ sends r units along the channel x_m in its context using the *get* construct, and the continuation remains with p units of potential. The typing rules for the dual constructor $\triangleleft^r A$ are the exact inverse. Finally, executing the tick (r) construct consumes r potential from the stored process potential q, and the continuation remains with $p = q - r$ units, as described in the tick rule in Figure 7.

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 3. 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 [26] describes that the base types in Ψ is copied to both Ψ_1 and Ψ_2, but the potential is split up. For instance, $L^{q_1+q_2}(\tau) \curlyvee (L^{q_1}(\tau), L^{q_2}(\tau))$. The rule $\supset L$ 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.

Operational Cost Semantics   The resource usage of a process (or message) is tracked in semantic objects $\text{proc}(c, w, P)$ and $\text{msg}(c, w, M)$ 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

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

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.

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

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.

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

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

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

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

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

where $\text{proc}(c, w, P)$ and $\text{msg}(c, w, M)$ are said to offer along channel c. A key question then is how to type these configurations. A configuration both uses and provides a collection of channels. The typing imposes a partial order among the processes and messages, requiring the provider of a channel to appear to the left of its client. 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 $\Sigma \ ; \ \Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma \ ; \ \Delta)$ defining a configuration Ω providing shared channels in Γ and linear channels in Δ. Additionally, we need to track the mapping between the shared channels and linear channels offered by a contract process, switching back and forth between them when the channel is acquired or released respectively. This mapping is stored in Γ_S. E is a natural number and stores the sum of the total potential and work as recorded in each process and message. We call E the energy of the configuration.

Finally, Σ denotes a signature storing the type and function definitions. A signature is well-formed if a) every type

definition $V = A_V$ is *contractive* [21] 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 $\Sigma \ ; \ \Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma \ ; \ \Delta)$. If the configuration takes a step, i.e. $\Omega \mapsto \Omega'$, then there exist Γ'_S, Γ' such that $\Sigma \ ; \ \Gamma'_S \overset{E}{\vDash} \Omega' :: (\Gamma' \ ; \ \Delta)$, i.e., the resulting configuration is well-typed. Additionally, $\Gamma_S \subseteq \Gamma'_S$ and $\Gamma \subseteq \Gamma'$.*

The preservation theorem is standard for expressions [26]. For processes, we proceed by induction on the operational cost semantics and inversion on the process typing judgment.

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

Theorem 2 (Progress). *Consider a closed well-formed and well-typed configuration Ω such that $\Gamma_S \overset{E}{\vDash} \Omega :: (\Gamma \ ; \ \Delta)$. Either Ω is poised, or it can take a step, i.e., $\Omega \mapsto \Omega'$, or some process in Ω is blocked along a_S for some shared channel a_S and there is a process $\text{proc}(a_L, w, P) \in \Omega$.*

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

8 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 $C_1, \ldots, C_m$ together with their type information $\Psi^i \ ; \ \cdot \ ; \ \Delta^i_P \vdash^{g_i} C_i :: (x^i_S : A^i_S)$. The functional context Ψ^i contains the data while the linear session-type context Δ^i_P contains the linear channels stored in the contract. 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.

One limitation that we seek to lift in the future is that contracts currently cannot directly interact with contracts. Such interactions can however be simulated by clients that act as a middle person. For reasons of elegance and readability, this solution is not ideal but somewhat justified since it greatly simplified the type theory of Nomos.

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

$$\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^q P :: (x_C : A_C)$$

Here, $\Gamma = x_S^1 : A_S^1, \dots, x_S^m : A_S^m$ 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, client process P, contracts $C_1, \dots, C_n$, 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 is also available to cover the gas cost. It is important to note that 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 client, thereby terminating the session. If the client received linear resources during the transaction, they must store it in their *wallet* process (which stores the client'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 client 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.

Another important point is that the implementation has to ensure that it is not possible to create linear coins out of thin air. To this end, the language cannot feature introduction forms for the type coin. Moreover, coins in the arguments of transactions have to be linked to actual cryptocoins that are submitted as part of the transaction. This is not completely trivial as coins can appear in data structures but it would not be hard to impose ad-hoc restriction to the data structures that are allowed to contain coins in transaction arguments.

Deadlocks   The only language specific reason a transaction can fail is a deadlock in the client code. Our progress theorem accounts for the possibility of deadlocks. Deadlocks may arise due to cyclic interdependencies on the contracts clients 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 will be readily transferable to our setting.

Surface Syntax and Client Code   In this paper, we did not focus on the usability of Nomos. However, we do not neglect this point and plan to work with the blockchain community to develop a more intuitive surface syntax. One point that we would like to make is that we do not expect users to write a new client process for every interaction she wants to have with a contract. We rather envision that a contract developer would create a contract together with several boilerplate clients that a user would then instantiate with the fitting arguments; for example the bid for an auction.

9   Preliminary Evaluation

To evaluate the practicality of our approach, we implemented several digital contracts in Concurrent C0 [10, 47], a type-safe C-like imperative language with session types. The implementation uses an asynchronous semantics, and supports functional variables, shared and linear channels. Concurrent C0 does not have the type-theoretic foundation of Nomos and, in particular, does not support potential annotations on types. However, the language is similar enough to Nomos to draw conclusions about the efficiency of its session-type based approach, especially since the implemented contracts are not imperative.

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

Contracts   In the evaluation 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.

$$\text{ballot} = \uparrow_L^S \lhd^{11} \oplus \{ \text{open} : \text{id} \supset \oplus \{ \text{vote} : \text{id} \supset \downarrow_L^S \text{ballot},$$
$$\text{novote} : \rhd^3 \downarrow_L^S \text{ballot} \},$$
$$\text{closed} : \text{id} \wedge \rhd^8 \downarrow_L^S \text{ballot} \}$$

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

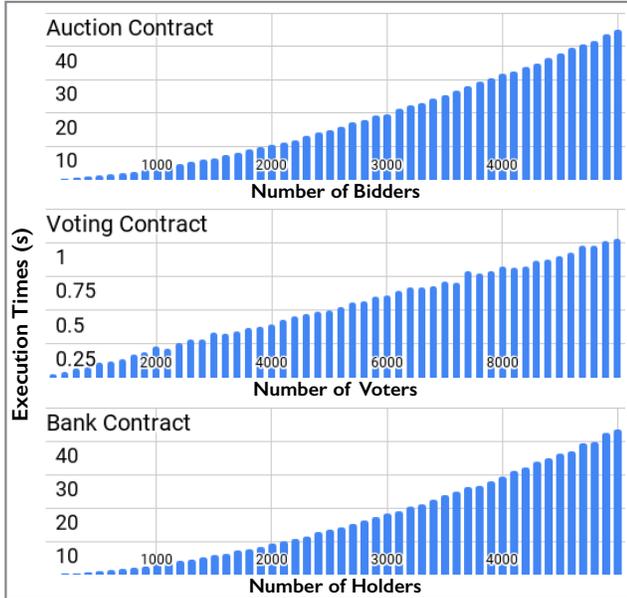The banking contract provides an account type, as follows.

Figure 8. Contract execution times

$$\uparrow^S_L \vartriangleleft^{13} \& \{signup : id \supset pwd \supset \vartriangleright^5 \downarrow^S_L account,$$
$$login : id \supset pwd \supset \oplus\{failure : \vartriangleright^7 \downarrow^S_L account,$$
$$success : \&\{deposit : money \multimap \downarrow^S_L account,$$
$$balance : int \wedge \vartriangleright^4 \downarrow^S_L account,$$
$$withdraw : int \supset money \otimes \downarrow^S_L account\}\}\}$$

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 8 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.
- 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.

10   Other Related Work

Existing smart contracts on Ethereum are predominantly implemented in Solidity [1], a statically typed object-oriented language influenced by Python and Javascript. Contracts in Solidity are similar to classes containing state variables and function declarations. However, the language provides no information about the resource usage of a contract. Languages like Vyper [6] address resource usage by disallowing recursion and infinite-length loops, thus making estimation of gas usage decidable. However, both languages still suffer from re-entrancy vulnerabilities. Bamboo [2], on the other hand, makes state transitions explicit and avoids re-entrance by design. However, none of these languages describe and enforce communication protocols statically.

Domain specific languages have also been designed for other blockchains apart from Ethereum. Rholang [4] is formally modeled by the ρ-calculus, a reflective higher-order extension of the π-calculus. Michelson [3] is a purely functional stack-based language that has no side effects. Liquidity [7] is a high-level language that complies with the security restrictions of Michelson. Scilla [43] is an intermediate-level language where contracts are structured as communicating automata providing a continuation-passing style computational model to the language semantics. In contrast to our work, none of these languages use linear type systems to track assets stored in a contract.

Session types have been integrated into a functional language in prior work [44]. However, this integration does not account for resource usage, nor sharing. Similarly, shared session types [10] have previously not been integrated with a functional layer or tracked for resource usage. Moreover, existing shared session types [10] disallow shared processes to rely on any linear resources, a restriction we lift in Nomos. Resource usage has previously been explored separately for a functional language [26] and the process layer [20], but the two have never been integrated together.

11   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 the 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.

In the future, we plan to design an efficient implementation of Nomos, with a focus on the practical aspects. Currently, Nomos doesn't allow direct contract-to-contract communication, and automatic inference of resource bounds. We plan to alleviate these limitations while maintaining type safety. There are also questions about integrating Nomos into a blockchain. These include the exact cost model and potential compilation to a lower-level language. We plan to answer these questions together with the blockchain community, and design further desirable features for programmers implementing smart contracts.

References

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

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

[3] 2018. The Michelson Language. https://www.michelson-lang.com/. Accessed: 2018-11-04.

[4] 2018. Rholang. https://github.com/rchain/Rholang. Accessed: 2018-11-04.

[5] 2018. The Rust Programming Language. https://doc.rust-lang.org/book.

[6] 2018. Vyper. https://vyper.readthedocs.io/en/latest/index.html. Accessed: 2018-11-04.

[7] 2018. Welcome to Liquidity's documentation! http://www.liquidity-lang.org/doc/index.html. Accessed: 2018-11-04.

[8] 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

[9] 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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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)*.

[16] 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).

[17] 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)*.

[18] 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.

[19] 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

[20] 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)*.

[21] 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

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

[23] 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.

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

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

[26] 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)*.

[27] 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)*.

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

[29] 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.

[30] 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.

[31] Blockchain Insurance Industry Initiative. 2008. B3i. (2008).

[32] 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)*.

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

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

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

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

[37] David Z. Morris. 2016. Blockchain-based Venture Capital Fund Hacked for $ 60 Million. http://fortune.com/2016/06/18/blockchain-vc-fund-hacked.

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

[39] 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.

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

[41] 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).

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

[43] 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

[44] 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.

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

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

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

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