Resource-Aware Session Types for Digital Contracts

Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, Ishani Santurkar
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

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

***Index Terms*—smart contracts, programming languages, session types, resource analysis**

I. Introduction

Digital contracts are programs that implement and enforce the execution of a contract. With the rise of blockchains and cryptocurrencies such as Bitcoin [1], Ethereum [2], and Tezos [3], digital contracts have become popular in the form of smart contracts, which provide potentially distrusting parties with programmable money and a distributed consensus mechanism. Smart contracts are used to implement auctions [4], investment instruments [5], insurance agreements [6], supply chain management [7], and mortgage loans [8]. They hold the promise to lower cost, increase fairness, and expand access to the financial infrastructure.

Many of today's prominent smart contract languages suffer from security vulnerabilities, which have severe financial consequences. A well-known example is the attack on The DAO [5], resulting in a \$60 million theft by exploiting a contract re-entrancy vulnerability. Smart contract languages have been typically derived from existing general-purpose languages [4], [9], [10] and fail to accommodate the domain-specific requirements of digital contracts. These requirements are: *(i)* expressing and enforcing protocols of interaction, *(ii)* controlling and computing resource (or gas) usage, and *(iii)* preventing duplication or deletion of a contract's assets.

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

To express and enforce the protocols underlying a contract, Nomos is based on *binary session types* [11]–[17]. Session types capture the protocols of interactions in the type, rather than the implementation code, and type-checking guarantees protocol adherence at run-time. Delimiting the sequences of actions that must be executed atomically, session types also prevent *re-entrance* into a contract in an inconsistent state. To control resource usage, Nomos employs *automatic amortized resource analysis (AARA)*, a type-based technique for automatically inferring symbolic resource bounds [18]–[22]. AARA is parametric in the cost model, allowing instantiation to track gas usage. As a result, Nomos contracts mitigate denial-of-service attacks without being vulnerable to out-of-gas exceptions. Moreover, resource bounds are integrated with session-typed protocols and enable precise path-sensitive descriptions of cost that avoid gaps between worst-case and average-case cost. To prevent duplication or deletion of assets, Nomos uses a *linear type system* [23]. The effectful session-typed language, which implements contract interfaces and contract-to-contract communication, is integrated with a strict, general-purpose functional language using a contextual monad.

Integrating these seemingly disparate approaches (session types, resource analysis, linearity, and functional programming) and combining them with the different roles that arise in a digital contract (contract, asset, transaction) in a way that the result remains consistent, presents unique challenges. For one, both the functional as well as session-typed language use potential annotations to bound the resource consumption, which requires care when functional values are exchanged as messages between processes. For another, prior work on integrating shared and linear session types [24] preclude contracts from persisting their linear assets across transactions, a feature essential to digital contract development; a restriction that we lift in this work. Fundamental is the use of different forms of *typing judgments* for expressions and processes along with *judgmental modes* to distinguish the different roles in a digital contract. The modes are essential in ensuring type safety, as they allow the expression of mode-indexed invariants on the

This article is based on research supported by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514 and Grant No. 1718276. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

typing contexts and their enforcement by the typing rules.

Nomos is completely formalized, including the type system, a cost semantics, and a transactional semantics to instantiate Nomos contracts on a blockchain. A type soundness proof ensures that protocols are followed at run-time and that types establish sound upper bounds on the resource consumption.

The soundness guarantees are meaningful in a *restricted attacker model* in which even the adversary cannot execute ill-typed code. Such a model is justified in a decentralized consensus setting such as a blockchain, where transactions and contracts are publicly type checked, thwarting attacks from adversaries intending to damage the blockchain state by submitting malformed code. In this setting, transaction validation and thus type checking are part of the attack surface and can be used by an adversary for denial-of-service attacks. To mitigate such attacks, we have carefully designed the Nomos type system that integrates its various features in a way that type checking is *linear-time* in the size of the program.

To evaluate Nomos, we implemented a publicly available open-source prototype [25] and conducted 8 case studies implementing common smart contracts such as auctions, elections, and currencies. Our experiments show that type-checking overhead is less than 0.7 ms for each contract and bound inference (can be performed off-chain) takes less than 10 ms. Moreover, gas bounds are tight for most contracts. To the best of our knowledge, this is the first implementation to integrate shared binary session types into a functional language with support for resource analysis.

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

Our main technical contributions are:

- design of Nomos, a language that addresses the domain-specific requirements of digital contracts by construction;
- a fine-tuned system of typing judgments (Section IV) that uses *modes* to orchestrate the sound integration of session types (Section III), functions (Section V), and resource analysis (Section VI);
- extension of shared session types to store linear assets;
- resource cost amortization by allowing gas storage in internal data structures (Section VI);
- type safety proof of Nomos using a novel asynchronous cost semantics (Section VII);
- a prototype implementation and case study of prominent blockchain applications (Section VIII);
- a transactional semantics to deploy and execute Nomos contracts and transactions on a blockchain (Section IX).

In addition, the technical report [26] details the technical development, provides additional explanations and the full implementation of the blockchain applications.

II. Nomos by Example

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

Explicit Protocols of Interaction: Digital contracts, like traditional contracts, follow a *predefined protocol*. For instance, an auction contract distinguishes a bidding phase, where bidders submit their bids, possibly multiple times, from a subsequent collection phase, where the highest bidder receives the lot while all other bidders receive their bids back. In Solidity [4], the bidding phase of an auction is typically implemented as the bid function below. This function receives a bid (msg.value) from a bidder (msg.sender) and adds it to the bidder's total previous bids (bidValue).

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

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

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

$$\begin{aligned}
&\textbf{stype}\ \mathsf{auction} = \\
&\uparrow_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{22} \oplus\{\textbf{running} : \&\{\textbf{bid} : \mathsf{id} \rightarrow \mathsf{money} \multimap \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}, \\
&\qquad\qquad\qquad\qquad \textbf{cancel} : \triangleright^{21} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}, \\
&\textbf{ended} : \&\{\textbf{collect} : \mathsf{id} \rightarrow \oplus\{\textbf{won} : \mathsf{lot} \otimes \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}, \\
&\qquad\qquad\qquad\qquad\qquad \textbf{lost} : \mathsf{money} \otimes \triangleright^{7} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}, \\
&\qquad\qquad \textbf{cancel} : \triangleright^{21} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}\}
\end{aligned}$$

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

Nomos uses *shared* session types [24] to guarantee that bidders interact with the auction in mutual exclusion from each other and that the sequences of actions are executed *atomically*. To demarcate the parts of the protocol that become a *critical section*, the above session type uses the $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ and $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ modalities. The $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ modality denotes the beginning of a critical section, the $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ modality denotes its end. Programmatically, $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ translates into an *acquire* of the auction session and $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ into its *release*, which is only sound if the protocol behaves like an auction afterwards (*equi-synchronizing* type).

Contracts are implemented by *processes*, revealing the concurrent, message-passing nature of session-typed languages. The process *run* below implements the auction's running phase. Line 3 gives the process' signature, indicating that it offers a shared session of type auction along the channel sa and uses a linear hash map b : hashmap$\langle$id, bid$\rangle$ of bids indexed by id and a linear lot l. Line 5 onward list the process body. Lines 1,2 define session types bid and bids, respectively.

```
1:  stype bid = $\&\{\mathbf{addr} : \mathsf{id} \wedge \mathsf{bid}, \mathbf{val} : \mathsf{money}\}$
2:  stype bids = hashmap$\langle \mathsf{id}, \mathsf{bid} \rangle$
3:  $(b : \mathsf{bids}), (l : \mathsf{lot}) \vdash run :: (sa : \mathsf{auction})$
4:    $sa \leftarrow run \leftarrow b\ l =$
5:      $la \leftarrow$ accept $sa$ ;
6:      $la.\mathbf{running}$ ;
7:      case $la$ ( $\mathbf{bid} \Rightarrow r \leftarrow$ recv $la$ ;
8:                        $m \leftarrow$ recv $la$ ;
9:                        $sa \leftarrow$ detach $la$ ;
10:                       $b' \leftarrow addbid\ r \leftarrow b\ m$ ;
11:                       $sa \leftarrow check \leftarrow b'\ l$
12:             | $\mathbf{cancel} \Rightarrow sa \leftarrow$ detach $la$ ;
13:                         $sa \leftarrow run \leftarrow b\ l$)
```

The contract process first *accepts* an acquire request by a bidder (line 5) and then sends the message **running** (line 6), indicating the auction status and waiting for the bidder's choice. Should the bidder choose to make a bid, the process waits to receive the bidder's identifier (line 7) followed by money equivalent to the bidder's bid (line 8). Internally, the process stores the pair of the bidder's identifier and bid in the data structure bids (line 10). After this linear exchange, the process leaves the critical section by issuing a *detach* (line 9), matching the bidder's release request, and tail calls the *check* process (line 11) that compares the number of bidders with a threshold. If the threshold is exceeded, the contract transitions to the **ended** phase implemented by a different process, otherwise the *run* process is called again.

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

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

```
1 function collect() public payable {
2   require (status == ended);
3   bidder = msg.sender;
4   bid = bidValue[bidder];
5   bidder.send(bid);
6   bidValue[bidder] = 0; }
7
8 function () payable {
9   auction.collect(); }
```

A bidder can now cause re-entrancy by creating a dummy contract with an unnamed *fallback* function (line 9) that calls the auction's collect function. This call is triggered when collect calls send (line 5), leading to an infinite recursive call to collect, depleting all funds from the auction. The message-passing framework of session types eliminates this vulnerability. While session types provide multiple clients access to a contract, the acquire-release discipline ensures that clients interact with the contract in mutual exclusion. To attempt re-entrancy, a bidder will need to acquire the auction contract twice without releasing it, but the second acquire would fail to execute.

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

Nomos uses resource-aware session types [27] to statically analyze the resource cost of a transaction. They operate by assigning an initial *potential* to each process. This potential is consumed by each operation that the process executes or can be transferred between processes to share and amortize cost. The cost of each operation is defined by a cost model. If the cost model assigns a cost to each operation equivalent to their gas cost during execution, the potential consumed during a transaction reflects an upper bound on the gas usage.

Resource-aware session types express the potential as part of the session type using the operators $\triangleleft$ and $\triangleright$. The $\triangleleft$ operator prescribes that the client must send potential to the contract, with the amount of potential indicated as a superscript. Dually, $\triangleright$ prescribes that the contract must send potential to the client. In case of the auction contract, we require the client to pay potential for the operations that the contract must execute, both while placing and collecting their bids. If the cost model assigns a cost of 1 to each contract operation, then the maximum cost of an auction session is 22 (taking the max number of operations in all branches). Thus, we require the client to send 22 units of potential at the start of the auction session type using $\triangleleft^{22}$. In the **lost** branch of the auction type, on the other hand, the contract returns 7 units of potential to the client using $\triangleright^{7}$. This mirrors gas usage in smart contracts, where the sender initiates a transaction with some initial gas, and the leftover gas at the end of the transaction is returned to the sender. In contrast to existing smart contract languages like Solidity, which provide no support for analyzing the cost of a transaction, Nomos' type checker has automatically inferred these potential annotations and guarantees that well-typed transactions cannot run out of gas. Thus, Nomos enforces static gas bounds on transactions without burdening the programmer to infer them.

Bringing It All Together: Combining all these features soundly in one language is challenging. In Nomos, we achieve this by using different *typing judgments* and *channel modes*, identifying the role of the process offered along that channel. The mode R denotes *purely linear processes* for linear assets or private data structures, such as b and l in the auction. The modes S and L denote *sharable processes*, i.e., contracts, that are either in their shared or linear phase such as sa and la, respectively. The mode T denotes a *transaction process* that can refer to shared and linear processes and is issued by a user, such as bidder in the auction. Modes are assigned to each channel and are carried over into the process typing judgments imposing invariants (Definition 1) that are key to type safety. To simplify programming, Nomos' inference engine automatically infers the channel modes, thus relieving the programmer from the burden of annotating each channel with its respective mode.

III. Base System of Session Types

Nomos builds on linear session types for message-passing concurrency [11]–[14], [17] 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 [23] and the session-typed π-calculus [14], [17]. Linear propositions can be viewed as resources that must be used *exactly once* in a proof. Under the Curry-Howard correspondence, an intuitionistic linear sequent $A_1, A_2, \ldots, A_n \vdash C$ can be interpreted as the offer of a session C by a process P using the sessions $A_1, A_2, \ldots, A_n$

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

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

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

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

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

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

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

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.

Following previous work on session types [15], [16], the process expressions of Nomos are defined as follows.

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

Because we adopt the intuitionistic version of linear logic, session types are expressed from the point of view of the provider. Table I 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. Columns 2 and 4 describe the type and term after the interaction. The last column describes the provider and client action. Figure 1 provides selected typing rules. As an illustration of the statics and semantics, we explain the internal choice ($\oplus$) connective.

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

The operational semantics is formalized as a system of *multiset rewriting rules* [28]. We introduce semantic objects $\mathsf{proc}(c_m, w, P)$ and $\mathsf{msg}(c_m, w, N)$ denoting process P and message N, respectively, being provided along channel c at mode m. The resource annotation w indicates the work performed so far, the discussion of which we defer to Section VI. Communication is *asynchronous*, allowing the sender $(c_m.l \; ; \; P)$ to continue with P without waiting for l to be received. As a technical device to ensure that consecutive messages arrive in the order they were sent, the sender also creates a fresh continuation channel c_m^+ so that the message

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

TABLE I: Overview of binary session types with their operational description

$\boxed{\Psi \; ; \; \Gamma \; ; \; \Delta \vdash^q P :: (x_m : A)}$ Process P uses linear channels in Δ, and provides type A along x.

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

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

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

Fig. 1: Selected typing rules for process communication

l is actually represented as $(c_m.l \; ; \; c_m \leftarrow c_m^+)$ (read: send l along c_m and continue as c_m^+):

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

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

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

The message $\mathsf{msg}(c_m, w, c_m.l \; ; \; c_m \leftarrow c_m^+)$ is just a particular form of process. Therefore, no separate typing rules for messages are needed; they can be typed as processes [24].

Channel Passing.: Nomos allows the exchange of channels over channels, also referred to as higher-order channels. A process providing $A \multimap_n B$ can receive a channel of type A at mode n and then continue with providing B. The provider process term is $(y_n \leftarrow \mathsf{recv}\; x_m \; ; \; P)$, where P is the continuation. The corresponding client sends this channel using $(\mathsf{send}\; x_m\; w_n \; ; \; Q)$. The dual type operator $A \otimes_n B$ requires the provider to send a channel of type A at mode n and then continue with providing B. The client receives this channel and continues to use B. An important distinction from standard session types is that the $\multimap$ and $\otimes$ types are decorated with the mode m of the channel exchanged. Since modes distinguish the status of the channels in Nomos, this mode decoration is necessary to ensure type safety.

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

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

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

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

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

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

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

$$\mathsf{proc}(c_k, w, [a_m/x_m]Q)$$

where a_m is a fresh channel. Here we write $[\overline{d}/\Psi]$ and $[\overline{e}/\Gamma\ \Delta]$ to denote substitution of the variables in $\overline{d}$ and $\overline{e}$ for the corresponding variables in Ψ and $\Gamma\ ;\ \Delta$ respectively in that order. Sometimes a process invocation is a *tail call*, written without a continuation as $x_m \leftarrow f\ \overline{y_1} \leftarrow \overline{y_2}$. This is a shorthand for $x'_m \leftarrow f\ \overline{y_1} \leftarrow \overline{y_2}\ ;\ x_m \leftarrow x'_m$ for a fresh variable x'_m, that is, we create a fresh channel and immediately identify it with x_m (although it is implemented more efficiently).

Session types can be naturally extended to include recursive types. For this purpose we allow (possibly mutually recursive) type definitions $X = A$ in the signature, where we require A to be *contractive* [29]. This means here that A should not itself be a type name. Our type definitions are *equi-recursive* so we can silently replace X by A during type checking, and no explicit rules for recursive types are needed.

IV. Sharing Contracts

Multi-user support is fundamental to digital contract development. Linear session types, as defined in Section III, 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 [24]. 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* [16], [30], [31], session types are stratified into a linear and shared layer with two *adjoint modalities* going back and forth between them:

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

The ${\uparrow}_{\mathsf{L}}^{\mathsf{S}}$ type modality translates into an *acquire*, while the dual ${\downarrow}_{\mathsf{L}}^{\mathsf{S}}$ type modality into a *release*. Whereas mutual exclusion is one key ingredient to guarantee 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 the key behind eliminating *re-entrancy attacks* since it prevents a user from interrupting an ongoing session in the middle and initiating a new one. In the Nomos typing judgment $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x_m : A)$, the contexts Γ and Δ store the shared and linear channels that P can refer to, respectively. The stratification of channels into layers arises from a difference in structural properties that exist for types at a mode. Shared propositions exhibit weakening, contraction and exchange, thus can be discarded or duplicated, while linear propositions only exhibit exchange.

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

Fig. 2: Grammar for shared session types

Allowing Contracts to Rely on Linear Assets: As exemplified by the auction contract, a digital contract typically amounts to a process that is shared at the outset, but oscillates between shared and linear to interact with clients, one at a time. Crucial for this pattern is the ability of a contract to maintain its linear assets (e.g., money or lot for the auction) regardless of its mode. Unfortunately, current shared session types [24] do not allow a shared process to rely on any linear channels, requiring any linear assets to be consumed before becoming shared. This precaution was logically motivated [32] and also crucial for type preservation.

A key novelty of our work is to lift this restriction while *maintaining type preservation*. To this end, we factorize the process typing judgment according to the *three roles* that arise in digital contract programs: *contracts*, *transactions*, and *linear assets*. Since contracts oscillate between shared and linear modes (due to acquire/release), we get 4 sub-judgments for typing processes, each characterized by the mode of the channel being offered.

Definition 1 (Process Typing)**.** *The judgment $\Psi\ ;\ \Gamma\ ;\ \Delta \vdash^{q} P :: (x_m : A)$ is categorized according to mode m imposing certain invariants on the judgment. $\mathbf{L}(A)$ denotes the language generated by the grammar of A.*

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

Figure 2 shows the session type grammar in Nomos. The first sub-judgment in Definition 1 is for typing linear assets. These type a purely linear process P using a purely linear context Δ (channels at mode R and types belonging to grammar A_{R} in Figure 2) and offering a purely linear type A along channel x_{R}. The mode R of the channel indicates that a purely linear session is offered. The second and third sub-judgments are for typing contracts. The second sub-judgment shows the type of a contract process P using a shared context Γ and a purely linear channel context Δ and offering shared type A on the shared channel x_{S}. Once this shared channel is acquired by a user, the shared process transitions to its linear phase, whose

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

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

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

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

typing is governed by the third sub-judgment. The offered channel transitions to linear mode L, while the linear context may now contain channels at modes R or L. Finally, the fourth typing judgment types a linear process, corresponding to a *transaction* holding access to shared channels Γ and linear channels Δ, and offering at mode T.

This novel factorization upholds preservation while allowing shared contract processes to rely on linear resources. The modes impose the ordering $\mathsf{R} < \mathsf{S} < \mathsf{L} < \mathsf{T}$ among the channels in the configuration. A process (offering a channel) at a certain mode is allowed to rely only on processes at the same or lower mode. These are exactly the conditions imposed by Definition 1. This introduces an implicit ordering among the linear processes depending on their mode, ensuring that no cyclic dependencies can arise among processes and imposing a tree structure on the process configuration. Relatedly, shared processes can only refer to shared channels (at mode S) or purely linear channels (at mode R) as exemplified by the judgment Δ purelin in Figure 3. Formally, Δ purelin denotes that for all $d_k \in \Delta \implies k = \mathsf{R}$. This ensures that a shared contract must release all processes it has acquired before itself being released. This further enforces an ordering in which the channels are acquired and released, thus *allowing contracts to interact with other contracts without compromising type safety*.

Shared session types introduce new typing rules into our system, concerning the *acquire-release* constructs (see Figure 3). 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}. To accept an acquire request, the contract must only contain channels at mode R (indicated by Δ purelin), in accordance with Definition 1. This premise is crucial to type preservation, since it ensures that a contract has not acquired another contract while it is accepting an acquire request itself. Implicitly, this imposes an order on the acquire of contracts, and the inverse order is followed for their release. 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.

V. Adding a Functional Layer

To support general-purpose programming patterns, Nomos combines linear channels with conventional data structures, such as integers, lists, or dictionaries. To reflect and track different classes of data in the type system, we take inspiration from prior work [15], [16] and incorporate processes into a functional core via a *linear contextual monad* that isolates session-based concurrency. To this end, we introduce a separate functional context to the typing of a process. The linear contextual monad encapsulates open concurrent computations, which can be passed in functional computations but also transferred between processes in the form of *higher-order processes*, providing a uniform integration of higher-order functions and processes.

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

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

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

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

Contextual Monad: The main novelty in the functional types are the three type formers for contextual monads, denoting the type of a process expression. The type $\{A_{\mathsf{R}} \leftarrow \overline{A_{\mathsf{R}}}\}_{\mathsf{R}}$ denotes a process offering a *purely linear* session type A_{R} and using the purely linear vector of types $\overline{A_{\mathsf{R}}}$. The corresponding introduction form in the functional language is the monadic value constructor $\{c_{\mathsf{R}} \leftarrow P \leftarrow \overline{d_{\mathsf{R}}}\}$, denoting a runnable process offering along channel c_{R} that uses channels $\overline{d_{\mathsf{R}}}$, all at mode R. The corresponding typing rule for the monad is $\{\}I_{\mathsf{R}}$ in Figure 4 (ignore the blue portions).

The monadic *bind* operation implements process composition and acts as the elimination form for values of type $\{A_{\mathsf{R}} \leftarrow \overline{A_{\mathsf{R}}}\}_{\mathsf{R}}$. The bind operation, written as $c_{\mathsf{R}} \leftarrow M \leftarrow \overline{d_{\mathsf{R}}} \, ; \, Q_c$, composes the process underlying the monadic term M, which offers along channel c_{R} and uses channels $\overline{d_{\mathsf{R}}}$, with Q_c, which uses c_{R}. The typing rule for the monadic bind is rule $\{\}E_{\mathsf{RR}}$ in Figure 4. The context Δ 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 VI. The shared context Γ is empty in accordance with the invariants established in Definition 1 *(i)*, since the mode of offered channel z is R. The effect of executing a bind is the spawn of the purely linear process corresponding to the monad M, and the parent process continuing with Q.

$$\begin{aligned} (\uparrow^{\mathsf{S}}_{\mathsf{L}} C) : {} & \mathsf{proc}(d_{\mathsf{R}}, w, x_{\mathsf{R}} \leftarrow \{x'_{\mathsf{R}} \leftarrow P_{x'_{\mathsf{R}}, \overline{y}} \leftarrow \overline{y}\} \leftarrow \overline{a} \, ; \, Q) \mapsto \\ & \mathsf{proc}(c_{\mathsf{R}}, 0, P_{c_{\mathsf{R}}, \overline{a}}), \ \mathsf{proc}(d_{\mathsf{R}}, w, [c_{\mathsf{R}}/x_{\mathsf{R}}]Q) \end{aligned}$$

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

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

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

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

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

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

The above rule spawns the process P offering along a globally fresh channel c_{R}, and using channels $\overline{a}$. The continuation process Q acts as a client for this fresh channel c_{R}. The other two monadic types correspond to spawning a shared process $\{A_{\mathsf{S}} \leftarrow \overline{A_{\mathsf{S}}} \; ; \; \overline{A_{\mathsf{R}}}\}_{\mathsf{S}}$ and a transaction process $\{A_{\mathsf{T}} \leftarrow \overline{A_{\mathsf{S}}} \; ; \; \overline{A}\}_{\mathsf{T}}$ at mode S and T, respectively. Their rules are analogous to $\{\}I_{\mathsf{R}}$ and $\{\}E_{\mathsf{RR}}$ and described in the technical report [26].

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

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

The type $\tau \rightarrow A$ 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 ($\rightarrow R, \rightarrow L$) are given in Figure 4 (rules for $\wedge$ are inverse). As indicated in the $\rightarrow R$ rule, receiving a value $y : \tau$ on a channel $x : \tau \rightarrow A$ adds it to the functional context Ψ. On the other hand, sending M on channel $x : \tau \rightarrow A$ requires that M has type τ (third premise).

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

$$\begin{array}{l} \textbf{stype}\ \mathsf{money} = \\ \&\{\textbf{value} : \mathsf{int} \wedge \mathsf{money}, \\ \quad \textbf{add} : \mathsf{money} \multimap_{\mathsf{R}} \mathsf{money}, \\ \quad \textbf{subtract} : \mathsf{int} \rightarrow \oplus\{\textbf{sufficient} : \mathsf{money} \otimes_{\mathsf{R}} \mathsf{money}, \\ \qquad\qquad\qquad\qquad\quad \textbf{insufficient} : \mathsf{money}\} \\ \quad \textbf{coins} : \mathsf{list}_{\mathsf{coin}}\} \end{array}$$

The type supports querying for value, and addition and subtraction. The type also expresses insufficiency of funds in the case of subtraction. The provider process only supplies money to the client if the requested amount is less than the current balance, as depicted in the **sufficient** label. The type is implemented by a *wallet* process that internally stores a linear list of coins and an integer representing its value. The technical report [26] contains its code and explanation.

VI. Tracking Resource Usage

The predominant approach for tracking resource cost on blockchains like Ethereum is to introduce a cost model that defines the *gas* consumption of low level operations. A transaction needs to be executed and validated before adding it to the global distributed ledger, i.e., blockchain. This validation is performed by *miners*, who charge fees based on the gas consumption of the transaction. This fee has to be estimated and provided by the sender prior to the transaction.

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. Our technique is parametric in the cost model applied by the programmer, thus making it directly applicable for gas cost analysis. The programmer only needs to specify the gas cost of each primitive operation, and our type system infers the corresponding gas bound of a transaction.

Functional Layer: Numerous techniques have been proposed to statically derive resource bounds for functional programs [33]–[37]. In Nomos, we adapt the work on automatic amortized resource analysis (AARA) [18], [20] that has been implemented in Resource Aware ML (RaML) [21]. 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.

As an illustration, consider the function *apply* that iterates over a list of balances and applies interest on each element, multiplying them by a constant c. We use *tick* annotations to define the resource usage of an expression in this article. One *tick* operation realizes a cost of 1. We have annotated the code to count the number of multiplications. The resource usage of an evaluation of *apply* b is $\mathsf{len}(b)$.

```
let rec apply balances =
  match balances with
  | [] -> []
  | hd::tl -> tick(1); (c*hd)::(apply tl)
```

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.

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

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. These annotations need not be provided by the programmer and can be inferred automatically by an off-the-shelf LP solver, even if the potential functions are polynomial [20], [21].

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 $M \Downarrow V \mid \mu$ which states that the closed expression M evaluates to the value V with cost μ. More details about AARA can be found in the literature [18], [21] and the technical report [26].

Process Layer: To bound the resource usage of a process, Nomos features resource-aware session types [27] 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. As an illustration, consider the following resource-aware list interface from prior work [27].

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

The type prescribes that the provider of list_A 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 [27], 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

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

With all aspects introduced, the process typing judgment

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

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

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

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

Figure 5 shows the rules that interact with the potential annotations. In the rule $\triangleleft R$, process P storing potential q receives r units along the offered channel $x_m : \triangleleft^{r} A$ using the *get* construct and the continuation executes with $p = q{+}r$ units of potential. In the dual rule $\triangleleft L$, a process storing potential $q = p{+}r$ sends r units along the channel $x_m : \triangleleft^{r} A$ in Δ using the *pay* construct, and the continuation remains with p units of potential. The typing rules for the dual constructor $\triangleright^{r} A$ are the exact inverse. Finally, executing the tick (r) construct consumes r potential from the stored process potential q, and the continuation remains with $p = q - r$ units, as described in the tick rule.

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

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

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

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

Fig. 5: Selected typing rules corresponding to potential.

The tick construct is used to simulate a cost model in Nomos. If an operation (e.g., sending a message, calling a function, etc.) has a cost of r, this cost is simulated by inserting tick (r) just before the operation. Then, the tick operations are the only ones that cost potential, thus simplifying the type system. These tick operations are automatically inserted by the Nomos type checker, using a predefined cost model that assigns a constant cost to each operation. The programmer is not allowed to insert their own tick operations, and cannot maliciously change the gas cost. In addition, our implementation provides some standard cost models that, for instance, assign a unit cost to each operation.

Integration: Since both AARA for functional programs and resource-aware session types are based on the integration of the potential method into their type systems, their combination is natural. The two points of integration of the functional and process layer are (i) spawning a process, and (ii) sending/receiving a value from the functional layer. Recall the spawn rule $\{\}E_{\mathsf{RR}}$ from Figure 4. A process storing potential $r = p + q$ can spawn a process corresponding to the monadic value M, if M needs p units of potential to evaluate, while the continuation needs q units of potential to execute. Moreover, the functional context Ψ is shared in the two premises as Ψ_1 and Ψ_2 using the judgment $\Psi \curlyvee (\Psi_1, \Psi_2)$. This judgment, explored in prior work [21] describes that the base types in Ψ are copied to both Ψ_1 and Ψ_2, but the potential is split up. For instance, $L^{q_1+q_2}(\tau) \curlyvee (L^{q_1}(\tau), L^{q_2}(\tau))$. The rule $\rightarrow L$ follows a similar pattern. Thus, the combination of the two type systems is smooth, assigning a uniform meaning to potential, both for the functional and process layer. Remarkably, this technical device of exchanging functional values can be used to exchange non-constant potential with messages.

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

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

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

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

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

VII. Type Soundness

The main theorems that exhibit the connections between our type system and the operational cost semantics are the usual *type preservation* and *progress*. First, Definition 1 asserts certain invariants on process typing judgment depending on the mode of the channel offered by a process. This mode, remains invariant, as the process evolves. This is ensured by the process typing rules, which remarkably preserve these invariants despite being parametric in the mode.

Lemma 1 (Invariants)**.** *The typing rules on the judgment $\Psi\ ;\ \Gamma\ ;\ \Delta \overset{q}{\vdash} (x_m : A)$ preserve the invariants outlined in Definition 1, i.e., if the conclusion satisfies the invariant, so do all the premises.*

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

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

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

The typing judgment for configurations has the form $\Sigma\ ;\ \Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta)$ defining a configuration Ω providing shared channels in Γ and linear channels in Δ. Additionally, we need to track the mapping between the shared channels and their linear counterparts offered by a contract process, switching back and forth between them when the channel is acquired or released respectively. This mapping, along with the type of the shared channels, is stored in Γ_0. E is a natural number and stores the sum of the total potential and work as recorded in each process and message. We call E the energy of the configuration. The technical report [26] details the configuration typing rules.

Finally, Σ denotes a signature storing the type and function definitions. A signature is well-formed if *(i)* every type definition $V = A_V$ is *contractive* [29] (A_V cannot be a type name) allowing an *equi-recursive* treatment [38] and *(ii)* every function definition $f = M : \tau$ is well-typed according to the expression typing judgment $\Sigma\ ;\ \cdot \overset{p}{\Vdash} M : \tau$. The signature does not contain process definitions; every process is encapsulated inside a function using the contextual monad.

Theorem 1 (Type Preservation)**.**

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

The preservation theorem is standard for expressions [21]. For processes, we proceed by induction on the operational cost semantics and inversion on the configuration and process typing judgment.

To state progress, we need the notion of a *poised* process [16]. A process $\mathsf{proc}(c_m, w, P)$ is poised if it is trying to receive a message on c_m. Dually, a message $\mathsf{msg}(c_m, w, N)$ is poised if it is sending along c_m. A configuration is poised if every message or process in the configuration is poised. Intuitively, this means that the configuration is trying to interact with the outside world along a channel in Γ or Δ. Additionally, a process can be *blocked* [24] if it is trying to acquire a contract process that has already been acquired by some process. This can lead to the possibility of deadlocks.

Theorem 2 (Progress)**.** *Consider a closed well-formed and well-typed configuration Ω such that $\Gamma_0 \overset{E}{\vDash} \Omega :: (\Gamma\ ;\ \Delta)$. Either Ω is poised, or it can take a step, i.e., $\Omega \mapsto \Omega'$, or some process in Ω is blocked along a_{S} for some shared channel a_{S} and there is a process $\mathsf{proc}(a_{\mathsf{L}}, w, P) \in \Omega$.*

The progress theorem is weaker than that for binary linear session types, where progress guarantees deadlock freedom.

VIII. Implementation and Evaluation

We have developed an open-source prototype implementation [25] of Nomos in OCaml. This prototype contains a lexer and parser (929 lines of code), a type checker (2388 lines of code), a pretty printer (451 lines of code), an LP solver interface (915 lines of code) and an interpreter (1286 lines of code) for implementing, type checking and executing Nomos programs. We also describe our efforts to simplify programming and improve accessiblity of Nomos to developers.

Syntax: The lexer and parser for Nomos have been implemented in Menhir [39], an LR(1) parser generator for OCaml. A Nomos program is a list of mutually recursive type and process definitions. To visually separate out functional variables from session-typed channels, we require that shared channels are prefixed by #, while linear channels are prefixed by \$. This avoids confusion between the two, both for the programmer and the parser. We also require the programmer to indicate the *mode* of the process being defined: *asset*, *contract* or *transaction*, assigning the respective modes R, S and T to the offered channel. The modes for all other channels are inferred automatically (explained later). The initial potential $\{q\}$ of a process is marked on the turnstile in the declaration. The syntax for definitions is

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

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

Type Checking: We implemented a bi-directional [40] type checker with a specific focus on the quality of error messages, which include, for example, *extent* (source code location) information for each definition and expression. The programmer provides the initial type of each variable and channel in the declaration and the definition is checked against it, while reconstructing the intermediate types. This helps localize the source of a type error as the point where type reconstruction fails. Type equality is restricted to reflexivity (constant time), although we have also implemented the standard co-inductive algorithm [29] which is quadratic in the size of type definitions. For all our examples, the reflexive notion of equality was sufficient. *Type checking is linear time in the size of the program*, which is important in the blockchain domain where type checking can be part of the attack surface.

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

Using ideas from existing techniques for type inference for AARA [18], [21], we reduce the reconstruction of potential annotations to linear optimization. To this end, Nomos' inference engine uses the Coin-Or LP solver. In a Nomos program, the programmer can indicate unknown potential using $*$. Thus, resource-aware session types can be marked with $\triangleright^*$ and $\triangleleft^*$, list types can be marked as $L^*(\tau)$ and process definitions can be marked with $|\{*\}-$ on the turnstile. The mode of all the channels is marked as 'unknown' while parsing.

The inference engine iterates over the program and substitutes the star annotations with potential variables and 'unknown' with mode variables. Then, the bidirectional typing rules are applied, approximately checking the program (modulo potential and mode annotations) while also generating linear constraints for potential annotations (see Figure 4). and mode annotations (see Definition 1 and Figure 3). Finally, these constraints are shipped to the LP solver, which minimizes the value of the potential annotations to achieve tight bounds. The LP solver either returns that the constraints are infeasible, or returns a satisfying assignment, which is then substituted into the program. The final program is pretty printed for the programmer to view and verify the potential and mode annotations.

A. Case Studies

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

ERC-20 Token Standard: ERC-20 [41] is a technical standard for smart contracts on the Ethereum blockchain that defines a common list of standard functions that a token contract has to implement. The majority of tokens on the Ethereum blockchain are ERC-20 compliant.

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

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

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

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

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

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

Hacker Gold (HKG) Token: The HKG token is one particular implementation of the ERC-20 token specification. Recently, a vulnerability was discovered in the HKG token smart contract based on a typographical error leading to a reissuance of the entire token [42]. When updating the receiver's balance during a transfer, instead of writing balance+=value, the programmer mistakenly wrote balance=+value (semantically meaning balance=value). Nomos' type system would have caught the linearity violation in the latter statement that drops the existing balance in the recipient's account.

Puzzle Contract: This contract, taken from prior work [43] rewards users who solve a computational puzzle and submit the solution. The contract allows two functions, one that allows the owner to update the reward, and the other that allows a user to submit their solution and collect the reward.

In Nomos, this contract is implemented to offer the type

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

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

Voting: The voting contract provides a ballot type.

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

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

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

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

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

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

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

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

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

Table II contains a compilation of our experiments with the case studies and the prototype implementation. The experiments were run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory. It presents the contract name, its lines of code (LOC), the type checking time (T (ms)), number of potential and mode variables introduced (Vars), number of potential and mode constraints that were generated while type checking (Cons) and the time the LP solver took

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

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

to infer their values (I (ms)). The last column describes the maximal gap between the static gas bound inferred and the actual runtime gas cost. It accounts for the difference in the gas cost in different program paths. However, this waste is clearly marked in the program by explicit *tick* instructions so the programmer is aware of this runtime gap, based on the program path executed.

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

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

IX. Blockchain Integration

To integrate Nomos with a blockchain, we need a mechanism to *(i)* represent contracts and their addresses in the current blockchain state, *(ii)* create and execute transactions, and *(iii)* construct the global distributed ledger.

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

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

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

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

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

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

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

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

A transaction can either create new contracts, or update the state of existing contracts. In the former case, new contracts are added to the blockchain state, making them visible in the type of the configuration for subsequent transactions to access. In the latter case, it *acquires* the contracts it wishes to interact with, followed by an update in the contracts' internal state and *releases* them. Since the contract types are equi-synchronizing, they remain unchanged at the end of transaction execution. This ensures that the subsequent transactions can access the same contracts at the same type. In the future we plan to allow *sub-synchronizing* types that enable a client to release a contract channel not at the same type, but a *subtype*. The subtype can then describe the phase of the contract. For instance, the ended phase of auction contract will be a subtype of the running phase. The technical report [26] details the technique for serialized execution of transactions.

Deterministic Execution: Since blockchains rely on consensus among the miners, it is important to ensure deterministic execution of transactions. However, Nomos semantics has one source of non-determinism: the *acquire-accept* rule where an accepting contract latches on to any acquiring transaction. Our approach to resolve this non-determinism is to determinize the process scheduler based on some heuristics. Another promising approach is *record-and-replay* [44], [45]. The miner records the order in which the contracts are acquired in the ledger, which is then replayed by others to compute the current blockchain state.

Interpreter: The Nomos implementation provides two functionalities: *(i) inference:* takes a program as input, infers the potential and mode annotations and outputs a well-typed program (discussed in Section VIII), and *(ii) execution:* which takes a well-typed transaction program and a valid blockchain state as input, executes the transaction against the state and outputs a valid blockchain state. Internally, the blockchain state is represented as a configuration, i.e. set of contracts and linear assets stored inside them. Our implementation serializes the configuration using OCaml S-expressions so that their snapshots can be written to and read from a file. This makes the blockchain state persistent through transactions.

Attacker Model: Our blockchain model requires that all code submitted for execution is well-typed. The soundness theorem of Nomos (Theorem 1) then guarantees that execution of well-typed code cannot *damage* the blockchain state, or render it unusable. Thus, we capitalize on the restriction requiring adversarial code to be type correct. Furthemore, the Nomos type checker is carefully implemented to be linear-time in the size of the program. Thus, an adversary cannot cause denial-of-service by submitting programs that take too long to typecheck!

Deadlocks: The only language specific reason a transaction can fail is a deadlock in the transaction code. Our progress theorem accounts for this possibility of deadlocks. Since a valid blockchain state represents a stuck configuration of a particular form (only shared contracts in the configuration), we verify at the end of the transaction execution if the new configuration has this form. If not, we conclude that a deadlock occurred during the execution, and we simply abort the whole transaction. We maintain snapshots of the configuration after every transaction execution, so we simply revert to the previous valid blockchain state. It is the user's responsibility to issue a new transaction that does not deadlock. In the future, we also plan to employ deadlock prevention techniques [46] to statically rule out deadlocks.

X. Other Related Work

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

Smart Contract Languages: Existing smart contracts on Ethereum are predominantly implemented in Solidity [4], a statically typed object-oriented language influenced by Python and JavaScript. Languages like Vyper [47] 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 [48], on the other hand, makes state transitions explicit and avoids re-entrance by design. In contrast to our work, none of these languages use linear type systems to track assets stored in a contract.

Domain specific languages have also been designed for other blockchains apart from Ethereum. Typecoin [49] 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 [50] is formally modeled by the ρ-calculus, a reflective higher-order extension of the π-calculus. Michelson [51] is a purely functional stack-based language that has no side effects. However, none of these languages describe and enforce communication protocols statically. Scilla [52] is an intermediate-level language where contracts are structured as communicating automata providing a continuation-passing style computational model to the language semantics. Scilla does not use session types or linearity but features static gas bounds. A difference is that Nomos' bounds are not asymptotic and are proved sound with respect to a cost semantics. The Move programming language [53] is a flexible language based on Rust [54] to implement contracts on the Libra blockchain. Similar to Nomos, it provides the ability to define custom linear types to represent assets. However, it does not provide support to express contract protocols or gas usage.

Static Analysis: Analysis of smart contracts has received substantial attention [55], [56] recently due to their security vulnerabilities [57], [58]. KEVM [59] 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 [43] 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 [60] 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. Ethereum contracts are also translated to F* [61] to prove runtime safety and functional correctness, although they do not support all syntactic features. VeriSol [62] is a highly-automated formal verifier for Solidity that can produce proofs as well as counterexamples and proves semantic conformance of smart contracts against a state machine model with access-control policy. However, in contrast to Nomos, where guarantees are proved by a soundness proof of the type system, static analysis techniques often do not explore all program paths, can report false positives that need to be manually filtered, and miss bugs due to timeouts and other sources of incompleteness.

Session types and Resource analysis: Session types were introduced by Honda [11] as a typed formalism for inter-process dyadic interaction. They have been integrated into a functional language in prior work [15]. However, this integration does not account for resource usage or sharing. Sharing in session types has also been explored in prior work [24], 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 [13], [63]. Automatic amortized resource analysis (AARA) has been introduced as a type system to derive linear [18] and polynomial bounds [21] for functional programming languages. Resource usage has also previously been explored separately for the purely linear process layer [27], but were never combined with shared session types or integrated with the functional layer.

XI. Conclusion

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

In future work, we plan to explore refinement session types [64], [65] for expressing and verifying functional correctness of contracts against their specifications and to target open questions regarding a blockchain integration. These include the exact cost model, fluctuation of gas prices, and potential compilation to a lower-level language. Since Nomos has a concurrent semantics, we also plan to support parallel execution of transactions using speculation techniques [66].

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," http://bitcoin.org/bitcoin.pdf, 2008.

[2] G. Wood, "Ethereum: A secure decentralized transaction ledger," http://gavwood.com/paper.pdf, 2014.

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

[4] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*, 1st ed. USA: Apress, 2017.

[5] D. Siegel, "Understanding the dao hack for journalists," https://medium.com/@pullnews/understanding-the-dao-hack-for-journalists-2312dd43e993, Jun. 2016.

[6] B. I. I. Initiative, "B3i," 2008.

[7] A. Law, "Smart contracts and their application in supply chain management," Ph.D. dissertation, Massachusetts Institute of Technology, 2017.

[8] V. Morabito, "Smart contracts and licensing," in *Business Innovation Through Blockchain*. Springer, 2017, pp. 101–124.

[9] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, vol. 310, 2016.

[10] "Welcome to liquidity's documentation!" http://www.liquidity-lang.org/doc/index.html, Aug. 2018, accessed: 2018-11-04.

[11] K. Honda, "Types for dyadic interaction," in *4th International Conference on Concurrency Theory (CONCUR)*. Springer, 1993, pp. 509–523.

[12] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *7th European Symposium on Programming (ESOP)*. Springer, 1998, pp. 122–138.

[13] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," in *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2008, pp. 273–284.

[14] L. Caires and F. Pfenning, "Session types as intuitionistic linear propositions," in *21st International Conference on Concurrency Theory (CONCUR)*. Springer, 2010, pp. 222–236.

[15] B. Toninho, L. Caires, and F. Pfenning, "Higher-order processes, functions, and sessions: a monadic integration," in *22nd European Symposium on Programming (ESOP)*. Springer, 2013, pp. 350–369.

[16] F. Pfenning and D. Griffith, "Polarized substructural session types," in *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. Springer, 2015, pp. 3–22.

[17] P. Wadler, "Propositions as sessions," in *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2012, pp. 273–286.

[18] M. Hofmann and S. Jost, "Static Prediction of Heap Space Usage for First-Order Functional Programs," in *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, 2003.

[19] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann, "Static Determination of Quantitative Resource Usage for Higher-Order Programs," in *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, 2010.

[20] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate Amortized Resource Analysis," in *38th Symposium on Principles of Programming Languages (POPL'11)*, 2011.

[21] J. Hoffmann, A. Das, and S.-C. Weng, "Towards Automatic Resource Bound Analysis for OCaml," in *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[22] Q. Carbonneaux, J. Hoffmann, T. Reps, and Z. Shao, "Automated Resource Analysis with Coq Proof Objects," in *29th International Conference on Computer-Aided Verification (CAV'17)*, 2017.

[23] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, pp. 1–102, 1987.

[24] S. Balzer and F. Pfenning, "Manifest sharing with session types," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. ICFP, pp. 37:1–37:29, 2017.

[25] "Nomos implementation," https://github.com/ankushdas/Nomos, 2019, accessed: 2019-11-11.

[26] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar, "Resource-aware session types for digital contracts," *CoRR*, vol. abs/1902.06056, 2019. [Online]. Available: http://arxiv.org/abs/1902.06056

[27] A. Das, J. Hoffmann, and F. Pfenning, "Work analysis with resource-aware session types," in *33rd ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*, 2018.

[28] I. Cervesato and A. Scedrov, "Relating state-based and process-based concurrency through linear logic (full-version)," *Information and Computation*, vol. 207, no. 10, pp. 1044 – 1077, 2009, special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S089054010900100X

[29] S. Gay and M. Hole, "Subtyping for session types in the pi calculus," *Acta Informatica*, vol. 42, no. 2, pp. 191–225, Nov 2005. [Online]. Available: https://doi.org/10.1007/s00236-005-0177-z

[30] P. N. Benton, "A mixed linear and non-linear logic: Proofs, terms and models," in *8th International Workshop on Computer Science Logic (CSL)*, ser. Lecture Notes in Computer Science, vol. 933. Springer, 1994, pp. 121–135, an extended version appeared as Technical Report UCAM-CL-TR-352, University of Cambridge.

[31] J. Reed, "A judgmental deconstruction of modal logic," January 2009, unpublished manuscript. [Online]. Available: http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf

[32] K. Pruiksma, W. Chargin, F. Pfenning, and J. Reed, "Adjoint logic," Carnegie Mellon University, Tech. Rep., April 2018.

[33] U. D. Lago and M. Gaboardi, "Linear Dependent Types and Relative Completeness," in *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

[34] M. Avanzini, U. Dal Lago, and G. Moser, "Analysing the complexity of functional programs: Higher-order meets first-order," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional*

Programming, ser. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 152–164. [Online]. Available: http://doi.acm.org/10.1145/2784731.2784753

[35] N. Danner, D. R. Licata, and R. Ramyaa, "Denotational cost semantics for functional languages with inductive types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 140–151. [Online]. Available: http://doi.acm.org/10.1145/2784731.2784749

[36] E. Cicek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann, "Relational Cost Analysis," in *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[37] I. Radiček, G. Barthe, M. Gaboardi, D. Garg, and F. Zuleger, "Monadic Refinements for Relational Cost Analysis," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 2017.

[38] K. Crary, R. Harper, and S. Puri, "What is a recursive module?" in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999, pp. 50–63.

[39] F. Pottier and Y. Régis-Gianas, *Menhir Reference Manual*, 2019.

[40] B. C. Pierce and D. N. Turner, "Local type inference," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 1–44, Jan. 2000. [Online]. Available: http://doi.acm.org/10.1145/345099.345100

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

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

[43] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[44] M. Ronsse and K. De Bosschere, "Recplay: A fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, May 1999. [Online]. Available: http://doi.acm.org/10.1145/312203.312214

[45] C. Lidbury and A. F. Donaldson, "Sparse record and replay with controlled scheduling," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 576–593. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314635

[46] S. Balzer, B. Toninho, and F. Pfenning, "Manifest deadlock-freedom for shared session types," in *Programming Languages and Systems*, L. Caires, Ed. Cham: Springer International Publishing, 2019, pp. 611–639.

[47] "Vyper," https://vyper.readthedocs.io/en/latest/index.html, Aug. 2018, accessed: 2018-11-04.

[48] "Bamboo," https://github.com/cornellblockchain/bamboo, Aug. 2018, accessed: 2018-11-04.

[49] K. Crary and M. J. Sullivan, "Peer-to-peer affine commitment using bitcoin," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 479–488. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737997

[50] "Rholang," https://github.com/rchain/Rholang, Aug. 2018, accessed: 2018-11-04.

[51] "The michelson language," https://www.michelson-lang.com/, Aug. 2018, accessed: 2018-11-04.

[52] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with scilla," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 185:1–185:30, Oct. 2019. [Online]. Available: http://doi.acm.org/10.1145/3360611

[53] S. Blackshear, D. L. Dill, S. Qadeer, C. W. Barrett, J. C. Mitchell, O. Padon, and Y. Zohar, "Resources: A safe language abstraction for money," 2020.

[54] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018.

[55] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2018, pp. 9–16.

[56] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 51–78.

[57] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[58] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust - 6th International Conference, POST 2017*, 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[59] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Ştefănescu, and G. Rosu, "Kevm: A complete semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.

[60] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 116:1–116:27, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276486

[61] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: ACM, 2016, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[62] S. K. Lahiri, S. Chen, Y. Wang, and I. Dillig, "Formal specification and verification of smart contracts for azure blockchain," *CoRR*, vol. abs/1812.08829, 2018. [Online]. Available: http://arxiv.org/abs/1812.08829

[63] A. Scalas and N. Yoshida, "Less is more: Multiparty session types revisited," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 30:1–30:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290343

[64] A. Das and F. Pfenning, "Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description)," in *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), Z. M. Ariola, Ed., vol. 167. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 33:1–33:17. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12355

[65] ——, "Session types with arithmetic refinements," in *31st International Conference on Concurrency Theory (CONCUR 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), I. Konnov and L. Kovács, Eds., vol. 171. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 13:1–13:18. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12825

[66] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts," *CoRR*, vol. abs/1901.01376, 2019. [Online]. Available: http://arxiv.org/abs/1901.01376