Nomos: A Protocol-Enforcing, Asset-Tracking, and Gas-Aware Language for Smart Contracts

Ankush Das
Amazon, Cupertino, USA

Jan Hoffmann
Carnegie Mellon University, USA

Frank Pfenning
Carnegie Mellon University, USA

Abstract

Nomos is a programming language based on resource-aware session types that has been designed to address the domain-specific challenges developers face while programming smart contracts. This article presents the instantiation of Nomos to a concrete blockchain similar to Ethereum with modifications that make use of Nomos' unique features. This Nomos Blockchain organizes smart contracts in a novel *process-oriented* fashion where contracts correspond to processes that can be accessed by a session-typed channel that the process offers service on. Session types are central to expressing and statically enforcing contract protocols, essentially prescribing interaction sequences between contracts and their clients. Being in a Curry-Howard isomorphism with linear logic, session types also provide a natural representation of assets and guaranteeing that they are preserved across transactions. *Gas* is the only intrinsic currency of the Nomos Blockchain, thus users and miners are provided with exclusive gas accounts to pay for the execution cost of transactions. Resource-aware types automatically infer this execution cost in gas units, deduct them from the sender's account and ensure that execution is always free of out-of-gas errors. This article also presents the various components of the Nomos toolchain, highlighting two aspects: (i) simplicity of programming, and (ii) efficiency of the system. For the former, we present language features designed to improve programmer experience such as precise error messages, built-in maps, and support for designing Non-Fungible Tokens. For the latter, our virtual machine features linear-time type checking, fast inference of resource bounds via off-the-shelf linear programming (LP) solvers, and an interpreter that no longer needs to track execution cost at runtime (since it is statically inferred).

2012 ACM Subject Classification Software and its engineering → Design languages; Theory of computation → Linear logic; Software and its engineering → Concurrent programming languages

Keywords and phrases Language Design, Session Types, Resource Analysis, Linear Logic

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Blockchains such as Ethereum [47], Hyperledger Fabric [6], and Tezos [5] allow complex transactions like auctions, insurances, elections, and other binding contracts with applications in healthcare, real estate, and finance. Such complex transactions are realized by the execution of programs, also known as *smart contracts*. These programs are implemented either in general-purpose languages such as Python or Javascript, or domain-specific languages such Solidity [16] and Vyper [29].

The recent popularity of smart contracts has exposed that they often suffer from programming errors and vulnerabilities. And since smart contracts deal in financial assets, such errors often have monetary consequences. Bugs in smart contracts [7] have already caused losses to the order of billions of dollars. An infamous example is the $60 million attack on The DAO [41], exploiting a re-entrancy vulnerability. To make matters worse, blockchains are generally *public*, i.e., the state of every deployed contract is publicly visible, and *immutable*,

i.e. deployed contracts cannot be updated even if vulnerabilities are detected and publicly known. These unique aspects make blockchains a fertile ground for malicious attacks.

Thus, to prevent such attacks, it is vital to ensure contracts are safe and devoid of errors *before* deployment. There are two common approaches to address this issue: one is to apply static analysis and verification techniques both for low-level languages such as EVM bytecode [32, 44, 33] and high-level languages such as Solidity [42, 38]. A complementary approach is to design domain-specific languages [10, 14, 29, 39] that rule out common sources of vulnerabilities by design.

One such domain-specific language is Nomos [17]. The defining feature of Nomos is the use of *resource-aware session types* [18] to address the specific requirements of smart contracts. These include *(i)* enforcing contract interaction protocols, *(ii)* preventing accidental duplication or deletion of a contract's assets, and *(iii)* automatically inferring resource (or gas) usage. Nomos enforces contract protocols by relying on *binary session types* [25, 26, 27, 11, 43, 35, 45] to provide a succinct specification of the contract's communication behavior. The built-in *linearity* of session types is used to track assets guaranteeing that they are preserved across transactions. Nomos infers execution cost by utilizing *automatic amortized resource analysis (AARA)*, a type-based technique for automatically inferring cost bounds [24, 28, 22, 23, 12] relying on an LP solver. Moreover, Nomos delimits the interaction between different parties with shared modalities [8], thus preventing *re-entrancy attacks* by construction. Prior work [17] describes how these components are integrated into a unified general-purpose programming language.

In this article, we describe one possible instantiation of Nomos onto an *account-model* blockchain. To this end, we assume a *Nomos Blockchain* that can be seen as a variant of Ethereum, with specific design decisions that are enabled by Nomos' features. Compared to Ethereum that organizes contracts as data, a distinguishing feature of the Nomos Blockchain is its novel *process-oriented* design. Contracts are organized as processes hosted at a unique address (identified by the channel they offer) and contract transactions are executed by sending messages to the corresponding process. Another unique aspect of the Nomos Blockchain is that *gas is the sole currency* and users and miners have *gas accounts* with an associated gas balance. Since Nomos can statically compute an upper bound on the user-cost of transactions, this bound is automatically deducted from the sender's account. We also introduce a novel `Nomos.deposit{r}` operation that deposits r gas units in the transaction sender's account. This enables describing precise path-sensitive gas bounds and safely returns leftover gas during execution. Lastly, Nomos supports *gas amortization* by storing gas inside contracts and data structures which simplifies cost analysis and leads to more equitable gas-distribution schemes [19].

Since linearity is inherent with session types in Nomos, the support for token creation is flexible and general, and can be integrated with custom behavior such as assigning a unique identifier to each token or storing the addresses of all previous token owners. To ensure that tokens are scarace and non-fungible, we introduce declaration of *private types*. The Nomos type checker hides private type definitions from external clients who cannot create or destroy tokens deemed private. Such tokens can only be exchanged amongst external users or returned back to the host contract. Nomos' resource-aware session types enable the creation of denominations and can support variable exchange rates between tokens and gas without any modifications to the language.

To improve its usability, we extend the Nomos language along two directions: *simplicity of programming* and *efficiency*. For the former, our automatic gas inference engine alleviates significant programmer burden. Additionally, we introduce *(i)* a built-in map data structure

accompanied with syntactic sugar for ease of use which can be flexibly used to store both linear assets and non-linear data, and *(ii)* a bi-directional type checker with precise error messages. For efficiency, we have carefully designed the bi-directional Nomos type checker to be *linear-time* in the size of the program. This is important in a blockchain setting where transaction validation which entails type checking is part of the attack surface and can be exploited by an adversary for denial-of-service attacks. Moreover, static gas computation and the `Nomos.deposit{r}` operations together ensure safe return of leftover gas to the user [19] *without performing dynamic gas monitoring*. This simplifies the Nomos interpreter and avoids significant runtime overhead.

The Nomos Blockchain has been implemented in OCaml and is available open-source [20]. The prototype implementation supports creation of and depositing gas into user accounts, implementation and deployment of smart contracts, and issuing transactions that modify the overall blockchain state. Users can also access the prototype via a web interface (hosted at `https://www.nomos-lang.org`).

2 The Nomos Blockchain System

While the Nomos language is not tied to a particular blockchain or, more generally, the implementation of smart contracts, we describe in this paper the adaptation of Nomos to a specific blockchain that we refer to as the Nomos Blockchain.

The Nomos Blockchain is inspired by Ethereum [47] but there are some differences that are enabled by and highlight Nomos' most prominent features. The blockchain state of Ethereum follows a *data-oriented* model where contracts are stored in the form of a distributed database. Transactions can then either add new contracts to the database or modify the state of the existing contracts by issuing functions provided by them.

The blockchain state of the Nomos Blockchain consists of a *configuration*, which contains a finite set of well-typed processes. A well-formed blockchain state provides a set of *shared channels* that uniquely identify the contracts that are available in the state. Each contract is implemented by a process that can either store linear assets or (functional) data structures. Linear assets are also represented as processes offering service along an *exclusive private channel*, which can only be referred by their owner. Both shared channels offered by contracts and linear channels offered by assets are typed with session types [25, 11, 8] that dictate their communication interface.

On the Nomos Blockchain, a transaction is an implementation of yet another process that is executed together with the configuration in the current blockchain state. The transaction process can either create new contracts processes, or modify existing contracts by interacting with them through their shared channels. Figure 1 describes the evolution of a blockchain state hosting an auction contract. Initially, the auction contract contains 2 linear channels $\$m_1$ and $\$m_2$, representing bids from 2 different clients (note that assets are represented using linear channels). Next, a client submits a transaction implemented as a `Bid Process` that is responsible for placing a bid in the auction. This process stores the linear channel $\$m_3$ representing the bid amount inside it. The bid process first *acquires* the auction contract hosted at a *shared* channel `#a` followed by transferring channel $\$m_3$ to the auction contract. Next, the bid process *releases* channel `#a` detaching from the contract and terminates leaving the auction contract with 3 linear channels, thus successfully placing a bid in the auction. Section 3 describes the code for the above transaction in Nomos syntax.

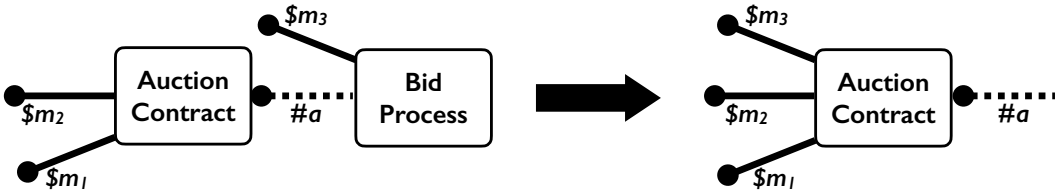

Figure 1 Transaction called Bid Process placing a bid ($\$m_3$) in the Auction Contract

Gas Accounts

Gas is the *only primitive currency* in the Nomos Blockchain. We therefore assume working in an *account-model* blockchain where we assign *gas accounts* to users. Formally, gas accounts are assigned an account name, an authentication password, and a gas balance. Like in Ethereum, gas must be consumed to pay for the execution of a transaction.

A unique feature of Nomos is that it allows *gas amortization* by storing gas inside data structures. This gas can be utilized to pay for the cost of expensive operations such as iterating over data structures. For instance, in an auction contract, users can pay additional gas while bidding into the auction which is stored inside the contract and consumed later to pay for the cost of iterating over all the bids and returning them to their respective bidders.

Nomos' type system statically and automatically assigns to each transaction a worst-case gas consumption. This worst-case bound accounts for the gas that is potentially transferred to contracts and actual execution cost as defined by a cost semantics. When a user submits a transaction, Nomos' embedded LP solver computes the transaction's worst-case gas cost and automatically deducts that amount from the user's gas account. If the user's gas balance is insufficient to pay for the transaction cost, the transaction is aborted. In addition, if there's any gas leftover after the transaction terminates, it is automatically returned back to the user via `Nomos.deposit` operations. Thus, at runtime, Nomos guarantees that transaction execution is free of out-of-gas errors.

Finally, gas accounts are the only external setup to bootstrap the Nomos Blockchain. If users wish to store additional data or tokens in their account, they must do it by creating additional smart contracts that are internal to the blockchain.

3 The Nomos Language with an Example

We highlight the main features of the Nomos language using the implementation of an auction contract. An auction operates in two phases: a *running* phase where bidders bid into the auction followed by an *ended* phase where bidders collect their earnings. If a bidder wins the auction, they receive the *lot*, otherwise they receive their bid back.

The first key idea behind Nomos is to express contract protocols via *session types*. As an illustration, the auction session type is defined as

```
type auction =
  /\ <{20}| +{running : money -o |{3}> \/ auction,
              ended : +{won : lot * |{5}> \/ auction,
                        lost : money * |{0}> \/ auction}}
```

We first ignore the operators `<{q}|` and `|{q}>` for natural numbers `q` (described later) and describe the remaining type. The type initiates with `/\` indicating that auction is a *shared* session type [8] that must be acquired by a bidder to interact with the contract. Shared session types guarantee that bidders interact with the auction in *mutual exclusion* and their

interaction with the auction executes atomically. Once the action contract is acquired, it replies either with `running` or `ended` indicating the phase of the auction. In the former case, the auction receives `money` using the `-o` constructor followed by the bidder releasing the contract matching the `\/` constructor in the type. In the latter case, the auction determines if the bidder won or lost the election. If the bidder wins, the auction sends the `won` label followed by sending the `lot` using the `*` constructor. If the bidder loses, the auction sends the `lost` label followed by returning the bidder's `money` back. In either case, the type recurses back to `auction` after a `\/` indicating that the bidder must release the auction.

The second key feature of Nomos is that it statically enforces that assets are never duplicated nor discarded, but only transferred between processes. Session types in Nomos [11] are carefully designed such that they are rooted in linear logic [21]. This linear type system tracks the assets stored in a process. For instance, the auction contract treats `money` and `lot` as linear assets, which is witnessed by the use of the linear logic operators $\multimap$ and $\otimes$ (`-o` and `*` in Nomos syntax) preventing their accidental deletion or duplication.

Finally, an important aspect of smart contracts is their *execution cost*. The third and final advantage of Nomos is that it uses resource-aware session types [18] to statically express the execution cost of a transaction. They operate by assigning a *potential* (can also be thought of as gas) 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 type system is parametric in a *cost model* that assigns a cost to each primitive operation. This cost model is specified by the Nomos Blockchain (equivalent to the gas schedule that specifies the gas cost of each bytecode operation).

Resource-aware session types express the potential as part of the session type using the operators `<{q}|` and `|{q}>`. The `<{q}|` operator prescribes that the client must send q potential to the contract. Dually, `|{q}>` prescribes that the contract must return q 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 20 (taking the maximum execution cost all across branches). Thus, we require the client to send 20 units of potential at the start of a session using `<{20}|`. Then, in each branch, depending on the execution cost of that branch, the leftover potential is returned back to the client. For instance, the execution cost of the `won` branch of the `auction` type turns out to be 15 units, therefore the contract returns 5 potential units to the client using `|{5}>`. This mirrors gas usage in smart contracts (potential is equivalent to gas), 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. All the above potential annotations have been automatically inferred by the LP solver integrated with the Nomos type checker. The soundness proof of the type systems implies that the potential indeed bounds the gas cost.

Process Implementations

Next, we describe how the aforementioned session types are realized by concrete contract and transaction code. Since the auction operates in two phases, we have two main processes: `running_auction` for the running phase, and `ended_auction` for the ended phase. The type and definition of `running_auction` process is presented in Figure 2. The keyword `contract` is used to define a contract process that offers the shared channel `#a` along which it can receive bid and withdraw requests. The process also uses two linear channels: `$bm` represents the mapping from `address` to `money`, and `$l` represents the `lot`. To syntactically distinguish channels from functional variables, Nomos prefixes linear channels with `$` (e.g. `$bm`) and

```
contract running_auction :
($bm : Map<address, money>), ($l : lot) |- (#a : auction) =
    $la <- accept #a ;                  % accept acquire request
    get $la {20} ;                      % get 20 potential units
    $la.running ;                       % send 'running' label
    $m <- recv $la ;                    % receive money from bidder
    pay $la {3} ;                       % pay leftover potential
    #a <- detach $la ;                  % detach from bidder
    let addr = Nomos.GetTxnSender ;     % get bidder's address
    $bm.insert(addr, $m) ;              % insert bid into bidmap
    #a <- run_or_end $bm $l             % call 'run_or_end' process

transaction bid_proc : ($m : money), (#a : auction) |- ($d : 1) =
  $la <- acquire #a ;                % acquire auction contract
  pay $la {20} ;                     % pay 20 potential units
  case $la (                         % branch on label received
    running => send $la $m ;         % send money to contract
               get $la {3} ;         % get leftover potential
               #a <- release $la ;   % release the auction contract
               close $d              % terminate the transaction
  | ended => abort )                 % abort if auction has ended
```

Figure 2 Contract and transaction code for the running phase of the auction

shared channels with `#` (e.g. `#a`). The process closely follows the protocol described by the `auction` session type. It first accepts the acquire request from the bidder followed by receiving 20 potential units. Since this process represents the running phase of auction, it sends the `running` label, receives the money from the bidder in `$m`, pays the leftover 3 potential units, and detaches from the bidder.[1] Internally, the process then computes the sender's address using the built-in `GetTxnSender` expression, and inserts key-value pair (`addr`, `$m`) into the `$bm` map. Maps have a built-in session type (explained in Section 6.1) and can be used as such, but we simplify programming by introducing syntactic sugar construct `$bm.insert(addr, $m)`. Finally, the process calls the `run_or_end` process which decides whether to call `running_auction` (if the auction is still running) or `ended_auction` (if the auction has moved to the ended phase).

The transaction that places this bid in the auction from the other end of the auction channel is implemented as `bid_proc` in Figure 2. The `transaction` keyword mandates that this process interacts with a shared contract and offers type `1` (we require all transaction processes to offer type `1` so that they can terminate by simply closing the offered channel). It uses linear channel `$m` representing the bid, and the shared channel `#a` that connects to the auction contract. The process initiates with acquiring the auction contract, pays 20 potential units, and case analyzes on the response. If the response is **running** (indicating that the auction is running), the process sends the money, gets the leftover potential, releases the contract, and terminates the transaction. If the response is **ended**, we simply abort the transaction. Note how the `running_auction` and `bid_proc` processes perform matching dual actions on the auction channel, as governed by the `auction` session type. This is precisely the advantage of using session types as a specification of the contract communication protocol.

[1] Note that the user does not have to provide the potential (20 and 3) for the forms `get` and `pay`. These numbers are added automatically in an elaboration phase that is part of the type inference.

Automatic Gas Computation

Nomos is embedded with an inference engine that automatically computes the potential annotations, both in the type definitions as well as the process implementations. Therefore, the programmer only needs to specify the following type for the auction

```
type auction =
  /\ <{*}| +{running : money -o |{*}> \/ auction,
              ended : +{won : lot * |{*}> \/ auction,
                        lost : money * |{*}> \/ auction}}
```

Using $*$ annotations minimizes the programmer burden who does not need to compute exact potential annotations and execution cost. Correspondingly, the potential in `pay` and `get` expressions in the code is also unspecified by a user. For instance the user-level code of `running_auction` contains `pay $la {*}` and `get $la {*}`. Nomos' inference engine the substitutes the star annotations with variables. For the auction type, this reduces to

```
type auction =
  /\ <{v0}| +{running : money -o |{v1}> \/ auction,
              ended : +{won : lot * |{v2}> \/ auction,
                        lost : money * |{v3}> \/ auction}}
```

In the process `running_auction`, we ellaborate the potential transfer to `pay $la {v0}` and `get $la {v1}`. Then, the type checker generates linear constraints based on applying the typing rules that essentially consumes one unit of potential to execute one operation (that is the cost model used for this example). Collecting the constraints generated by the type checker, we obtain

$$v_0 - v_1 \geq 17 \qquad v_0 - v_2 \geq 15 \qquad v_0 - v_3 \geq 20$$

Finally, these constraints are shipped to an 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. For the above example, we obtain the solution: $v_0 = 20, v_1 = 3, v_2 = 5, v_3 = 0$.

4 Custom Tokens

The naturally encoded linear types in Nomos allow users to flexibly define tokens with custom behavior. The only additional feature that is needed are *private types* that enable a contract to control how tokens are minted. Then we can express gas tokens that carry a fixed quantity of gas similar to the gold standard, tokens with variable or fixed exchange rates, and denominations that ensure that large quantities of tokens can be exchanges efficiently.

Private Types

Nomos has a structural type system, which means that type equality is based on the structure of the types. If we would introduce a new token type then other entities could freely create processes of an equal type that would be indistinguishable from our token. To enable users to control the creation of tokens of a given type, we have to introduce nominal types that are distinguished based on their name.

To control the minting of tokens, we use *private types*, declared with a `private` keyword in front of the type definition. The important distinction between private and regular types

is that the type system hides the definition of private types from external clients. For an external client two private types are equal if and only if their names are equal.

The scope of a private type declaration is the transaction where it is defined. Concretely, we can define a smart contract which is the exclusive owner with a transaction like the following, which contains omitted code that spawns a contract of type `token_contract`.

```
private type token = ...
type token_contract =
  /\ <{*}| &{buy : token * \/ token_contract,
             sell : token -o |{*}> \/ token_contract} ...
```

We omit the definition of the token type; the contract owner can decide the type they would like. To interact with the contract, a user must acquire it followed by paying some amount of gas (which will be statically determined). Then, the client decides whether to buy or sell a token. In the former case, the client sends the `buy` label and the contract sends back a token. In the latter case, the clients sends the `sell` label and a token, and receives some leftover gas. Then, in either case, the contract detaches from the client.

Since the `token` type is private, we are guaranteed that no other transaction outside the contract can mint this token; it can only be transferred between users but not otherwise manipulated. To ensure that the private type has a unique name, it is prefixed by the transaction ID in the blockchain state.

Gas Tokens

An interesting token type is one that is guaranteed to be backed by a fixed amount of gas. This is similar to the gold standard monetary system in which a central bank guarantees that money corresponds to a claim to a certain amount of gold, which is backed by gold reserves. In Nomos, such a gas reserve can be decentralized and stored with the tokens.

But how do we store the reserve in the tokens? Since Nomos allows gas amortization, we can store gas inside tokens to assign them a fixed (minimal) value.

```
private type token_five = |{5}> 1
asset createFive : . |- ($x : token_five) = pay $x {5} ; close $x
```

The type `token_five` specifies that a process of this type will send 5 gas units and terminate. The `createFive` process produces a token of type `token_five`.

A process within the scope of the type definition with access to a token of type `token_five` can receive these 5 units which can be used to pay towards execution cost. A client outside the type definition is not able to extract the 5 gas units. However, the definition of the token is available on the blockchain and thus the user has the guarantee that the token indeed stores 5 gas units. A contract that mints private tokens of type `token_five` could provide the functionality of dismantling a token and returning the gas that is stored inside.

Pegged Tokens

Pegged tokens are minted by a process that guarantees a fixed gas exchange rate for tokens. The private token type of a pegged token can be arbitrary. To see how a contract can offer a fixed exchange rate, say 5 gas units for a token, consider again the type `token_contract`. The implementation of such a contract can store a list of processes offering type `token_five`, which internally contain 5 gas units per element. If a client buys a token, the contract spawns a new process that offers type `token_five` and adds this process to the list. Similarly, the contract removes a process from the list if a client sells a token (in fact, we cannot return

a token if the list is empty which is not reflected in the type for simplicity.) In this way, Nomos' gas bound inference will determine that sending the label `buy` must have a cost of 5 additional gas units to pay for the spawning. Similarly, sending the label `sell` should cost 5 gas units less than the actual gas consumption (due to 5 gas units being freed up during the removal of the element). This corresponds to the desired fixed exchange rate.

Free-Floating Tokens

Tokens are particularly useful if they are *free-floating*, that is, they can be traded and exchanged to gas at a market rate. To define tokens with a flexible exchange rate, we first introduce natural numbers that store gas inside them. We call this type `gasnat` defined as

```
type gasnat = +{succ : |{1}> gasnat, zero : 1}
```

This type dictates the following protocol: the provider initiates with an internal choice. It either sends the `succ` message followed by sending 1 gas unit and recursing back to `gasnat`. Or it sends the `zero` message and terminates. The unique aspect of `gasnat` is that it can store a dynamic amount of gas equal to the value of the natural number since it stores 1 gas unit per element.

A contract can create free-floating tokens by pegging them against `gasnat`. We present a modified version of the `token_contract` type to achieve this.

```
type token_contract =
/\ <{*}| &{buy: gasnat -o +{suff: token * |{*}> \/ token_contract,
                           insuff: gasnat * |{*}> \/ token_contract},
          sell: token -o gasnat * |{*}> token_contract
```

The protocol of buying changes slightly and initiates by the client sending a channel of type `gasnat` (after sending gas and `buy` label). If the value of `gasnat` is sufficient to pay for the *current value* of the token, the contract replies with the `suff` label followed by the free-floating token. In case the value of `gasnat` is not sufficient to buy the token, the contract replies with `insuff` and sending the `gasnat` back. The sell functionality simply returns the gasnat that reflects the current market price. However, it could be altered so that the client can specify a minimal price they are willing to accept.

Non-Fungible Tokens

Like traditional money, cryptocurrencies like Bitcoin and Ethereum are *fungible* assets, i.e., one bitcoin (or ether) is completely indistinguishable from another. Recently, a new kind of asset has been gaining popularity in the blockchain community, *non-fungible tokens* (NFTs). An NFT is a uniquely identifiable non-interchangeable asset that can be stored on the blockchain. NFTs have been widely used to buy and sell digital assets in the art, virtual gaming, and music and movie industry [46]. The uniqueness of each NFT makes them highly valuable. In 2021 alone, collectors and traders spent more than $20 billion on NFTs [40].

The token types so far are essentially interchangeable; the type system cannot distinguish between two tokens of type `token_five`. But since session types support more general communication, we can use this feature to encode more sophisticated tokens with custom behavior. For instance, we can define a token type `uniqueID` that internally stores a unique identifier with each token.

```
private type uniqueID = &{ID : <{1}| uint ^ uniqueID}
asset createUnique : (n : uint) |- ($u : uniqueID) =
  case $u ( ID => get $u {1} ; send $u n ; $u <- uniqueToken n)
```

The type `uniqueID` describes the following interface: on receipt of label `ID`, it expects one gas unit from its user and then sends back a unique integer and recurses back to the original type. The process `createUnique` takes a unique integer `n` as argument and provides this interface. If it receives the `ID` label, it gets 1 gas unit, consumes it to pay for the cost of sending back the identifier `n` and recurses.

Denominations

A known issue with tokens that are represented by linear types is that exchanging large quantities of such tokens can be inefficient and lead to expensive transactions with high gas consumption. For instance, a naïve way of sending 10000 tokens would be to send a list of tokens of length 10000. Fortunately, Nomos session types are expressive enough to build up denominations that enable to exchange large quantities of tokens efficiently.

```
private type token = ...  // definition omitted
private type five_tokens = token * token * token * token * token
asset createFive :
($x1 : token), ($x2 : token), ... ($x5 : token) |- ($x : token) =
  send $x $x1; send $x $x2; send $x $x3 ; send $x $x4 ; $x <-> $x5

private type ten_tokens = five_tokens * five_tokens
asset createTen :
($x1 : five_tokens), ($x2 : five_tokens) |- ($x : ten_tokens) =
  send $x $x1 ; $x <-> $x2
```

Consider a token of type `ten_tokens`. The user has to pay the cost once while creating a process offering type `ten_tokens`. Once the process has been created, its channel can be freely exchanged by sending a *single message*! For instance, if a user wants to send a channel `$x : ten_tokens` to another user over channel `$c`, they only need to use the expression `send $c $x`. Pictorially, this is exactly what happens in Figure 1! The channel `$m3` is simply transferred from the `Bid Process` to the `Auction Contract` using the construct `send $la $m` (in the `bid_proc` code in Figure 2). Internally, `$m3` could potentially represent a large sum of money but the transfer only requires the exchange of one channel which occurs quite efficiently in practice. Thus, the flexibility of the linear type system avoids expensive transactions by building up large denominations.

5 Exact Gas Bound Computation

Statically, the Nomos toolchain can only compute an upper gas bound on the execution cost, since the execution cost of different program paths is often different. But users should be charged only for the actual gas consumption instead of this worst-case upper bound. At runtime, however, the actual execution cost might be significantly lower than the upper bound, in which case we need to return the leftover gas back to the client. To return this leftover gas, existing blockchains such as Ethereum and Diem [9] use a *dynamic gas monitor* that tracks the gas cost during execution. Thus, despite the benefits of static gas analysis, blockchains still need to track gas consumption at runtime which creates a significant overhead. For the Diem blockchain, this overhead is about 20% of execution time [9].

To eliminate this runtime overhead, we introduce a new syntactic construct called `Nomos.deposit{r}` that enables us compute exact gas bounds [19]. The main obstacle in computing exact bounds is *branching* since *(i)* gas cost may vary along different branches, and *(ii)* statically predicting execution path is undecidable, in general. However, if we can

ensure that branches at any program point have *equal* execution cost, the execution path can no longer impact the gas cost. This is where the novel `Nomos.deposit{r}` operation comes in. Consider a process expression `if b then P else Q` such that gas cost of `P` is p and gas cost of `Q` is q. If $p > q$, we augment the branch `Q` with `Nomos.deposit{p-q}`. Executing `Nomos.deposit{r}` at runtime deposits r gas units in the sender's account. Therefore statically, the gas cost of this expression is r. Hence, the transformed process expression `if b then P else Nomos.deposit{p-q} ; Q` has cost $p - q + q = p$ in the `Q` branch as well, equalizing the gas cost across both branches. In general, the difference between the worst-case upper gas bound and the exact gas bound on a particular execution trace is covered by the `Nomos.deposit{r}` operations that exist on that trace. This simple modification safely returns the excess gas units back to the sender and thus, we no longer need to monitor the execution cost at runtime.

The deposit annotations do not have to be inserted into the code manually. To reduce programmer overhead, we automatically augment both branches with the expression `Nomos.deposit{*}`. During type inference, the values of all such $*$ annotations are inferred by the LP solver. Then, all the `Nomos.deposit{*}` expressions that are substituted with 0 value are simply deleted from the program.

Paying Miners' Transaction Fee

Each miner in Nomos is assigned an exclusive gas account. When mining a block of transactions, Nomos automatically adds a special *implicit* transaction to the end of the block that is responsible for paying the block miner. The miner payment is directly proportional to the gas cost of the transactions in the block. In existing blockchains, the dynamic gas monitor tracks each transaction's gas cost which then determines the miner's fee.

However, as we just described, the Nomos Blockchain performs no dynamic gas monitoring. So, how do we determine the miner's fee? Remarkably, static gas computation along with `Nomos.deposit` operations come to our rescue! Since gas is inherently part of the Nomos Blockchain, the total gas stored in a blockchain state is visible in the type of the configuration. Thus, the difference in the total gas in the blockchain state before and after a transaction execution provides a worst-case upper bound on the transaction gas cost. Deducting the total gas deposited in the sender's account using `Nomos.deposit` operations from this difference exactly determines the transaction execution cost. Our implicit transaction computes this cost for each transaction in the block, determines the miner's fee from this cost, and adds an operation to transfer this fee from the sender's to the miner's gas account. Of course, it would be also possible to add additional gas that is created (or mined) by the addition of the new block.

6 Implementation

We have developed an open-source Nomos implementation [20] in OCaml (8469 lines of code). The implementation contains a lexer and parser (594 lines), a type checker (3486 lines), a pretty printer (531 lines), a cost inference engine with an LP solver interface (969 lines of code), and an interpreter (1942 lines). Figure 3 outlines the Nomos toolchain with all its components. A Nomos program is first parsed into an abstract syntax tree, which is then instrumented with work constructs to realize the cost model. The program is then fed into the inference engine which computes potential annotations by generating LP constraints and shipping them to the LP solver that computes a satisfying assignment while minimizing the total potential. The abstract syntax tree is then type checked to verify its session type

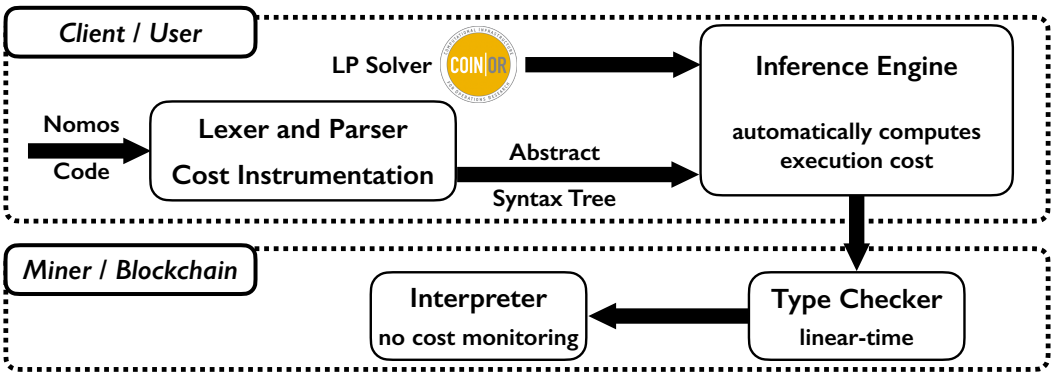

Figure 3 The Nomos Toolchain

and potential annotations. Finally, the interpreter runs the program against the current blockchain state to obtain a new blockchain state. We follow a brief description of each component of the toolchain.

Lexer, Parser, and Cost Instrumentation

The Nomos lexer and parser have been implemented in Menhir [37], an LR(1) parser generator for OCaml. A Nomos program is a list of mutually recursive (possibly private) type and process definitions. The syntax for definitions is

```
<private?> type v = A
<role> f : (x1 : T), (#c2 : A2), ... |{q}- ($c : A) = P
```

The first line describes a type definition: `A` is the type expression that stands for the definition of type name `v` (e.g. `auction` type in Section 2). The second line describes a process definition. We start by writing the process role that desribes the purpose of the process: *(i) asset:* for processes that offer a linear channel (e.g. token processes in Section 4), *(ii) contract:* for processes that offer a shared channel and denote a smart contract (e.g. `running_auction` in Section 3), or *(iii) transaction:* for processes that are added to the configuration and modify the blockchain state (e.g. `bid_proc` in Section 3). These role descriptions are not mere annotations but enforce certain well-formedness restrictions on the process declaration (more details can be found in the theory of Nomos [17]). Next, `f` denotes the process name followed by its input and output types. The input types are denoted by a context that contains both functional variables and session-typed channel variables: e.g., `x1 : T` defines a functional variable `x1` of type `T` ; `#c2 : A2` defines a channel `#c2` with type `A2`. The `{q}` on the turnstile denotes the potential stored in the process, and `$c : A` denotes that the process offers service of type `A` on channel `$c`. Finally, `P` stands for the process expression corresponding to `f`. We follow the syntactic convention that functional variables (e.g. booleans, integers, etc.) are denoted by regular variables (e.g., `x1 : T`), linear session-typed channels are prefixed by `$` and shared channels are prefixed by `#`.

Once a program has been converted into an abstract syntax tree, we instrument it with work constructs based on the cost model. The cost model intuitively defines the execution cost of each construct. The instrumentation engine takes the program and the cost model as input and produces a program with work constructs inserted at appropriate places. We use the following rule ($[\![P]\!]$ denotes the work-instrumented version of process P):

$$[\![S\ ;\ P]\!] ::= \mathsf{work}\,\{C_S\}\ ;\ S\ ;\ [\![P]\!]$$

Here, S is a process construct with P as its continuation, and C_S denotes the cost of construct S according to the given cost model. This instrumentation simplifies the cost analysis which can simply assign a cost of c to $\mathsf{work}\,\{c\}$ and cost 0 to all other process expressions.

LP Solver for Gas-Cost Inference

Using ideas from existing techniques for type inference for AARA [24, 23], we reduce the reconstruction of potential annotations to linear optimization. As a first step, the programmer can indicate unknown potential using $*$ annotations. Such annotations occur in the following program constructs:

- `|{*}>` and `<{*}|` annotations in session type definitions used for transferring gas
- `|{*}-` on the turnstile in a process definition denotes the gas stored in a process
- `T list {*}` is used to store a constant amount of gas per list element so that functional data structures can contain gas as well
- `pay $x {*}` and `get $x {*}` process expressions
- `Nomos.deposit{*}` to deposit gas into the transaction sender's account

Thus, gas constructs are quite pervasive in a Nomos program and computing their precise annotation can be quite challenging for a programmer. Thus, using $*$ annotations reduces a significant burden for users and developers.

The Nomos cost instrumentation engine then replaces these $*$ annotations with gas variables. Next, Nomos' inference engine generates linear constraints on these gas variables (as described in Section 3). These constraints are then shipped to an off-the-shelf LP solver called Coin-OR [31] which minimizes the total sum of gas annotations to obtain precise bounds. The LP solver returns a satisfying assignment for all gas variables which is then substituted back into the program, thus yielding a concrete program that can be fed into the type checker.

Type Checker

Instead of performing full type inference, the Nomos type system is based on bi-directional type checking [36]. Intuitively, the programmer provides the initial type of each variable and channel in the declaration (as shown in the examples). While checking the process definition, the type checker reconstructs the intermediate types following the typing rules. If there is a type error in the program, reconstruction fails and this program point is accurately identified as its source. Therefore, this bi-directional type checker further assists the programmer with precise error messages.

Another reason to use a bi-directional type checker is its efficiency. Full type inference can be very expensive, in the worst-case. However, in the Nomos Blockchain, type checking is a part of transaction validation and hence, an attack surface. If type checking is too slow, malicious users can issue transactions that take too long to type check, effectively causing a denial-of-service attack. Realizing this, we restrict type equality to *reflexivity* (constant time) ensuring that *type checking is linear time in the size of the program.*

Interpreter

A blockchain state in Nomos is described using a configuration. At runtime, a configuration is a set of processes and messages interacting with each other through communication channels. The Nomos interpreter uses OCaml S-expressions to represent configurations. The interpreter has read/write functionality which converts a configuration into an S-expression

and vice-versa. This helps persist the configuration across transactions. The interpreter takes a configuration as an S-expression and a transaction program as input, executes the program against the configuration, and produces an output configuration, which is finally converted to an S-expression. The interpreter is quite efficient in practice, since it no longer needs to track execution cost at runtime.

6.1 Map Data Structure

To further simplify programming, we enhance Nomos with blockchain-specific features. The most prominent feature is the addition of *maps*, the most widely used data structure in smart contracts. It is often used in contracts to store a mapping from users to their balance. We provide surface syntax to make it easier for programmers to interact with maps.

Since Nomos combines functional and session-typed programming, we need to dinstinguish between regular maps from maps that store linear channels. Although the two maps differ in their statics and semantics, we want to provide a unified syntax for ease of programming. We first describe the session type for a regular map with key type `kt` and value type `vt`.

```
type Map<kt, vt> = &{insert : kt -> vt -> Map<kt, vt>,
                     delete : kt -> vt option ^ Map<kt, vt>,
                     size : int ^ Map<kt, vt>,
                     close : 1}
```

The map is implemented with a recursive session type initiating with an external choice. It accepts one of four messages: `insert`, `delete`, `size` or `close`. In the case of `insert`, the map receives a key and value and inserts the pair in the dictionary. If the key already exists, the existing value is overwritten. In the case of `delete`, the map receives a key and returns an optional value depending on whether the key exists in the map or not. In the case of `size`, the map returns an integer corresponding to its size. In each of these three cases, the type then recurses back to `Map<kt, vt>`. Finally, in the case of `close`, the map simply terminates with a `close` message.

For a linear map, we use the same type name but the checker distinguishes linear and regular maps based on whether `vt` is a linear session type or not. The type is as follows:

```
type Map<kt, vt> = &{insert : kt -> vt -o Map<kt, vt>,
                     delete : kt -> vt option * Map<kt, vt>,
                     size : int ^ Map<kt, vt>,
                     close : +{empty : 1,
                               nonempty: Map<kt, vt>}}
```

Since `vt` is a linear type, we use `-o` and `*` constructors to exchange them. As an additional requirement, closing a linear map is only allowed when it is empty. Hence, on receiving a `close` message, a linear map will respond with either the `empty` message followed by termination. Or with the `nonempty` message followed by recursing back to its original type.

Finally, we provide surface syntax for easier programming with maps.

- `$m <- new Map<kt, vt>`: for creating a new map `$m` of key type `kt` and value type `vt`
- `$m.insert(k, v)`: for inserting key `k` and value 'v' into map `$m`. If the map is linear, the value is replaced by a channel `$v`.
- `v = $m.delete(k)`: for deleting key `k` from map `$m`. If the map is linear, the expression changes to `$v <- $m.delete(k)`.
- `n = $m.size`: for obtaining the size of map `$m` and storing it in variable `n`.
- `$m.close`: for closing the map.

7 Related Work

We classify the related work in two main categories: *(i)* state-of-the-art smart contract languages and *(ii)* static analysis techniques particularly directed to gas computation.

Solidity [16] and Vyper [29] are two of the most popular traditional languages for the Ethereum [47] blockchain. They are based on JavaScript and Python respectively enabling a familiar programming experience. However, unlike session types in Nomos, neither of the languages provides a static system for enforcing contract protocols and tracking assets. Domain-specific languages have also been designed for other blockchain systems. Typecoin [15] generalizes Bitcoin to solve the peer-to-peer commitment problem where transaction deals in types rather than numbers. Contrary to Nomos' linear type system, Typecoin employs affine logic that admits weakening which ensures that assets are not duplicated but can be deleted. Michelson [2] is a stack-based language for the Tezos blockchain that features high-level data types and static type checking. Rholang [1] is a behaviorally typed, concurrent programming language, with a focus on message-passing and formally modeled by the ρ-calculus, a reflective, higher-order extension of the π-calculus. Closer to Nomos, Obsidian [14] employs typestate to express state transition during transaction execution. It also utilizes linear types to track and preserve assets. The Move language [10], modeled on Rust [30] provides the ability to define custom linear types to represent assets. However, none of these languages have a mechanism for static and automatic execution cost computation. On the UTXO side, Simplicity [34] is a combinator-based functional language without loops and recursion that comes equipped with a denotational semantics formalized in Coq. Its Turing incompleteness enables design of static analysis techniques for cost computation; however, it has no mechanism for expressing contract protocols. The Plutus language [13] draws from Haskell that distinguishes on-chain code from off-chain code compiling on-chain code using the Plutus compiler and off-chain code using GHC. Unlike Nomos, Plutus has no mechanism for static gas-cost computation.

To address resource usage, many languages come equipped with static cost analysis tools. The Scilla language [39] disallows loops and recursion inferring gas usage of a function as a polynomial of the size of its parameters and contract fields. In contrast, Nomos allows recursion and bounds are proven sound w.r.t. a gas semantics. Gastap [4] infers gas bounds on Solidity or EVM contracts in terms of size of the input parameters and contract state. Gasol [3] extends Gastap by offering a variety of cost models. It further detects under-optimized storage patterns and automatic optimization of such patterns. Nomos differs from these tools in its goal of computing a trusted exact gas bound which can be verified in linear time and eliminating dynamic gas metering.

8 Conclusion

This paper presented the implementation of the Nomos language with a focus towards integration with a blockchain system. The domain-specific features guaranteed by Nomos include *(i)* enforcement of contract protocols via session types, *(ii)* asset tracking via a linear type system, and *(iii)* automatic inference of execution cost bounds via resource-aware types. The article described each component of the Nomos toolchain supplemented with our efforts towards enhancing ease of programming and efficiency. As part of future work, we plan to perform an extensive evaluation of Nomos by implementing standard smart contracts and an in-depth comparison of Nomos with other state-of-the-art contract languages. We also plan to design an efficient compiler from Nomos to an executable low-level bytecode language such as EVM or Move bytecode to integrate with an existing blockchain system. We would also like to identify how Nomos can assist in implementing and analyzing off-chain transactions.

References

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

2 The Michelson Language. https://tezos.gitlab.io/009/michelson.html, August 2018. Accessed: 2018-11-04.

3 Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–125. Springer International Publishing, 2020.

4 Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes – Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts Using Static Resource Analysis. In *Verification and Evaluation of Computer and Communication Systems*, pages 63–78. Springer International Publishing, 2019.

5 Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the tezos blockchain. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 1–10, 2019. doi:10.1109/HPCS48598.2019.9188227.

6 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3190508.3190538.

7 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017*, pages 164–186, 2017.

8 Stephanie Balzer and Frank Pfenning. Manifest Sharing with Session Types. *Proceedings ACM Programming Languages*, ICFP, August 2017.

9 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain, 2019. URL: https://developers.diem.com/main/docs/state-machine-replication-paper.

10 Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon, and Yoni Zohar. Resources: A Safe Language Abstraction for Money, 2020. arXiv:2004.05106.

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

12 Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated Resource Analysis with Coq Proof Objects. In *Computer Aided Verification*, pages 64–85. Springer International Publishing, 2017.

13 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTXO Model. In *Financial Cryptography and Data Security*, pages 525–539. Springer International Publishing, 2020.

14 Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Trans. Program. Lang. Syst.*, 42(3), November 2020.

15 Karl Crary and Michael J. Sullivan. Peer-to-Peer Affine Commitment Using Bitcoin. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 479–488. Association for Computing Machinery, 2015.

16 Chris Dannen. *Introducing Ethereum and Solidity*, volume 318. Springer, 2017.

17 A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. Resource-Aware Session Types for Digital Contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 111–126. IEEE Computer Society, 2021.

18 Ankush Das, Jan Hoffmann, and Frank Pfenning. Work Analysis with Resource-Aware Session Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 305–314. Association for Computing Machinery, 2018.

19 Ankush Das and Shaz Qadeer. Exact and Linear-Time Gas-Cost Analysis. In *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event*, pages 333–356. Springer, 2020.

20 Ankush Das, Ishani Santurkar, and Stephen McIntosh. Nomos Implementation. `https://github.com/ankushdas/Nomos`, 2019. Accessed: 2019-11-11.

21 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

22 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 357–370. Association for Computing Machinery, 2011.

23 Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 359–373. Association for Computing Machinery, 2017.

24 Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 185–197. Association for Computing Machinery, 2003.

25 Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, pages 509–523. Springer Berlin Heidelberg, 1993.

26 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer Berlin Heidelberg, 1998.

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

28 Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 223–236. Association for Computing Machinery, 2010.

29 Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. Vyper: A Security Comparison with Solidity Based on Common Vulnerabilities, 2020. `arXiv:2003.07435`.

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

31 R. Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, 2003. `doi:10.1147/rd.471.0057`.

32 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269. Association for Computing Machinery, 2016.

33 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts, 2019. `arXiv:1907.03890`.

34 Russell O'Connor. Simplicity: A New Language for Blockchains, 2017. `arXiv:1711.03028`.

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

36 Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22:1–44, 2000.

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

38 Raine Revere. Solgraph. `https://github.com/raineorshine/solgraph`, 2019. Accessed: 2019-11-11.

39 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer Smart Contract Programming with Scilla. *Proceedings ACM Programming Languages*, OOPSLA, 2019.

40 Shanti Escalante-De Mattei. $22 B. Spent on NFTs in 2021: Market for Burgeoning Medium Rapidly Expanded, Report Says. `https://www.artnews.com/art-news/market/2021-nft-sales-report-1234613782/`, 2021. [Online; accessed 16-February-2022].

41 David Siegel. Understanding the dao hack for journalists. `https://medium.com/@pullnews/understanding-the-dao-hack-for-journalists-2312dd43e993`, June 2016.

42 Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, page 9–16. Association for Computing Machinery, 2018.

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

44 Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82. Association for Computing Machinery, 2018.

45 Philip Wadler. Propositions as Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 273–286. Association for Computing Machinery, 2012.

46 Wikipedia contributors. Non-fungible token — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Non-fungible_token&oldid=1072142954`, 2022. [Online; accessed 16-February-2022].

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