Anoma Research Topics | TECHNICAL REPORT

# Anoma Resource Machine Specification

**Yulia Khalniyazova**[a] **and Christopher Goes**[a]

[a]Heliax AG

* *E-Mail: {yulia, cwgoes}@heliax.dev*

## Abstract

The article explores the Anoma Resource Machine (ARM) within the Anoma protocol, providing a comprehensive understanding of its role in facilitating state updates based on user preferences. Drawing parallels with the Ethereum Virtual Machine, the ARM introduces a flexible transaction model, diverging from traditional account and UTXO models. Key properties such as atomic state transitions, information flow control, account abstraction, and an intent-centric architecture contribute to the ARM's robustness and versatility. Inspired by the Zcash protocol, the ARM leverages commitment accumulators to ensure transaction privacy. The article outlines essential building blocks, computable components, and requirements for constructing the ARM, highlighting its unique approach to resource-based state management.

**Keywords:** Anoma Resource Machine ; resource model ; virtual machine ; transaction privacy ;

(Received Nov 10, 2023; Version June 25, 2024)

## Contents

# 1. Introduction

In the Anoma protocol, users submit preferences about the system state and the system continuously updates its state based on those preferences. The Anoma Resource Machine (ARM) is the part of the Anoma protocol that defines and enforces the rules for valid state updates that satisfy users' preferences. The new proposed state is then agreed on by the consensus participants. In that sense the role of the Anoma Resource Machine in the Anoma protocol is similar to the role of the Ethereum Virtual Machine in the Ethereum protocol.

The atomic unit of the ARM state is called a **resource**. Resources are immutable, they can be created once and consumed once, which indicates that the system state has been updated. The resources that were created but not consumed yet make the current state of the system.

The ARM transaction model is neither the account nor UTXO model. Unlike the Bitcoin UTXO model, which sees UTXOs as currency units and is limited in expressivity, the resource model is generalised and provides flexibility — resource logics — programmable predicates associated with each resource — can be defined in a way to construct applications that operate in any desired transaction model, including the account and UTXO models. For example, a token

operating in the account model would be represented by a single resource containing a map $user : balance$ (unlike the UTXO model, where the token would be represented by a collection of resources of the token type, each of which would correspond to a portion of the token total supply and belong to some user owning this portion). Only one resource of that kind can exist at a time. When users want to perform a transfer, they consume the old balance table resource and produce a new balance table resource.

The Anoma Resource Machine has the following properties:

- *Atomic state transitions of unspecified complexity* — the number of resources created and consumed in every atomic state transition is not bounded by the system.
- *Information flow control* — the users of the system can decide how much of the information about their state to reveal and to whom. From the resource machine perspective, states with different visibility settings are treated equally (e.g., there is no difference between transparent — visible to anyone — and shielded — visible only to the parties holding the viewing keys — resources), but the amount of information revealed about the states differs. It is realised with the help of *shielded execution*, in which the state transition is only visible to the parties involved.
- *Account abstraction* — each resource is controlled by a *resource logic* — a custom predicate that encodes constraints on valid state transitions for that kind of resource and determines when a resource can be created or consumed. A valid state transition requires a resource logic validity proof for every resource created or consumed in the proposed state transition.
- *Intent-centric architecture* — the ARM provides means to express intents and ensures their correct and complete fulfilment and settlement.

The design of the Anoma Resource Machine was significantly inspired by the Zcash protocol [HBHW23].

The rest of the document contains the definitions of the ARM building blocks and the necessary and sufficient requirements to build the Anoma Resource Machine.

## 2. Preliminaries

### 2.1. Notation

For a function $h$, we denote the output finite field of $h$ as $\mathbb{F}_h$. If a function $h$ is used to derive a component $x$, we refer to the function as $h_x$, and the corresponding to $h$ finite field is denoted as $\mathbb{F}_{h_x}$, or, for simplicity, $\mathbb{F}_x$.

## 3. Resource

A *resource* is a composite structure $R = (l, label, q, v, eph, nonce, npk, rseed) : Resource$ where:

- $Resource = \mathbb{F}_l \times \mathbb{F}_{label} \times \mathbb{F}_Q \times \mathbb{F}_v \times \mathbb{F}_b \times \mathbb{F}_{nonce} \times \mathbb{F}_{npk} \times \mathbb{F}_{rseed}$
- $l : \mathbb{F}_l$ is a succinct representation of the predicate associated with the resource (resource logic)
- $label : \mathbb{F}_{label}$ specifies the fungibility domain for the resource. Resources within the same fungibility domain are seen as equivalent kinds of different quantities. Resources from different fungibility domains are seen and treated as distinct asset kinds. This distinction comes into play in the balance check described later.
- $q : \mathbb{F}_Q$ is a number representing the quantity of the resource
- $v : \mathbb{F}_v$ is the fungible data associated with the resource. It contains extra information but does not affect the resource's fungibility
- $eph : \mathbb{F}_b$ is a flag that reflects the resource's ephemerality. Ephemeral resources do not get checked for existence when being consumed
- $nonce : \mathbb{F}_{nonce}$ guarantees the uniqueness of the resource computable components
- $npk : \mathbb{F}_{npk}$ is a nullifier public key. Corresponds to the nullifier key $nk$ used to derive the resource nullifier (nullifiers are further described in 3.1.2)
- $rseed : \mathbb{F}_{rseed}$: randomness seed used to derive whatever randomness needed

To distinguish between the resource data structure consisting of the resource components and a resource as a unit of state identified by just one (or some) of the resource computed fields, we sometimes refer to the former as a resource plaintext.

### 3.1. Computable Components

Resource computable components are the components that are derivable from the resource components, other computed components, and possibly some secret data by applying a function from class $H$.

#### 3.1.1. Resource Commitment

Information flow control property implies working with flexible privacy requirements, varying from transparent contexts, where almost everything is publicly known, to contexts with stronger privacy guarantees, where as little information as possible is revealed.

From the resource model perspective, stronger privacy guarantees require operating on resources that are not publicly known in a publicly verifiable way. Therefore, proving the resource's existence has to be done without revealing the resource's plaintext.

One way to achieve this would be to publish a **commitment** to the resource plaintext. For a resource $r$, the resource commitment is computed as $r.cm = h_{cm}(r)$. Resource commitment has binding and hiding properties, meaning that the commitment is tied to the created resource but does not reveal information about the resource beyond the fact of creation. From the moment the resource is created, and until the moment it is consumed, the resource is a part of the system's state.

**Remark 1.** The resource commitment is also used as the resource's address $r.addr$ in the content-addressed storage. Consumption of the resource does not necessarily affect the resource's status in the storage (e.g., it doesn't get deleted).

All resource commitments are stored in an append-only data structure called a **commitment accumulator** *CMacc*. Every time a resource is created, its commitment is added to the commitment accumulator. The resource commitment accumulator *CMacc* is external to the resource machine, but the resource machine can read from it. A commitment accumulator is a cryptographic accumulator [ÖMBS21] that allows to prove membership for elements accumulated in it, provided a witness and the accumulated value.

Each time a commitment is added to the *CMacc*, the accumulator and all witnesses of the already accumulated commitments are updated. For a commitment

that existed in the accumulator before a new one was added, both the old witness and the new witness (with the corresponding accumulated value parameter) can be used to prove membership. However, the older the witness (and, consequently, the accumulator) that is used in the proof, the more information about the resource it reveals (an older accumulator gives more concrete boundaries on the resource's creation time). For that reason, it is recommended to use fresher parameters when proving membership.

The commitment accumulator *Acc* must support the following functionality:

- $ADD(acc, cm)$ adds an element to the accumulator, returning the witness used to prove membership.
- $WITNESS(acc, cm)$ for a given element, returns the witness used to prove membership if the element is present, otherwise returns nothing.
- $VERIFY(cm, w, val)$ verifies the membership proof for an element $cm$ with a membership witness $w$ for the accumulator value $val$.
- $VALUE(acc)$ returns the accumulator value.

Currently, the commitment accumulator is assumed to be a Merkle tree $CMtree$ of depth $depth_{CMtree}$, where the leaves contain the resource commitments and the intermediate nodes' values are computed using a hash function $h_{CMtree}$.

**Remark 2.** The hash function $h_{CMtree}$ used to compute the nodes of the $CMtree$ Merkle tree is not necessarily the same as the function used to compute commitments stored in the tree $h_{cm}$.

For a Merkle tree, the witness is the path to the resource commitment, and the tree root represents the accumulated value. To support the systems with stronger privacy requirements, the witness for such a proof must be a private input (4) when proving membership.

### 3.1.2. Resource Nullifier

A resource nullifier is a computed field, the publishing of which consumes the associated with the nullifier resource. For a resource $r$, the nullifier is computed from the resource's plaintext and a key called a nullifier key: $r.nf = h_{nf}(nk, r)$. A resource can be consumed only once. Nullifiers of consumed resources are stored in a public add-only structure called the resource nullifier set ($NFset$). This structure is external to the resource machine, but the resource machine can read from it.

Every time a resource is consumed, it has to be checked that the resource existed before (the resource's commitment is in the $CMtree$) and has not been consumed yet (the resource's nullifier is not in the $NFset$).

The nullifier set must support the following functionality:

- $WRITE(nf)$ adds an element to the nullifier set.
- $EXISTS(nf)$ checks if the element is present in the set, returning a boolean.

#### 3.1.3. Resource Kind

For a resource $r$, its kind is computed as $r.kind = h_{kind}(r.l, r.label)$.

#### 3.1.4. Resource Delta

Resource deltas are used to reason about the total quantities of different kinds of resources in transactions. For a resource $r$, its delta is computed as $r.\Delta = h_\Delta(r.kind, r.q)$.

The function used to derive $r.\Delta$ must have the following properties:

- For resources of the same kind $kind$, $h_\Delta$ should be *additively homomorphic*: $r_1.\Delta + r_2.\Delta = h_\Delta(kind, r_1.q + r_2.q)$
- For resources of different kinds, $h_\Delta$ has to be *kind-distinct*: if there exists $kind$ and $q$ s.t. $h_\Delta(r_1.kind, r_1.q) + h_\Delta(r_2.kind, r_2.q) = h_\Delta(kind, q)$, it is computationally infeasible to compute $kind$ and $q$.

**Remark 3.** An example of a function that satisfies these properties is the Pedersen commitment scheme: it is additively homomorphic, and its kind-distinctness property comes from the discrete logarithm assumption.

### 3.2. Non-linear resources

Non-linear resource is a resource that can be consumed multiple times, as opposed to a linear resource, that can only be consumed exactly once. Such resources could be useful to hold external data (e.g., the current gas price) that can be read multiple times. However, having native non-linear resources introduces some challenges as some basic assumptions about resources (e.g., nullifier uniqueness) wouldn't hold any more. At the same time, the resource structure might be unnecessary and excessive for storing such data. For these reasons, **the ARM doesn't support native non-linear resources.**

Without having non-linear resources, such functionality can be achieved from storing the data intended to be read separately and passing it to resource logics as arbitrary input. The authenticity of the provided data has to be verified first, and then it can be used by the resource logic as a valid source of information.

## 4. Proving system

A proving system allows proving statements about resources, which is required to create or consume a resource. Depending on the security requirements, a proving system might be instantiated, for example, by a signature scheme, a zk-SNARK, or a trivial transparent system where the properties are proven by openly verifying the properties of published data.

To support the intended spectrum of privacy requirements, varying from the strongest (where the relationship between the published parameters does not allow an observer to infer any kind of meaningful information about the state transition) to the weakest, where no privacy is required, we divide the proving system inputs into public (instance) and private (witness). The inputs that could potentially reveal the connection between components or other kinds of sensitive information are usually considered private, and the components that have to be and can be safely published regardless of the privacy guarantees of the system would be public inputs. Note that in the context of a transparent only system, this distinction is not meaningful because all inputs are public in such a system.

We define a set of structures required to define a proving system $PS$ as follows:

- Proof $\pi : PS.Proof$
- Instance $x : PS.Instance$ is the public input used to produce a proof.
- Witness $w : PS.Witness$ is the private input used to produce a proof.
- Proving key $pk : PS.ProvingKey$ contains the secret data required to produce a proof for a pair $(x, w)$.
- Verifying key $vk : PS.VerifyingKey$ contains the data required, along with the witness $x$, to verify a proof $\pi$.

A **proof record** carries the components required to verify a proof. It is defined as a composite structure $PR = (\pi, x, vk) : ProofRecord$, where:

- $ProofRecord = PS.VerifyingKey \times PS.Instance \times PS.Proof$
- $vk : PS.VerifyingKey$
- $x : PS.Instance$
- $\pi : PS.Proof$ is the proof of the desired statement.

A proving system $PS$ consists of a pair of algorithms, $(Prove, Verify)$:

- $Prove(pk, x, w) : PS.ProvingKey \times PS.Instance \times PS.Witness \longrightarrow PS.Proof$
- $Verify(pr) : PS.ProofRecord \longrightarrow \mathbb{F}_b$

A proving system used to produce the ARM proofs should have the following properties (as defined in [Tha23]):

- *Completeness.* This property states that any true statement should have a convincing proof of its validity.
- *Soundness.* This property states that no false statement should have a convincing proof.
- Proving systems used to provide privacy should additionally be *zero-knowledge*, meaning that the produced proofs reveal no information other than their own validity.

A proof $\pi$ for which $Verify(pr) = 1$ is considered valid.

The party responsible for creating proofs is also responsible for providing the input to the proving system. Public inputs are required to verify the proof and must be available to any party that verifies the proof; private inputs do not have to be available and can be stored locally by the proof creator. The same rule applies to custom (inputs not specified by the ARM) public and private inputs.

## 5. Roles and requirements

Table 1 contains a list of resource-related roles. In the Anoma protocol, the role of the resource creator will often be taken by a solver, which creates additional security requirements compared to the case when protocol users solve their own intents. Because of that, extra measures are required to ensure reliable distribution of the information about the created resource to the resource receiver.

### 5.1. Reliable resource plaintext distribution

In the case of in-band distribution of created resources in contexts with higher security requirements, the resource creator is responsible for encrypting the resource plaintext. Verifiable encryption must be used to ensure the correctness of the encrypted data: the encrypted plaintext must be proven to correspond to the resource plaintext, which is passed as a private input.

| Role | Description |
|---|---|
| Authorizer | approves the resource consumption on the application level. The resource logic encodes the mechanism that connects the authorizer's external identity (public key) to the decision-making process |
| Annuler | knows the data required to nullify a resource |
| Creator | creates the resource and shares the data with the receiver |
| Owner | can both authorize and annul a resource |
| Sender | owns the resources that were consumed to create the created resource |
| Receiver | owns the created resource |

**Table 1.** Resource-related roles.

### 5.2. Reliable nullifier key distribution

Knowing the resource's nullifier reveals information about when the resource is consumed, as the nullifier will be published when it happens, which might be undesirable in the contexts with higher security requirements. For that reason, it is advised to keep the number of parties who can compute the resource's nullifier as low as possible in such contexts.

In particular, the resource creator should not be able to compute the resource nullifier, and as the nullifier key allows to compute the resource's nullifier, it shouldn't be known to the resource creator. At the same time, the resource plaintext must contain some information about the nullifier key. One way to fulfil both requirements is, instead of sharing the nullifier key itself with the resource creator, to share some parameter derived from the nullifier key, but that does not allow computing the nullifier key or any meaningful information about it. This parameter is called a nullifier public key and is computed as $npk = h_{npk}(nk)$.

**Remark 4.** Note that these concerns are not meaningful in the contexts with lower security requirements.

## 6. Transaction

A transaction is a composite structure $TX = (rts, cms, nfs, \Pi, \Delta, extra, \Phi)$, where:

- $rts \subseteq \mathbb{F}_{rt}$ is a set of roots of $CMtree$
- $cms \subseteq \mathbb{F}_{cm}$ is a set of created resources' commitments.
- $nfs \subseteq \mathbb{F}_{nf}$ is a set of consumed resources' nullifiers.
- $\Pi : \{\pi : ProofRecord\}$ is a set of proof records.
- $\Delta_{tx} : \mathbb{F}_{\Delta}$ is computed from $\Delta$ parameters of created and consumed resources. It represents the total delta change induced by the transaction.

- $extra : \{(k, (d, deletion\_criterion)) : k \in \mathbb{F}_{key}, d \subseteq \mathbb{F}_d\}$ contains extra information requested by the logics of created and consumed resources. The deletion criterion field is described in 7.4.4.
- $\Phi : PREF$ where $PREF = TX \longrightarrow [0, 1]$ is a preference function that takes a transaction as input and outputs a normalised value in the interval $[0, 1]$ that reflects the users' satisfaction with the produced transaction. For example, a user who wants to receive at least $q = 5$ of resource of kind A for a fixed amount of resource of kind B might set the preference function to implement a linear function that returns 0 at $q = 5$ and returns 1 at $q = q_{max} = |\mathbb{F}_q| - 1$.
- $IFCPpredicate : TX \longrightarrow ExternalIdentity \longrightarrow \mathbb{F}_2$ is a predicate that specifies the transaction visibility.

## 6.1. Information flow control

The transaction visibility specified by the *IFCpredicate* describes what parties are and are not allowed to process the transaction. In the current version it is assumed that every node is following the policy and enforcing the conditions specified by the predicate.

### 6.1.1. Predicate options

In principle, the information flow predicate can be arbitrary as long as it satisfies the defined signature, but for now we define a set of allowed options to instantiate the IFC predicate as $BasePredicate : BaseData -> IFCPredicate$, where *BaseData* can be:

- *AllowAny* - always returns 1 - $AllowOnly(Set\ ExternalIdentity)$ - returns 1 for the specified set of identities - $RequireShielded(Set\ Hash)$ - returns 1 if the transaction doesn't contain the specified set of hashes in its fields - $And(Set\ Predicate)$ - returns 1 when all the specified predicates (instantiated by one of the base predicates) are satisfied - $Or(Set\ Predicate)$ - returns 1 when at least one of the specified predicates (instantiated by one of the base predicates) is satisfied

## 6.2. Transaction balance change

$\Delta_{tx}$ of a transaction is computed from the delta parameters of the resources (3.1.4) consumed and created in the transaction. It represents the total quantity change per resource kind induced by the transaction which is also referred to as *transaction balance*.

From the homomorphic properties of $h_\Delta$, for the resources of the same kind *kind*: $\sum_j h_\Delta(kind, r_{i_j}.q) - \sum_j h_\Delta(kind, r_{o_j}.q) = \sum_j r_{i_j}.\Delta - \sum_j r_{o_j}.\Delta = h_\Delta(kind, q_{kind})$.

The kind-distinctness property of $h_\Delta$ allows computing $\Delta_{tx} = \sum_j r_{i_j}.\Delta - \sum_j r_{o_j}.\Delta$ adding resources of all kinds together without the need to explicitly distinguish between the resource kinds: $\sum_j r_{i_j}.\Delta - \sum_j r_{o_j}.\Delta = \sum_j h_\Delta(kind_j, q_{kind_j})$

**Remark 5.** Note that only transactions with $\Delta_{tx}$ committing to 0 (or any other balancing value specified by the system) can be executed and settled.

### 6.3. Proofs

Each transaction refers to a set of resources to be consumed and a set of resources to be created. Creation and consumption of a resource requires a set of proofs that attest to the correctness of the proposed state transition. There are three proof types associated with a transaction:

- Resource logic proof $\pi_{RL}$. For each resource consumed or created in a transaction, it is required to provide a proof that the logic of the resource evaluates to 1 given the input parameters that describe the state transition (the exact resource machine instantiation defines the exact set of parameters).
- A delta proof (balance proof) $\pi_\Delta$ makes sure that $\Delta_{tx}$ is correctly derived from $\Delta$ parameters of the resources created and consumed in the transaction and commits to the expected publicly known value, called a *balancing value*.
- A resource machine compliance proof $\pi_{compl}$ is required to ensure that the provided transaction is well-formed. The resource machine compliance proof must check that each consumed resource was consumed strictly after it was created, that the resource commitments and nullifiers are derived according to the commitment and nullifier derivation rules, and that the resource logics of created and consumed resources are satisfied.

**Remark 6.** It must also be checked that the created resource was created exactly once and the consumed resource was consumed exactly once. These checks can be performed separately, with read access to the $CMtree$ and $NFset$.

**Remark 7.** Every proof is created with a proving system $PS$ and has the type $PS.Proof$. The proving system might differ for different proof types.

**Remark 8.** For privacy-preserving contexts, all proving systems in use should support data privacy, and the proving system used to create resource logic proofs should provide function privacy in addition to data privacy: provided proofs

of two different resource logics, an observer should not be able to tell which proof corresponds to which logic. It is a stronger requirement compared to data privacy, which implies that an observer does not know the private input used to produce the proof.

### 6.4. Composition

Having two transactions $tx_1$ and $tx_2$, their composition $tx_1 \circ tx_2$ is defined as a transaction $tx$, where:

- $rts_{tx} = rts_1 \cup rts_2$
- $cms_{tx} = cms_1 \sqcup cms_2$
- $nfs_{tx} = nfs_1 \sqcup nfs_2$
- Proofs:
  - delta proof: $\Pi^{\Delta}_{tx} = AGG(\Pi^{\Delta}_1, \Pi^{\Delta}_2)$, where $AGG$ is an aggregation function s.t. for $bv_1$ being the balancing value of the first delta proof, $bv_2$ being the balancing value of the second delta proof, and $bv_{tx}$ being the balancing value of the composed delta proof, it satisfies $bv_{tx} = bv_1 + bv_2$. The aggregation function takes two delta proofs as input and outputs a delta proof.
  - resource logic proofs: $\Pi^{RL}_{tx} = \Pi^{RL}_1 \sqcup \Pi^{RL}_2$
  - compliance proofs: $\Pi^{compl}_{tx} = \Pi^{compl}_1 \sqcup \Pi^{compl}_2$
- $\Delta_{tx} = \Delta_1 + \Delta_2$
- $extra_{tx} = extra_1 \cup extra_2$
- $\Phi_{tx} = G(\Phi_1, \Phi_2)$, where $G : PREF \times PREF \longrightarrow PREF$, and $G$ is a preference function composition function
- $IFCPredicate_{tx} = IFCPredicate^{I}_1 FCPredicate_2$

**Remark 9.** Composing sets with disjoint union operator $\sqcup$, it has to be checked that those sets do not have any elements in common. Otherwise, the transactions cannot be composed.

### 6.5. Validity

A transaction is considered *valid* if the following statements hold:

- $rts$ contains valid $CMtree$ roots that are correct inputs for the membership proofs
- input resources have valid resource logic proofs and the compliance proofs associated with them
- output resources have valid resource logic proofs and the compliance proofs associated with them
- $\Delta$ is computed correctly, and its opening is equal to the balancing value for that transaction

## 7. Resource Machine

A resource machine is a deterministic stateless machine that creates, composes, and verifies transaction functions.

It has read-only access to the external global state, which includes the content-addressed storage system (which in particular stores resources), global commitment accumulator, and the global nullifier set, and can produce writes to the external local state that will later be applied to the system state.

The resource machine has two layers: the outer layer, the resource machine shell, that creates and processes **transaction functions**, and the inner layer, the resource machine core, that creates and processes *transactions*.

We assume the shell is trivial in this version of the ARM: it only evaluates the transaction function without any verification steps. The result is a transaction that is then passed to the core. The distribution of responsibilities between the shell and the core is expected to change.

To support the shell layer, the resource machine must have the functionality to produce, compose, and evaluate transaction functions. Assuming the shell is trivial in the current version of the specification, the following description of the resource machine functionality describes the functionality of the resource machine core.

A transaction function is defined as $TransactionFunction : () \rightarrow Transaction$.

See section 8.1 for a description of what data the transaction function can read during execution.

### 7.0.1. Post- and pre-ordering execution

*Pre-ordering execution* implies partial evaluation of the transaction function. In practice pre-ordering execution happens before the transactions are ordered by the ordering component external to the ARM.

*Post-ordering execution* implies full evaluation of the transaction function. As the name suggests, post-ordering execution happens after the ordering component external to the ARM completed the ordering of transaction functions.

## 7.1. Create

Given a set of components required to produce a transaction, the create function produces a transaction data structure, which involves computing the nullifiers of the consumed resources, commitments of the created resources, transaction $\Delta$, and all the required proofs.

Assuming that the produced transaction induces a state change consuming resources $r_{i_1}, \cdots, r_{i_n}$ and creating resources $r_{o_1}, \cdots, r_{o_m}$, the inputs and outputs of the create function are defined as follows.

Input:

- a set of $CMtree$ roots $\{rt_{i_k}, k \leq n\}$
- a set of resources $\{r_{i_1}, ..., r_{i_n}, r_{o_1}, ..., r_{o_m}\}$
- a set of nullifier secret keys $\{nk_{i_1}, ..., nk_{i_n}\}$
- extra data $extra$
- preference function $\Phi$
- custom inputs required for resource logic proofs

Output: a transaction $tx = (rtscms, nfs, \Pi, \Delta_{tx}, extra, \Phi)$, where:

- $rts = \{rt_{i_1}, .., rt_{i_n}\}$
- $nfs = \{nf_{i_k} = h_{nf}(nk_{i_l}, r_{i_l}), k = 1..n\}$
- $cms = \{cm_{o_1} = h_{cm}(r_{o_l}), k = 1..m\}$
- $\Pi = \{\pi_{\Delta_{tx}}, \pi_{compl_1}, ..., \pi_{compl_c}, \pi_{i_1}, ..., \pi_{i_n}, \pi_{o_1}, ..., \pi_{o_m}\}$, where $1 \leq c \leq m + n$
- $\Delta_{tx} = \sum_k \Delta_{i_k} - \sum_l \Delta_{o_l}$
- $extra$
- $\Phi$

| Name | Structure | Key Type | Value Type |
|---|---|---|---|
| Commitment accumulator (node) | Cryptographic accumulator | timestamp | $\mathbb{F}$ |
| Commitment accumulator (leaf) | - | (timestamp, $\mathbb{F}$) | $\mathbb{F}$ |
| Nullifier set | Set | $\mathbb{F}$ | $\mathbb{F}$ |
| Hierarchical index | Chained Hash sets | Tree path | $\mathbb{F}$ |
| Data blob storage | Key-value store with del. criteria | $\mathbb{F}$ | (var. length byte array, del. criteria) |

**Figure 1.** Stored Data Format.

### 7.2. Compose

Taking two transactions $tx_1$ and $tx_2$ as input, produces a new transaction $tx = tx_1 \circ tx_2$ according to the transaction composition rules (6.4).

### 7.3. Verify

Taking a transaction as input, verifies its validity according to the transaction validity rules (6.5). If the transaction is valid, the resource machine outputs a state update. Otherwise, the output is empty.

### 7.4. Stored data format

The ARM state that needs to be stored includes resource plaintexts, the commitment accumulator and the nullifier set. Figure 1 defines the format of that data assumed by the ARM.

#### 7.4.1. *CMtree*

Each commitment tree node has a timestamp associated with it, such that a lower depth (closer to the root) tree node corresponds to a less specified timestamp: a parent node timestamp is a prefix of the child node timestamp, and only the leaves of the tree have fully specified timestamps (i.e. they are only prefixes of themselves). For a commitment tree of depth $d$, a timestamp for a commitment $cm$ would look like $t_{cm} = t_1 : t_2 : .. : t_d$, with the parent node corresponding to it having a timestamp $t_1 : t_2 : .. : *$. The timestamps are used as keys for the key-value store. For the tree leaves, $< cm, t_{cm} >$ pairs are used as keys. Merkle paths to resource commitments can be computed from the hierarchy of the timestamps.

#### 7.4.2. *NFset*

Nullifiers are used as keys in the key-value store. In future versions, a more complex structure that supports efficient non-membership proofs might be used for storing the nullifier set.

### 7.4.3. Hierarchical index

The hierarchical index is organised as a tree where the leaves refer to the resources, and the intermediate nodes refer to resource *subkinds* that form a hierarchy. The label of a resource $r$ stored in the hierarchical index tree is interpreted as an array of *sublabels*: $r.label = [label_1, label_2, label_3, ...]$, and the i-th subkind is computed as $r.subkind_i = H_{kind}(r.l, r.label_i)$.

**Remark 10.** In the current version, only the subkinds derived from the same resource logic can be organized in the same hierarchical index path.

The interface of the tree enables efficient querying of all children of a specific path and verifying that the returned children are the requested nodes. Permissions to add data to the hierarchical index are enforced by the resource logics and do not require additional checks.

### 7.4.4. Data blob storage

Data blob storage stores data without preserving any specific structure. The data is represented as a variable length byte array and comes with a deletion criterion that determines for how long the data will be stored. The deletion criterion, in principle, is an arbitrary predicate, which in practice currently is assumed to be instantiated by one of the following options:

- delete after *block*
- delete after *timestamp*
- delete after *sig* over *data*
- delete after either predicate $p_1$ or $p_2$ is true; the predicates are instantiated by options from this list
- store forever

## 7.5. ARMs as intent machines

Together with $(CMtree, NFset)$, the Anoma Resource Machine forms an instantiation of the intent machine, where the state $S = (CMtree, NFset)$, a batch $B = Transaction$, and the transaction verification function of the resource machine corresponds to the state transition function of the intent machine [HR24]. To formally satisfy the intent machine's signature, the resource machine's verify function may return the processed transaction along with the new state.

# 8. Program Formats

## 8.1. Transaction function

The system used to represent and interpret transaction functions must have a deterministic computation model; each operation should have a fixed cost of space and time (for total cost computation). To support content addressing, it must have memory and support memory operations (specifically *read*, *write*, *allocate*).

The system must support the following I/O operations:

- $READ\_STORAGE(address : \mathbb{F}_{cm})$: read the global content-addressed storage at the specified address and return the value stored at the address. If the value is not found, the operation should return an error. Storage not accessible to the machine should be treated as non-existent.
- $DATA\_BY\_INDEX(index\_function)$: read data from the storage (either resources or arbitrary data kept in the storage requested by the transaction function) at the execution time by the specified index function. If the index function output is invalid or uncomputable, or the data cannot be located, the operation should return an error. Typically, the index functions allowed will be very restricted, e.g. an index function returning current unspent resources of a particular kind.

## 8.2. Gas model

To compute and bound the total cost of computation, the transaction function system must support a gas model. Each evaluation would have a gas limit $g_{limit}$, and the evaluation would start with $g_{count} = 0$. Evaluating an operation, the system would add the cost of the operation to the counter $g_{count}$ and compare it to $g_{limit}$. When making recursive calls, $g_{count}$ is incremented before the recursion occurs. If the value of $g_{count}$ is greater than $g_{limit}$, the execution is terminated with an error message indicating that the gas limit has been surpassed.

## 8.3. Resource Logic

A resource logic is a predicate associated with a resource that checks that the provided data satisfies a set of constraints. It does not require I/O communication and is represented by or can feasibly be turned into a zk-SNARK circuit if desired to support shielded execution.

Each resource logic has a set of public and private input values as in 4. Resource logics are customizable on both implementation of the ARM (different

instantiations might have different requirements for all resource logics compatible with this instantiation) and resource logic creation level (each instantiation supports arbitrary resource logics as long as they satisfy the requirements). A concrete implementation of the ARM can specify more mandatory inputs and checks (e.g., if the resources are distributed in-band, resource logics have to check that the distributed encrypted value indeed encrypts the resources created/consumed in the transaction), but the option of custom inputs and constraints must be supported to enable different resource logic instances existing on the application level.

The proving system used to interpret resource logics must provide the following properties:

- Verifiability. It must be possible to produce and verify a proof of type $PS.Proof$ that given a certain set of inputs, the resource logic output is true value.
- The system $PS$ used to interpret resource logics must be zero-knowledge- and function-privacy-friendly to support privacy-preserving contexts.

Resource logics take as input a subset of resources created and consumed in the transaction:

**Resource Logic Public Inputs**:

- $nfs \subseteq nfs_{tx}$
- $cms \subseteq cms_{tx}$
- $tag : \mathbb{F}_{tag}$ — identifies the resources being checked
- $extra \subseteq tx.extra$

**Resource Logic Private Inputs**:

- input resources corresponding to the elements of $nfs$
- output resource corresponding to the elements of $cms$
- custom inputs

**Resource Logic Constraints**:

- for each output resource, check that the corresponding $cm$ value is derived according to the rules specified by the resource machine instance

- for each input resource, check that the corresponding $nf$ value is derived according to the rules specified by the resource machine instantiation
- custom checks

### 8.4. Preference Function

Preference functions do not require I/O communication or have any other special requirements. They are stateless. It may make sense to interpret them using the same system used for transaction functions for simplicity.

### 8.5. Nockma

Nockma (Nock-Anoma) is a modification of the Nock4K specification [Urb] and a Nock standard library altered and extended for use with Anoma. Nockma is designed to support the transaction function interpreter requirements (Section 8.1), namely, global storage read and deterministic bounded computation costs.

Nockma is parameterized over a specific finite field $\mathbb{F}_h$ and function $h$. The function $h$ takes an arbitrary noun (a data unit in Nockma) as input and returns an element of $\mathbb{F}_h$. This function is used for verifying reads from content-addressed storage.

A *scry* (inspired by Urbit's concept of the same name) is a read-only request to Anoma's global content-addressed namespace or indices computed over values stored in this namespace. Scrying is used to read data that would be inefficient to store in the noun, to read indices whose value might only be known at execution time, or to read data that may not be accessible to the author of the noun.

Scrying comes in two types: "direct" or "index". A direct lookup simply returns the value stored at the address (integrity can be checked using $h$), or an error if a value is not found. An index lookup uses the value stored at the address as an index function and returns the results of computing that index or an error if the index is not found, invalid, or uncomputable. The lookup type is the only parameter required apart from the content address (which must be an element of $\mathbb{F}_h$).

Typically, the index functions allowed will be very restricted, e.g. current unspent resources of a particular kind. Gas costs of scrying will depend on the index function and the size of the results returned.

Scrying may be used to avoid unnecessary, redundant transmission of common Nockma subexpressions, such as the standard library.

Nockma is a combinator interpreter defined as a set of reduction rules over nouns. A noun is an atom or a cell, where an atom is a natural number and a

cell is an ordered pair of nouns.

The Nockma reduction rules as presented in Figure 8 are applied from top to bottom, the first rule from the top matches. Variables match any noun. As in regular Nock4K, a formula that reduces to itself is an infinite loop, which we define as a crash ("bottom" in formal logic). A real interpreter can detect this crash and produce an out-of-band value instead.

The only difference between Nockma and Nock4K reduction rules is that instruction 12 is defined for scrying.

Used with the resource machine, Nockma should return a set of modifications to the state transition expressed by the input transaction:

- a set of resources to additionally create (resource plaintexts)
- a set of resources to additionally consume (addresses)
- a set of storage writes (in the format specified in Section 7.4)

The Nockma standard library must include the following functions.
For a finite field $\mathbb{F}_n$ of order $n$, it should support:

- additive identity of type $\mathbb{F}_n$
- addition operation $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_n$
- additive inversion $\mathbb{F}_n \to \mathbb{F}_n$
- multiplicative identity of type $\mathbb{F}_n$
- multiplication operation $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_n$
- multiplicative inversion $\mathbb{F}_n \to \mathbb{F}_n$
- equality operation $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_2$
- comparison operation based on canonical ordering $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_2$

For a ring $Z_n$ of unsigned integers mod $n$, it should support:

- additive identity of type $Z_n$
- addition operation $Z_n \times Z_n \to Z_n \times \mathbb{F}_2$ (with overflow indicator)
- subtraction operation $Z_n \times Z_n \to Z_n \times \mathbb{F}_2$ (with overflow indicator)

- multiplicative identity of type $Z_n$
- multiplication operation $Z_n \times Z_n \to Z_n \times \mathbb{F}_2$ (with overflow indicator)
- division operation (floor division) $Z_n \times Z_n \to Z_n$
- equality $Z_n \times Z_n \to \mathbb{F}_2$
- comparison $Z_n \times Z_n \to \mathbb{F}_2$

Additionally, it should provide a parametrized conversion function $conv_{i,j,k,l}$, where

- $i$ is a flag that defines the input type: $i = 0$ corresponds to a finite field, $i = 1$ corresponds to a ring of unsigned integers
- $j$ is the input structure order
- $k$ is a flag that defines the output type: $k = 0$ corresponds to a finite field, $k = 1$ corresponds to a ring of unsigned integers
- $l$ is the output structure order

If the order of the input structure is bigger than the order of the output structure ($j > l$), the conversion function would return a flag (of type $\mathbb{F}_2$) indicating if overflow happened in addition to the converted value.

The conversion function must use canonical ordering and respect the inversion laws.

## 9. Applications

The ARM applications are characterised by a set of resource logics and a set of transaction functions.

$Application = (ApplicationLogic, ApplicationInterface)$, where

1. $ApplicationLogic \subseteq \mathbb{F}_l$ is a set of resource logics.
2. $ApplicationInterface = \{t : TransactionFunction\}$ is a set of transaction functions.

As any abstract action can be represented as a transaction consuming and creating resources of certain kinds (or a transaction function that evaluates to such a transaction), the transaction functions associated with the application

represent the set of actions that the application can provide to its users. Each transaction function would require a subset of the application resource logics to approve the transaction in order to realise the desired action. The transaction function evaluated with the exact resources to be created and consumed forms a transaction.

The resources that are bound with the application resource logics are said to belong to the application and constitute the application state. When the application does not have any resources that were created but not consumed yet, the application only exists virtually but not tangibly.

The abstraction of an application is virtual - applications are not deployed or tracked in any sort of global registry, and the ARM is unaware of the existence of applications.

We define $AppKinds \subseteq \mathbb{F}_{kind}$ as a union of all resource kinds that are involved in the transaction functions that comprise the application interface.

### 9.1. Composition

Applications are composable. The composition of two (or more) applications would be a composition of the corresponding logics and interfaces.

$App_12 = App_1 \circ App_2$:

1. $AppLogic_{12} = AppLogic_1 \cup AppLogic_2$
2. $AppInterface_{12} = AppInterface_1 \cup AppInterface_2$
3. $AppKinds_{12} = AppKinds_1 \cup AppKinds_2$

In this type of composition the order in which the applications are composed doesn't matter.

### 9.2. Application extension

Application extension is a way to generate a new application starting from an existing one by enhancing the application logic and the application interface with operations on more resource kinds. The new application is dependent on the initial one, meaning that the new application logic includes constraints involving the first application resource kinds, and the new interface requires the presence of resources of the first application kinds.

### 9.3. Distributed application state synchronisation

In [She24], a controller is defined as a component that orders transactions. The resource machine is designed to work in both single-controller and multi-controller

environments, such as Anoma. In the context of multi-controller environments, each resource contains information about its current controller, can only be consumed on its controller, and can be transferred from one controller to another, meaning that a new controller becomes responsible for the correct resource consumption. Transferring a resource can be done by consuming a resource on the old controller and creating a similar resource on the new controller [She24].

Applications do not have to exist within the bounds of a single controller, and can maintain a single virtual state while the application resources being distributed among multiple controllers, which forms a distributed application state. To make sure such a distributed state correctly represents the application state, state synchronisation between multiple controllers is required.

#### 9.3.1. Controller state synchronisation

Each controller would have their own commitment tree associated with it. Treated as subtrees of a larger Merkle tree, the controller commitment trees comprise a global commitment tree, where the leaves are the roots of the controller trees.

## 10. Transaction Examples

This section describes how resources and transactions get created and composed in various use cases to provide some intuition for how the described resource model works.

In the examples below, the superscript parameter indicates the party associated with the resource, e.g., a resource $R^{Alice}$ is associated with Alice. In the case of proofs, it indicates the party created the proof.

### 10.1. Two Party Exchange

Let us consider an example of a two-party exchange. One party's intent is precise, and the other party's intent implies options (the requested NFT's exact properties may also vary). We assume that the resources parties initially consume were already created at some point in the past.

**Step 1:** specify intents

- **Alice's intent:** exchange either a resource $R_{1A}$ of kind A and quantity 1 or a resource $R_{2B}$ of kind B and quantity 2 for a blue dolphin NFT resource $R_{NFT}$. The intent is contained in the resource logic of a resource $R_I$ of kind I.

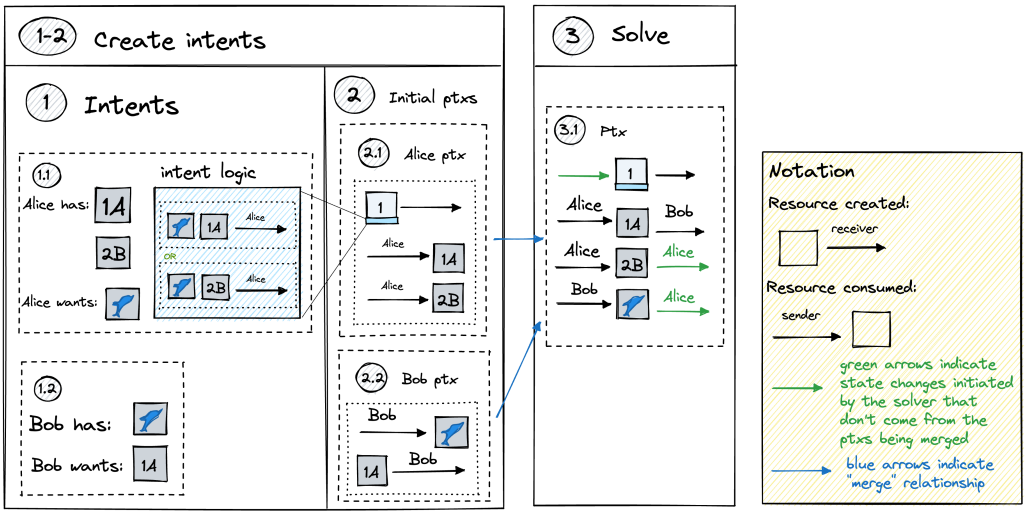


**Figure 2.** Two-party exchange

- **Bob's intent:** exchange a blue dolphin NFT resource for a resource of kind A and quantity 1. The intent is referred to in a transaction.

**Remark 11.** For simplicity, in the examples in this section, the set of compliance proofs for initial transactions (the transactions that were not composed of other transactions) is assumed to contain all the necessary compliance proofs, but the proofs themselves are not specified. Additionally, the delta proof aggregation function can take an arbitrary number of arguments. $AGG(X, Y, Z)$ in practice would be implemented as $AGG(AGG(X, Y), Z)$, similarly defined for any number of proofs.

**Step 2:** create initial transactions
Alice creates a transaction $TX^A$ creating $R^A_I$, and consuming $R^A_{1A}$ and $R^A_{2B}$:

- $rts = \{rt_{R^A_{1A}}, rt_{R^A_{2B}}\}$
- $cms = \{cm_{R^A_I}\}$
- $nfs = \{nf_{R^A_{1A}}, nf_{R^A_{2B}}\}$
- Proofs:
  - $\Pi^A_\Delta$

  - $\Pi^A_{compl}$
  - $\Pi^A_{rl} = \{\pi^A_A, \pi^A_B, \pi^A_I\}$
- $\Delta \mapsto \{I : 1, A : -1, B : -2\}$. For simplicity, represent $\Delta$ as a dictionary
- $extra = extra^A$
- $\Phi = \Phi^A$

Bob creates a transaction $TX^B$ creating $R^B_{1A}$ and consuming $R^B_{NFT}$:

- $rts = \{rt_{R^B_{NFT}}\}$
- $cms = \{cm_{R^B_{1A}}\}$
- $nfs = \{nf_{R^B_{NFT}}\}$
- Proofs:
  - $\Pi^B_\Delta$
  - $\Pi^B_{compl}$
  - $\Pi^B_{rl} = \{\pi^B_A, \pi^B_{NFT}\}$
- $\Delta \mapsto \{NFT : -1, A : 1\}$
- $extra = extra^B$
- $\Phi = \Phi^B$

**Step 3:** solve
A solver $S$, having $TX^A$ and $TX^B$, creates a transaction $TX^S$:

- $rts = \{rt_{R^A_I}\}$
- $cms = \{cm_{R^A_{2B}}, cm_{R^A_{NFT}}\}$
- $nfs = \{nf_{R^A_I}\}$
- Proofs:

  - $\Pi^S_\Delta$
  - $\Pi^S_{compl}$
  - $\Pi^S_{rl} = \{\pi^S_B, \pi^S_{NFT}, \pi^S_I\}$
- $\Delta \mapsto \{NFT : 1, B : 2, I : -1\}$
- $extra = extra^S$
- $\Phi = \Phi^S$

and composes all three transactions together, producing a balanced transaction $TX$:

- $rts = \{rt_{R^A_I}, rt_{R^A_{2B}}, rt_{R^A_{1A}}, rt_{R^B_{NFT}}\}$
- $cms = cms^A \sqcup cms^B \sqcup cms^S = \{cm_{R^A_I}, cm_{R^A_{2B}}, cm_{R^A_{NFT}}, cm_{R^B_{1A}}\}$
- $nfs = nfs^A \sqcup nfs^B \sqcup nfs^S = \{nf_{R^A_I}, nf_{R^A_{2B}}, nf_{R^A_{1A}}, nf_{R^B_{NFT}}\}$
- Proofs:
  - $\Pi_\Delta = AGG(\Pi^A_\Delta, \Pi^B_\Delta, \Pi^S_\Delta)$
  - $\Pi_{compl} = \Pi^A_{compl} \sqcup \Pi^B_{compl} \sqcup \Pi^S_{compl}$
  - $\Pi_{rl} = \Pi^A_{rl} \sqcup \Pi^B_{rl} \sqcup \Pi^S_{rl}$
- $\Delta \mapsto \{A : 0, B : 0, I : 0, NFT : 0\}$
- $extra = extra^A \cup extra^B \cup extra^S$
- $\Phi = G(\Phi^A, \Phi^B, \Phi^S)$

In practice, the step of creation of the transaction $TX_{S_1}$ can be merged with the composing step, but we separate the steps for clarity.

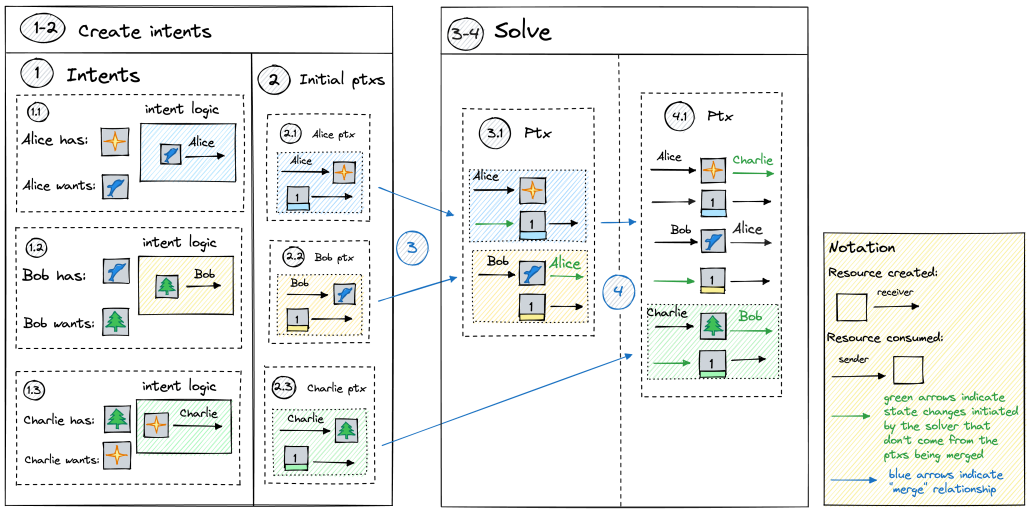


**Figure 3.** Three-party exchange cycle

## 10.2. Three-party NFT exchange cycle

Another example is a three-party exchange cycle. Each party uses ephemeral resource logics to express their intents.

**Step 1:** specify intents

- **Alice's intent:** Alice wants to exchange her star NFT resource $R^A_{star}$ for a blue dolphin NFT resource $R_{dolphin}$
- **Bob's intent:** Bob wants to exchange his blue dolphin NFT $R^B_{dolphin}$ for a tree NFT resource $R_{tree}$
- **Charlie's intent:** Charlie wants to exchange his tree NFT $R^C_{tree}$ for a star NFT resource $R_{star}$

**Step 2:** create initial transactions
Alice's initial transaction:

- $rts = \{rt_{R^A_{star}}\}$
- $cms = \{cm_{R^A_{IA}}\}$
- $nfs = \{nf_{R^A_{star}}\}$
- Proofs:
  - $\Pi^A_\Delta$

  - $\Pi^A_{compl}$
  - $\Pi^A_{rl} = \{\pi^A_{star}, \pi^A_I\}$
- $\Delta \mapsto \{I^A : 1, star : -1, \}$ – for simplicity, represent $\Delta$ as a dictionary
- $extra = extra^A$
- $\Phi = \Phi^A$

Bob's initial transaction:

- $rts = \{rt_{R^B_{dolphin}}\}$
- $cms = \{cm_{R^B_{I^B}}\}$
- $nfs = \{nf_{R^B_{doplhin}}\}$
- Proofs:
  - $\Pi^B_\Delta$
  - $\Pi^B_{compl}$
  - $\Pi^B_{rl} = \{\pi^B_{dolphin}, \pi^B_I\}$
- $\Delta \mapsto \{I^B : 1, dolphin : -1\}$
- $extra = extra^B$
- $\Phi = \Phi^B$

Charlie's initial transaction:

- $rts = \{rt_{R^C_{tree}}\}$
- $cms = \{cm_{R^C_{I^C}}\}$
- $nfs = \{nf_{R^C_{tree}}\}$
- Proofs:
  - $\Pi^C_\Delta$

  - $\Pi^C_{compl}$
  - $\Pi^C_{rl} = \{\pi^C_{tree}, \pi^C_I\}$
- $\Delta \mapsto \{I^C : 1, tree : -1, \}$
- $extra = extra^C$
- $\Phi = \Phi^C$

**Step 3:** solve

A solver $S_1$, seeing $TX^A$ and $TX^B$, creates a transaction $TX^{S_1}$ (on the diagram: $TX_{3.1}$, green arrows):

- $rts = \{rt_{R^A_{IA}}\}$
- $cms = \{cm_{R^A_{dolphin}}\}$
- $nfs = \{nf_{R^A_{IA}}\}$
- Proofs:
  - $\Pi^{S_1}_{\Delta}$
  - $\Pi^{S_1}_{compl}$
  - $\Pi^{S_1}_{rl} = \{\pi^{S_1}_{dolphin}, \pi^{S_1}_{I_A}\}$
- $\Delta \mapsto \{dolphin : 1, I^A : -1\}$
- $extra = extra^{S_1}$
- $\Phi = \Phi^{S_1}$

and composes all three transactions together, producing a transaction $TX_{3.1}$:

- $rts = \{rt_{R^A_{star}}, rt_{R^B_{dolphin}}, rt_{R^A_{IA}}\}$
- $cms = cms^A \sqcup cms^B \sqcup cms^{S_1} = \{cm_{R^A_{IA}}, cm_{R^B_{IB}}, cm_{R^A_{dolphin}}\}$
- $nfs = nfs^A \sqcup nfs^B \sqcup nfs^{S_1} = \{nf_{R^A_{star}}, nf_{R^B_{dolphin}}, nf_{R^A_{IA}}\}$

- Proofs:
  - $\Pi_{\Delta}^{3.1} = AGG(\Pi_{\Delta}^{A}, \Pi_{\Delta}^{B}, \Pi_{\Delta}^{S_1})$
  - $\Pi_{compl}^{3.1} = \Pi_{compl}^{A} \sqcup \Pi_{compl}^{B} \sqcup \Pi_{compl}^{S_1}$
  - $\Pi_{rl}^{3.1} = \Pi_{rl}^{A} \sqcup \Pi_{rl}^{B} \sqcup \Pi_{rl}^{S_1}$
- $\Delta \mapsto \{I^A : 0, I^B : 1, star : -1, dolphin : 0\}$
- $extra = extra^A \cup extra^B \cup extra^{S_1}$
- $\Phi = G(\Phi^A, \Phi^B, \Phi^{S_1})$

**Step 4:** continue solving

Seeing $TX^C$ and $TX_{3.1}$, a solver $S_2$ creates a transaction $TX_{S_2}$ (on the diagram: $TX_{4.1}$, green arrows):

- $rts = \{rt_{R_{IC}^C}, rt_{R_{IB}^B}\}$
- $cms = \{cm_{R_{star}^C}, cm_{R_{tree}^B}\}$
- $nfs = \{nf_{R_{IC}^C}, nf_{R_{IB}^B}\}$
- Proofs:
  - $\Pi_{\Delta}^{S_2}$
  - $\Pi_{compl}^{S_2}$
  - $\Pi_{rl}^{S_2} = \{\pi_{star}^{S_2}, \pi_{I_C}^{S_2} \pi_{tree}^{S_2}, \pi_{I_B}^{S_2}\}$
- $\Delta \mapsto \{I^C : -1, I^B : -1, star : 1, tree : 1\}$
- $extra = extra^{S_2}$
- $\Phi = \Phi^{S_2}$

and composes all three into a balanced transaction $TX_{4.1}$:

- $rts = \{rt_{R_{star}^A}, rt_{R_{dolphin}^B}, rt_{R_{tree}^C}, rt_{R_{IA}^A}, rt_{R_{IB}^B}, rt_{R_{IC}^C}\}$

- $cms = cms^{TX^{3.1}} \sqcup cms^{S_2} = \{cm_{R^A_{dolphin}}, cm_{R^B_{tree}}, cm_{R^C_{star}}, cm_{R^A_{I^A}}, cm_{R^B_{I^B}}, cm_{R^C_{I^C}}\}$
- $nfs = nfs^{TX^{3.1}} \sqcup nfs^{S_2} = \{nf_{R^A_{star}}, nf_{R^B_{dolphin}}, nf_{R^C_{tree}}, nf_{R^A_{I^A}}, nf_{R^B_{I^B}}, nf_{R^C_{I^C}}\}$
- Proofs:
  - $\Pi^{4.1}_{\Delta} = AGG(\Pi^{3.1}_{\Delta}, \Pi^{C}_{\Delta}, \Pi^{S_2}_{\Delta})$
  - $\Pi^{4.1}_{compl} = \Pi^{C}_{compl} \sqcup \Pi^{3.1}_{compl} \sqcup \Pi^{S_2}_{compl}$
  - $\Pi^{4.1}_{rl} = \Pi^{C}_{rl} \sqcup \Pi^{3.1}_{rl} \sqcup \Pi^{S_2}_{rl}$
- $\Delta \mapsto \{I^A : 0, I^B : 0, I^C : 0, star : 0, dolphin : 0, tree : 0\}$
- $extra = extra^A \cup extra^B \cup extra^C \cup extra^{S_1} \cup extra^{S_2}$
- $\Phi = G(\Phi^A, \Phi^B, \Phi^{S_1}, \Phi^C, \Phi^{S_2})$

In practice, the step of creation of the transactions $TX_{S_1}$ and $TX_{S_2}$ can be merged with the composing step, but we separate the steps for clarity.

## 11. Application examples

This section provides some examples of applications that can be built with a resource model. The applications in this section are simple and do not reflect all the capabilities of the resource model but can be helpful to gain some intuition for how to build applications using this model. More application examples can be found on the Anoma Research Forum[1].

### 11.1. Token transfer with identity isolation

This example describes the design of a token application where a transfer is authorised by the token owner's signature isolated into a separate resource logic.

Identity isolation allows the user to bind the account resource to multiple applications and have a simple mechanism to update the key used for authorisation just by updating a single account resource instead of individually updating the key in each application that requires authorisation. The user may have multiple accounts bound to different sets of applications.

It is implemented with the help of three resource types:

[1] https://research.anoma.net/c/applications/36

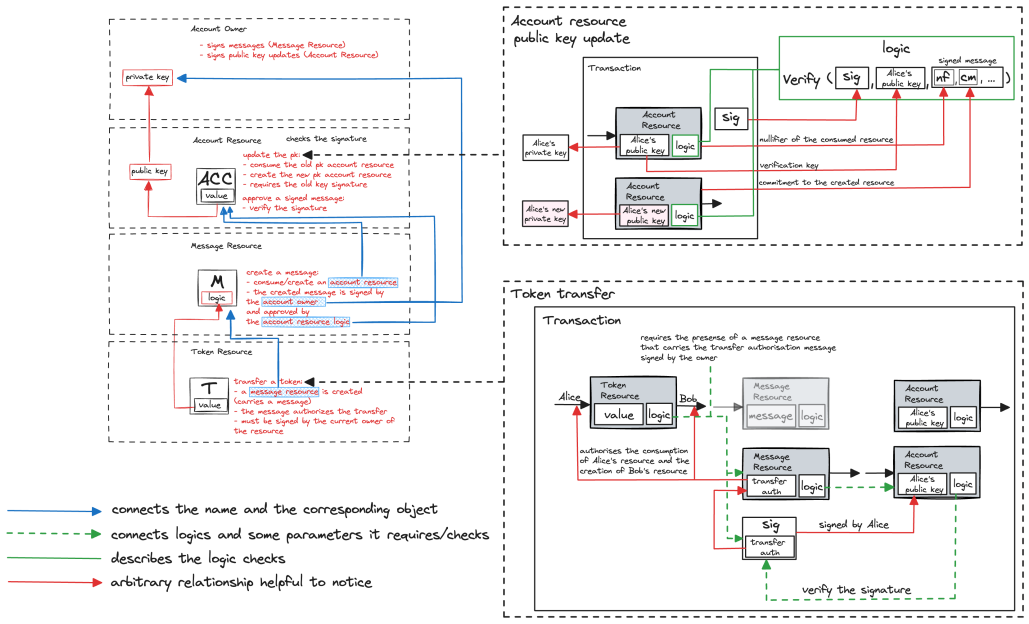


**Figure 4.** Identity isolation example diagram

1. *token resource* corresponds to the token being transferred,
2. *message resource* carries arbitrary information,
3. and *account resource* allows approving arbitrary data by signing it.

### 11.1.1. Account resource

An account resource stores a public key that corresponds to a user's private key (that is stored privately by the user) and enables public key authorisation mechanics: the user signs some data using their private key and publishes the signature; the corresponding public key, that is stored in the account resource, is used to publicly verify the signature.

Using such account resource mechanics allows to require a signature presence and its validity as a resource logic constraint, so that it becomes an explicit and a verifiable requirement for a valid transaction.

To verify the signature, the account resource is consumed, with the corresponding account resource logic verifying the signature, using the public key the resource stores.

The keypair associated with the account can also be updated, so that the new account resource contains a new public key. This action requires the authorization by the old keypair.

### 11.1.2. Message resource

Message resource is a resource that carries an arbitrary message. Creating a message requires the user to sign the message with their private key and consume an account resource to verify the signature, as described above.

### 11.1.3. Token resource

Token resource represents a token. To transfer a token, a message authorising the transfer must be created.

**Remark 12.** In practice, transferring a token means consuming the current token resource(s) and creating equivalent token resource(s) but referring to a different owner, but abstractly it can be seen as transferring.

## 11.2. Counter application

This example describes the mechanics of counters associated with a specific controller, implemented in a resource model.

### 11.2.1. CounterId resource

CounterId resource represents a counter tied to a specific controller that is used to initialise other counters. In a sense, it can be seen as a counter of counters. There are two actions associated with a CounterId resource:

1. **Init:** A new CounterId resource can be initiated to 0.
2. **Update:** An existing CounterId resource can be updated from value $n$ to value $n + 1$.

Both operations require a relevant controller to approve the transaction.

### 11.2.2. Counter resource

Counter resource represents a simple counter. There are two actions associated with a Counter resource:

1. **Init:** A new counter can be initialised from the current *CounterId* resource (the *CounterId* value goes to the counter resource label). When a new counter is created, the *CounterId* resource's value is incremented by 1 to ensure there are no counters that have the same label.
2. **Update:** An existing *Counter* resource can be updated from value $n$ to value $n + 1$.

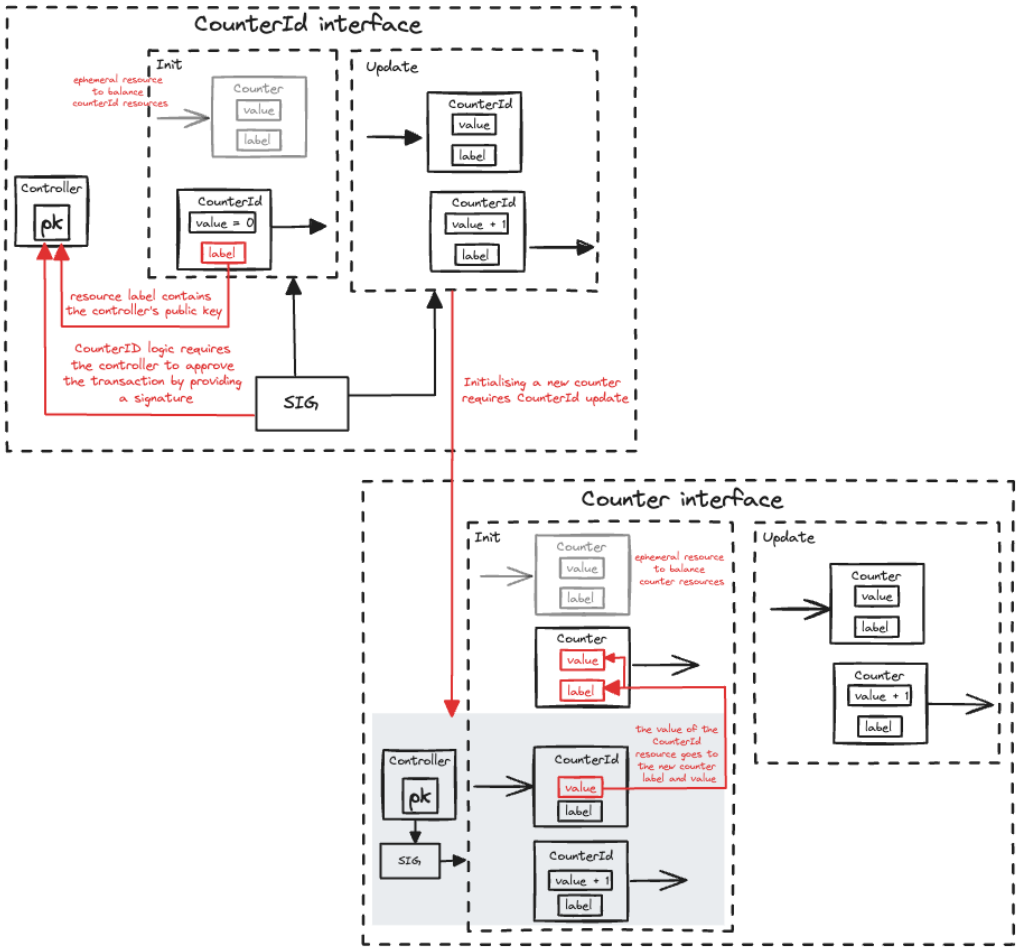


**Figure 5.** Counter application example diagram

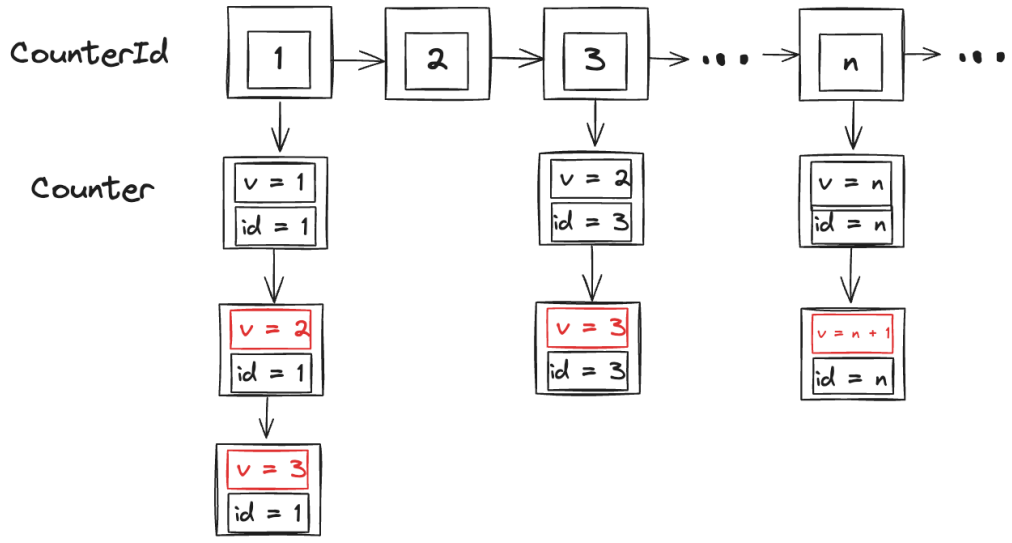


**Figure 6.** CounterId resource is a counter of counters

In this simple version, updating the counter doesn't require any special permissions but the counter resource logic can be modified to be more complex with more restricted counter update rules.

### 11.3. Proof-of-Stake

This section describes an example of a simplified version of the Proof-of-Stake application. The goal of the PoS protocol is to assign voting power in the BFT consensus algorithm: users delegate their tokens to validators to signal that they trust the chosen validator with making decisions. The voting power of each validator is determined from the amount of tokens delegated to them. The delegated tokens are locked for a period of time, so that if the validator misbehaves, this behaviour could be tracked and reacted on by burning a part of the delegated tokens. This application can be a useful building block for other applications or, more generally, for the contexts that require a mechanism for decision-making.

**Remark 13.** The advantage of having infractions as resources as opposed to using proofs of misbehaviour directly is that in order to be created such resources have to follow the specified format which is guaranteed by the resource logic.

1. **Delegate**: create a new bond given that a user transferred their tokens to the pool.

| Resource kind | Description | Create | Consume |
|---|---|---|---|
| Token | Governance token used to distribute the voting power. | Generic token logic | Generic token logic |
| Pool | An account that owns bonded tokens. Does not necessarily have to be a special resource type, can be just a dedicated type of the owner. | Generic account logic | Generic account logic |
| Bond | Re presents a bonded token delegated to the desired validator. Always owned by the pool. $B_{amount}$ contains the delegated quantity, $B_{validator}$ refers to the validator the token is delegated to, $B_{owner}$ refers to the delegator. Looking at all the bonds allows seeing how much tokens are delegated to each validator and determine their voting power. | Requires the user to send their assets to the pool | Can be consumed to create a withdrawal. |
| Withdrawal | Represents the asset being undelegated. $W_{amount}$, $W_{validator}$, $W_{owner}$ defined the same way as for bonds. Additionall, it contains $W_{unlock}$ field that defines when the resource can be consumed. | Created from consuming a bond | Can be consumed when the portion of assets remaining after the slashing is performed are sent from the pool to the user and strictly after $W_{unlock}$ |
| Infraction | Represents a proof of misbehaviour of a certain validator $I_{validator}$ and defines the infraction rate $I_{rate}$ for slashing. $I_{timestamp}$ defines when the misbehaviour was committed. | Created provided a proof of misbehaviour | Never consumed |
| Voting power | Contains the distribution of voting power among the validators | Can be created provided a proof of correct computation of voting power from the existing bonds. | Never consumed |

**Table 2.** Resource kinds involved in the PoS application

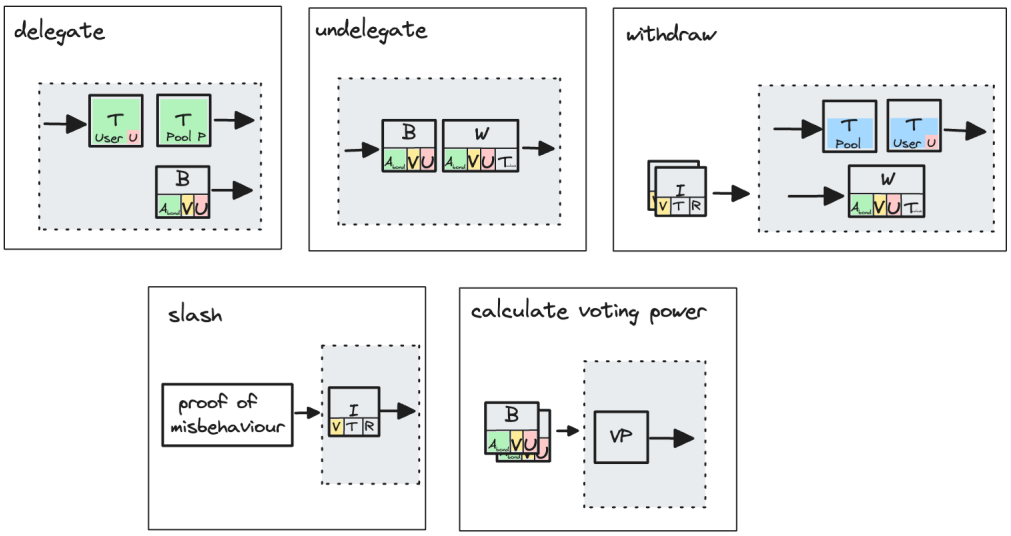


**Figure 7.** PoS application interface

2. **Undelegate**: consume a bond, create a withdrawal resource.

3. **Withdraw**: consume a withdrawal resource, transfer the token from the pool back to the user. To consume the correct amount to return, iterate over all infractions (read, not consume) for this validator created after the assets were delegated but before the withdrawal period initiated, i.e., only account for infractions created while the assets were delegated. The amount of the returned tokens is then calculated as: $\Pi 1 - I_{rate}$, where $I_{rate}$ is the infraction rate of the infraction resource $I$.

4. **Slash**: create an infraction resource provided a proof of misbehaviour.

5. **Calculate voting power**: iterate over all bonds (read, but not consume), computing the voting power of each validator.

## 12. Conclusion and Future directions

This report contains the necessary information to build a resource machine that has the desired properties, but there are more properties we might want and more questions worth investigating. One such question would be whether resource logics should be able to see all resources in a transaction. This would allow us to perform "for all" checks — for example, a resource logic might want

to enforce a non-inclusion of resources of a certain type in this transaction. However, enforcing such a feature is a non-trivial task, and it is not clear if it is as beneficial as it seems: for example, if there is a valid way to escape such checks (e.g., by wrapping a resource in another resource kind), it will not be helpful to have a mechanism for checking.

## References


HBHW23. Daira Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2023. (cit. on p. 4.)

HR24. Anthony Hart and D Reusche. Abstract Intent Machines. *Anoma Research Topics*, February 2024. (cit. on p. 18.)

ÖMBS21. Ilker Özçelik, Sai Medury, Justin T. Broaddus, and Anthony Skjellum. An overview of cryptographic accumulators. *CoRR*, abs/2103.04330, 2021. (cit. on p. 6.)

She24. Issac Sheff. Cross-Chain Integrity with Controller Labels and Endorsement. *Anoma Research Topics*, Apr 2024. (cit. on pp. 24 and 25.)

Tha23. Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. 2023. (cit. on p. 10.)

Urb. Urbit. Nock definition. (cit. on p. 21.)

# A. The Nockma reduction rules

| Pattern | Reduces to |
|---|---|
| $nock(a)$ | $*a$ |
| $[a\ b\ c]$ | $[a\ [b\ c]]$ |
| $?[a\ b]$ | $0$ |
| $?a$ | $1$ |
| $+[a\ b]$ | $+[a\ b]$ |
| $+a$ | $1\ +\ a$ |
| $=[a\ a]$ | $0$ |
| $=[a\ b]$ | $1$ |
| $/[1\ a]$ | $a$ |
| $/[2\ a\ b]$ | $a$ |
| $/[3\ a\ b]$ | $b$ |
| $/[(a\ +\ a)\ b]$ | $/[2\ /[a\ b]]$ |
| $/[(a\ +\ a\ +\ 1)\ b]$ | $/[3\ /[a\ b]]$ |
| $/a$ | $/a$ |
| $\#[1\ a\ b]$ | $a$ |
| $\#[(a\ +\ a)\ b\ c]$ | $\#[a\ [b\ /[(a\ +\ a\ +\ 1)\ c]]\ c]$ |
| $\#[(a\ +\ a\ +\ 1)\ b\ c]$ | $\#[a\ [/[(a\ +\ a)\ c]\ b]\ c]$ |
| $\#a$ | $\#a$ |
| $*[a\ [b\ c]\ d]$ | $[*[a\ b\ c]\ *[a\ d]]$ |
| $*[a\ 0\ b]$ | $/[b\ a]$ |
| $*[a\ 1\ b]$ | $b$ |
| $*[a\ 2\ b\ c]$ | $*[*[a\ b]\ *[a\ c]]$ |
| $*[a\ 3\ b]$ | $?*[a\ b]$ |
| $*[a\ 4\ b]$ | $+*[a\ b]$ |
| $*[a\ 5\ b\ c]$ | $=[*[a\ b]\ *[a\ c]]$ |
| $*[a\ 6\ b\ c\ d]$ | $*[a\ *[[c\ d]\ 0\ *[[2\ 3]\ 0\ *[a\ 4\ 4\ b]]]]$ |
| $*[a\ 7\ b\ c]$ | $*[*[a\ b]\ c]$ |
| $*[a\ 8\ b\ c]$ | $*[[*[a\ b]\ a]\ c]$ |
| $*[a\ 9\ b\ c]$ | $*[*[a\ c]\ 2\ [0\ 1]\ 0\ b]$ |
| $*[a\ 10\ [b\ c]\ d]$ | $\#[b\ *[a\ c]\ *[a\ d]]$ |
| $*[a\ 11\ [b\ c]\ d]$ | $*[[*[a\ c]\ *[a\ d]]\ 0\ 3]$ |
| $*[a\ 11\ b\ c]$ | $*[a\ c]$ |
| $*[a\ 12\ b\ c\ d]$ | $result\ \leftarrow\ SCRY\ b\ c;\ *[a\ result\ d]$ |
| $*a$ | $*a$ |

**Figure 8.** Nockma reduction rules.