Anoma Research Topics | TECHNICAL REPORT

Intent-centric Applications for the Anoma Resource Machine

Michael A. Heuer[a] **and D Reusche**[a]

[a]Heliax AG

* *E-Mail: michael@heliax.dev*

Abstract

Anoma introduces a universal intent machine, allowing developers to write applications in terms of intents, which can be ordered, solved, and settled anywhere. This work illustrates how intent-centric applications can be built using the Juvix language in Anoma's resource model and in compliance with the Anoma resource machine. First, we detail the general architecture of applications and their relation to and dependencies on other components of the Anoma protocol. Second, we describe recurring design patterns and primitives related to resource creation and initialization, authorization, ownership, intents, and tokens in a broader context. Last and by employing these primitives, we present Kudos, an accounting primitive incorporating trust relationships between identities.

Keywords: Anoma Resource Machine ; Resource Model ; Anoma Applications ;

(Received: August 15, 2024; Version: August 26, 2024)

Contents

1. Introduction

An application is a computer program designed to carry out a specific task [Wik24]. As such, it provides an interface to read and write state on a machine according to predefined processes and presents a practical abstraction over direct interaction with the machine and the mechanisms it provides.

With the advent of distributed ledger technologies, decentralized applications grew in popularity and can be found in various sectors such as payments, finance, governance, identity, supply chain management, insurance, law, arts and social media, but are fragmented over an ever-increasing number of blockchains.

Generally, these blockchains are not interoperable. Consequently, application developers are increasingly faced with building additional infrastructure, such as indexers, ZK circuits and provers, and rely on bridges as intermediaries, compromising on decentralization and increasing complexity.

Application users are facing this increased complexity as well. Being predominantly written in imperative languages, applications require users to either understand increasingly complex execution traces, often crossing multiple

different protocols, or trust the processes, which poses usability and security problems. Moreover, most blockchains are fully transparent and provide no user privacy.

The Anoma protocol [GYB23] as a universal intent machine [HR24] addresses these problems of application developers and users by providing four affordances: a permissionless intent infrastructure, intent-level composability, information flow control, and heterogeneous trust[Goe24] which is achieved through three mechanisms: the resource machine [KG24] a heterogeneous trust node architecture [SWRM24, KS24, She24, HKS24], and languages for explicit service commitments (paper in progress).

As described in a recent forum post [Goe24], Anoma's unique affordances and mechanisms result in desirable application properties. Written in the declarative paradigm and resource model, users can universally express preferences and constraints over desired states without worrying about execution traces and specific state machine instantiations. These properties make applications portable and composable at the intent and intent-machine level, allowing them to move freely across concurrency and security domains without additional development work or protocol barriers. Related intents can be ordered, solved, and settled anywhere–on the Ethereum main chain, on Ethereum virtual machine (EVM) and non-EVM rollups, on Eigenlayer actively validated services (AVSs), on Cosmos chains, Solana, or any sufficiently programmable state machine. Accordingly, applications can treat the entire Anoma network as a virtualized state space–they do not need to be deployed separately to different chains, integrate different bridges, or pick any specific security model [Goe24]. Consequently, Anoma applications must be designed and written differently than conventional decentralized applications, which are predominantly written in imperative languages and must adhere to computation-ordering virtual machine (VM) designs.

In this report, we first detail how Anoma applications – their state and logic – are structured and managed and how they interface with other components in Anoma's architectural topology. In this context, we outline how application state and logic can persist and rely on different service providers . Second, we explain how to design applications for the Anoma resource machine (ARM) and show recurring patterns related to resource initialization, finalization, authorization, intents, and tokens in Juvix [Cza23, CPCMRC24a]. Lastly, these are used to build an initial version of Kudos, an accounting primitive incorporating trust relationships between identities.

This report focuses on the transparent case, where information is generally known by observing parties.

2. Application Architecture

Anoma, as a universal intent machine [HR24], transitions its state by executing transactions settling batches of intents, which are commitments to user preferences and constraints over the space of possible state transitions.
In this context, applications provide known objects, formats, and processes for the specific task they carry out, thus allowing users to send transactions and declare, match, and settle their intents by facilitating coordination and counterparty discovery around shared interests and topics. Token, messenger or crowdfunding applications, for example, encompass the logic and manage the state of resources associated with tokens, messages, and crowdfunding, who sees or has control over those resources and how they persist over time.

Well-designed applications can also be composed with each other. For example, a message can be sent to a token recipient with the transfer as part of a crowdfunding application in a single atomic transaction. In the following, we describe the architecture of intent-centric applications and how they interact with different components of the Anoma protocol.

Applications consist of a set of resources constituting the state and carrying the application resource logic being verified by the ARM and a user interface facilitating the generation of transaction objects consuming and creating said resources in accordance with their logic. For some applications, no counterparty is required so that users can unilaterally create balanced transaction objects constituting a valid state transition that can be executed immediately. For example, a token transfer to a known recipient does not require intent solving (unless the token logic explicitly requires approval by the receiving party).
Other application cases require counterparty discovery. Here, application users create unbalanced transaction objects that solvers then match by composing them with each other to form a balanced and valid transaction. This is where intents come into play. Intents can be encoded as resources allowing application users to encode the constraints and preferences for the state transition they desire. For example, a token swap requires at least one counterparty (and multiple ones in case of a ring trade) to produce a balanced, valid and, therefore, executable transaction. In addition to intent resources being part of the transaction, users can specify their overall satisfaction with the transaction object through a preference function attached to it (see Appendix A.2).

More details on resource- and transaction-related definitions can be found in Appendices A.1 and A.2 or the ARM specifications [see KG24, pp. 5 ff.].
In the following, we describe how application state and logic are organized in detail and how applications interact with different services provided by Anoma network operators.

```
type Resource :=
  mkResource {
    logic : Resource -> Transaction
    ↪  -> Bool;
    label : Nat;
    quantity : Nat;
    data : Nat;
    eph : Bool;
    nonce : Nat;
    npk : Nat;
    rseed : Nat
  };
```

Listing 1. The resource type definition taken from the `juvix-anoma-stdlib` library `v0.5.0`.

```
type Transaction :=
  mkTransaction {
    roots : List Nat;
    commitments : List Commitment;
    nullifiers : List Nullifier;
    proofs : List Proof;
    complianceProofs : List
    ↪  ComplianceProof;
    delta : Delta;
    extra : Nat;
    preference : Nat
  };
```

Listing 2. The transaction type definition taken from the `juvix-anoma-stdlib` library `v0.5.0`.

2.1. State

Application state is modelled as *resources* (see Listing 1) that are created and consumed adhering to Anoma's state transition function being processed and verified by the ARM. This state is commonly distributed over different nodes. Only two computable resource components (see Appendix A.2) must be stored and synchronized by all Anoma nodes participating in the network: Resource commitments and revealed nullifiers. The former and latter are obtained from the hash of the resource plaintext and the hash of the resource plaintext and associated nullifier key, respectively, and are stored in an accumulator data structure, currently a Merkle tree, and a set data structure, respectively.
All other application state is stored as content-addressed binary large objects (BLOBs), either in the local key-value storage of the node or that of a storage provider, both of which can be permanent or temporary with pre-defined deletion criteria, which we describe later in Section 2.4.1. This state includes resource plaintexts, extra data required by the application in the context of a transaction, resource logic function objects, and function objects constituting the application interface that we introduce in the following.

2.2. Logic

Application logic can be divided into three components in agreement with the Model View Controller (MVC) architecture for user interfaces. More precisely, it adheres to the less known Model View Intent (MVI) architecture, which is characterized by a strict, unidirectional dependency between the components represented by referentially transparent functions.

First, *projection functions* representing the *view* component that we denote as the *Application Read Interface*. Projection functions observe the application (or global) state and compute and aggregate data that the application interface

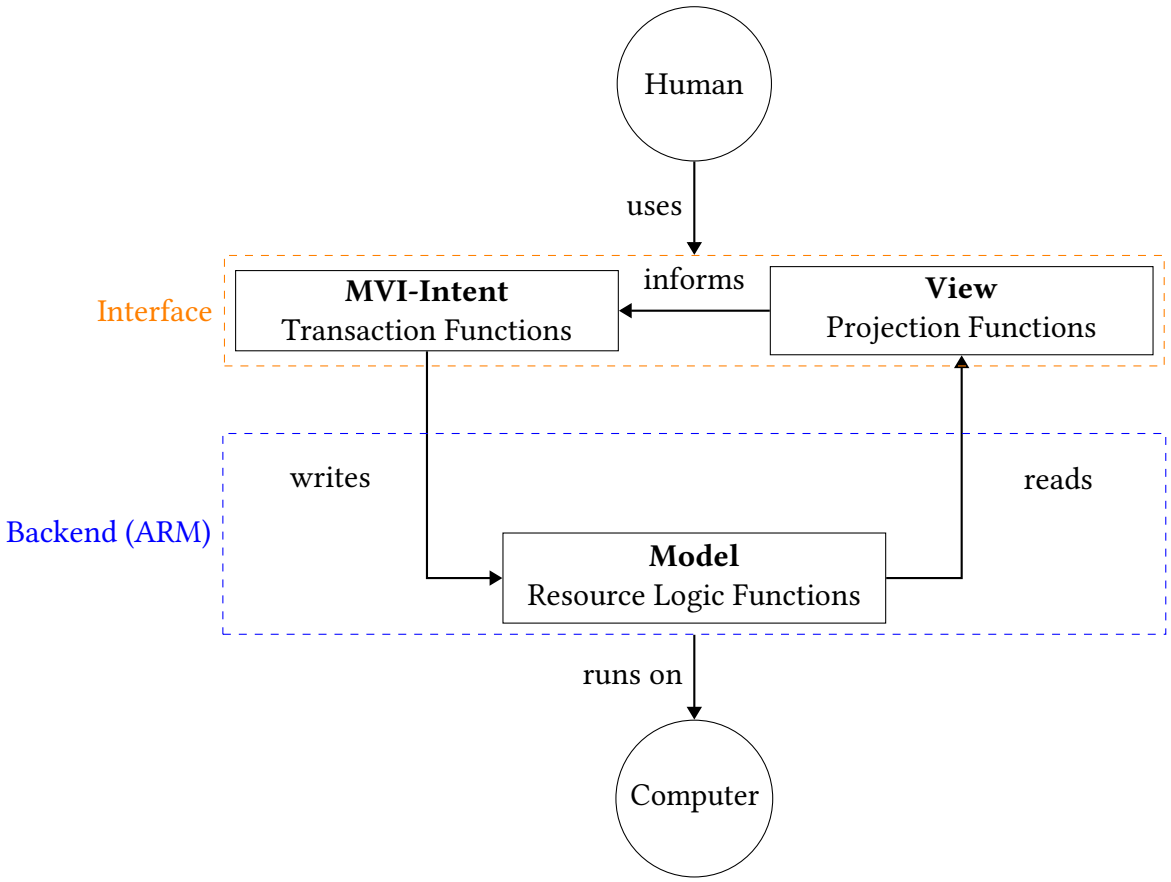

Figure 1. Schematic visualization of the three logic components of applications and their unidirectional dependency within the MVI architecture for user interfaces: Projection, transaction, and resource logic functions constituting the application read interface, write interface, and backend, respectively.

requires. Second, transaction functions representing the *intent* (or, more commonly, the *controller*) component, which we denote as the *Application Write Interface*. Transaction functions observe the application view and require additional user input data to construct balanced transactions to be executed by the ARM or unbalanced *transactions* to be matched by network peers, i.e., solvers. Note that the linguistic convergence on the term *intent* is slightly misleading here since transaction objects do not generally require an intent (and therefore counterparties) as they can often be settled unilaterally. Lastly, the *resource logic functions* representing the *model* component being associated with each individual resource, encoding the rules of their creation and consumption, and constituting the application backend. In the following, we describe all components in detail.

```
alwaysTrueLogic (self : Resource) (tx : Transaction) : Bool := true;
```

Listing 3. A resource logic function always returning `true`.

2.2.1. Resource Logic Functions

Resource logic functions (`logic : Resource -> Transaction -> Bool`[1]) are processed by the ARM as part of the validity checking and return a boolean value. For a transaction to be executable, all resource logic functions must be valid and return `true`. In the current Anoma node implementation, logic functions receive two input arguments.
First, the resource object (see Appendix A.1) the logic function is part of and that we will refer to as `self`.
Second, the transaction object (see Appendix A.2) constituting the context in which the self resource is created or consumed. In particular, it contains the set of commitments and nullifiers of other resources, extra data, as well as a *preference function* and *transaction balance value* Δ_{tx} being relevant in the context of solving.

Given these arguments, resource logic functions can declare predicates over the transaction consuming or creating them. These predicates can be related to the associated resource itself and the state it carries, but also to other resources and data being present in the context of the transaction, which we will discuss in more detail in Section 3. On one hand, this allows resource logics to declare precise rules for application state transitions, e.g., "The updated counter value resource must be equal to the old counter value incremented by one". On the other hand, this allows the declaration of user intents without specifying the exact execution trace, e.g., "I give 3 Alice Kudos for at least 2 Bob Kudos or 3 Carol Kudos.", thus allowing counterparties to match and settle the transaction. The simplest, valid resource logic function possible, the function always returning true, is shown in Listing 3.

Resource logic functions are stored separately from the resource objects in the processing node's key-value storage (see Appendix A.1) as content-addressed data BLOBs.[2] Accordingly, the same resource logic function can be associated with multiple resource objects.

2.2.2. Projection Functions

Projection functions constitute the *Application Read Interface* between the Anoma node client and the user (or an intermediary frontend), and as such, provide defined processes, formats, and objects to project data from the application state model. As an input argument, projection functions receive a set of

[1] The `logic` function type should be `Set Nullifier -> Set Commitment -> Tag -> Transaction -> Bool` according to the specs but this is currently not implemented.

[2] The Anoma node implementation currently deviates from the ARM specifications and stores the function object encoded as a `Nat` in the resource object.

```
doNothing : Transaction :=
  Transaction.mk@{
    roots := [];
    commitments := [];
    nullifiers := [];
    proofs := [];
    complianceProofs := [];
    delta := [];
    extra := 0;
    preference := 0
  };
```

Listing 4. A transaction function creating an empty transaction object.

resource plaintexts (`Set Resource`) and can have custom, developer-defined output arguments. An example projection function is shown in Listing 27 in Section 3.10.4 and calculates the total quantity of a set of resources. In general, these resources are obtained from a preceding data discovery and filtering process, before they are passed to the projection function.

Projection function objects are stored as content-addressed data BLOBs as well and loaded into the node's memory when required.

2.2.3. Transaction Functions

Transaction functions constitute the *Application Write Interface* between the user (or an intermediary frontend) and the Anoma node client, and, as such, handle the creation and consumption of resource objects and insert their commitments and nullifiers in the transaction object, populating the transaction extra data map (e.g., with message objects and signatures), and define the preference function. They can have custom, developer-defined input arguments but must return a `transaction` object[3] that can be unbalanced. Like resource logic function objects, transaction function objects are stored as content-addressed data BLOBs and loaded into the node's memory when required. A simple example transaction function creating an empty transaction is shown in Listing 4. More sophisticated transaction functions might initialize or finalize resources (see Listings 18 and 19), transfer, split, and merge resource objects (e.g., Listings 20 to 22), combine or select between lower-level transaction functions for convenience (e.g., Listing 23), call projection functions to obtain or check input values or handle errors that might occur.

[3] Alternatively, it can return an `Result Error (Pair Transaction (List Assignment))` type, which will be supported by the Anoma node client implementation in the future.

```
type Result E A :=
  | error E
  | ok A;
```

Listing 5. A type either resolving into an error of type `E` or the return argument of type `A`.

2.3. Error Handling

Transaction and projection functions can encounter errors preventing them from computing meaningful outputs, or resource logic functions can be invalid. In the case of projection functions, this can be caused by missing or unexpected BLOBs in Anoma's key-value storage, e.g., a non-existent lookup key. For transaction functions, this can be caused by wrong user input, e.g., if a list of resources is expected to be of the same kind but is not in practice, but also because of missing `extra` data, e.g., a missing message and signature. Lastly, resource logic functions can be invalid if constraints are unmet, e.g., if an expiration date has passed or the caller lacks a required role.

Instead of silently failing, the application should return detailed information about the error cause and context, allowing the user or developer to debug the underlying issue. This becomes even more important when applications are composed. Accordingly, on the interface side, transaction and projection functions can return a term of the `Result` type (see Listing 5), either resolving to an error or to the actual return argument. This is used, for example, in Listings 18 to 23. For resource logic functions, a logging functionality will be provided in a future version of the Anoma node implementation.

Instead of just printing errors, they can be handled more sophisticatedly. Typed errors, in particular, can contain additional information and be used in a resolution attempt. For example, if the BLOB is unavailable in the node's local storage, it can attempt to fetch it from network peers, which can take the role of dedicated storage service providers. This is described in the following sections.

2.4. Application Configurations

An application configuration aids fetching all required BLOBs associated with an application and specifies service provider commitments, e.g., to ensure the BLOBs are available. These BLOBs contain the resource logic, transaction, and projection function objects[4]. These can be stored in application repositories and be discoverable via the address of the resource logic, requested from counterparties transferring resources of a new application to a user for the first time, or shipped with the Anoma node distribution in the case of core applications.

[4]The only mandatory and unique component of an application is the resource logic since it encodes the constraints that transactions of the application must fulfill. Its address is provided in every resource belonging to the application. Transaction and projection functions are provided for "convenience", since arbitrary ways to arrive at transactions fulfilling the resource logic are permitted.

2.4.1. Service commitments

Different services will be needed to run an application on the Anoma network depending on the task it carries out. Nodes in the network can commit to providing these services to other network participants. For example, the application developers might commit to making BLOBs for resource logic, transaction, and projection function objects available for the next N blocks of a specific consensus provider, including storage and bandwidth needs. These *storage* and *bandwidth* commitments, which we can consolidate as *data availability (DA)* commitments, can be unconditional or contingent on, e.g. a minimal sum of user contributions in specific denominations to a specific address.

Another type of commitment would be an *ordering commitment*, in which a consensus provider commits to producing ordering in blocks for transactions containing resources of a specific application for a certain fee. This could be bundled with a commitment to make block data (i.e., the commitment tree and nullifier set) and transaction data available for a specific duration. The ordering and data availability service will often be co-located on the same node (set), using the same key-value store. Alternatively, they could be run on disjoint infrastructure and have internal commitments with each other.

Lastly, nodes can also provide *compute* services, e.g. on-demand data aggregation and projection or proof generation. Although yet to be specified, these could be metered in Nockma operations following the gas model of the ARM [see KG24, p. 19].

A more complex, derived service would be an *indexer* for the transparent subset of transactions of an application providing balance information for all identities in respect to a specified set of consensus providers. A more complex, derived service would be an *indexer* for an application's transparent subset of transactions, providing balance information for all identities for a specified set of consensus providers. This would entail user-facing service commitments specifying cost per query and internal commitments to secure data, bandwidth, and compute provisioning necessary for the indexer's operation.

3. Application Design Patterns

Applications written for the EVM in imperative, computation-ordering-oriented programming languages differ significantly from ARM applications, which are written in the declarative and object-oriented paradigm and within Anoma's resource model.

Accordingly, before presenting detailed reusable design patterns related to the initialization and finalization of resource quantities, authorization, property checking, roles, intents, and tokens, we give general advice on designing application state and logic.

Concerning application state, it is advisable to divide it into granular resource types having a single responsibility. This way, transaction functions only result in minimal overall state change and resource logic functions require fewer operations for evaluation, thus reducing computational costs. For example, instead of creating a single token resource that maintains a mapping of account balances, creating a token resource for each account owner is advisable. This way, multiple tokens can be sent in one transaction without requiring any particular order, transfers can occur locally only involving related controllers, and minimal state needs to be synchronized between token holders after each transfer.
Application logic must be specific enough to ensure that the intended task the application is designed for is carried out (although there is no fixed set of possible execution traces, as this is the case, for example, for the EVM). For owned resources, for example, checks must be written forward-facing, i.e., predicate logic must be located on consumed resources and monitor the created ones. Otherwise, if checks were written backward-facing and located on created resources, malicious actors getting hold of the transaction object before settlement, such as solvers or relayers, could alter resources, still producing valid logic and balance checks. For example, a node getting hold on a transfer transaction could replace a created resource to be owned by a receiver with one having a different owner still satisfying the resource logic checks. Another desirable property is compositionality, which allows application logic to be used with or as part of other applications in a single atomic transaction. This translates into the requirement that a valid transaction can be composed with all other valid transactions so that the resulting composed transaction is still valid. To achieve this, resource logic predicates should be robust and tolerate other resources of the same or different kinds present. Moreover, they should be written succinctly and minimally, i.e., only check the resources related to their consumption (see Section 3.5). Next, we introduce patterns satisfying these requirements being written in the Juvix language [Cza23, CPCMRC24a, CPCMRC24b, CMRCPC24, CHMR24].

3.1. Resource Initialization & Finalization

The first pattern we introduce is about the initial creation and final consumption of resource quantities. Every transaction must be balanced to be settled, which means that the total quantities of created and consumed resources of the same kind must be equal. However, if a new resource quantity of specific kind should be created, there exists no resource of this kind that could be consumed. To satisfy the balance checker, an ephemeral, consumed resource of matching kind and quantity for which the existence check is skipped (see Appendix A.1) must be added to the transaction object. We denote this as *initialization* and visualized it schematically in the top of Figure 2, where an

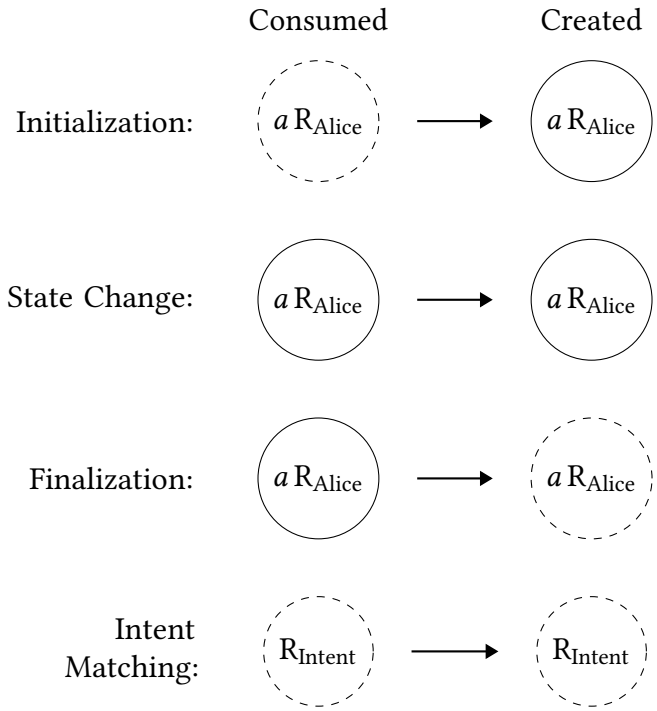

Figure 2. Schematic visualization of (top) resource initialization, (upper mid) state change, (lower mid) resource finalization, and (bottom) intent matching with ephemeral and non-ephemeral resources indicated by dashed and solid circles, respectively.

a Alice-kind resource is consumed, and an ephemeral *a* Alice-kind resource is created. Likewise, for a resource of specific kind and quantity to be finally removed (e.g., to reduce its supply), an ephemeral resource of matching kind and quantity must be added to the transaction object. We denote this as *finalization* and visualized it schematically in the lower middle of Figure 2, where an *a* Alice-kind resource is consumed, and an *a* ephemeral Alice-kind resource is created.

Code examples are shown in Listing 6 for a dummy resource and later in the context of a token in Listings 18 and 19, where an ephemeral token must be part of the mint and burn function and described further in Section 3.10.3.

If not restricted by the resource logic function, ephemeral resources allow everyone to initialize resources for themselves (by setting the nullifier public key `npk` to their identity) or finalize resources they know the nullifier key `nk` for, thus inflating or deflating the resource supply. Although this might be desirable sometimes, it is often not wanted. In the following sections, we show how constraints can be added.

3.2. Finitely Callable

Instead of allowing a resource logic function (or a branch within) to be repeatedly callable, we can constrain it to be callable only a finite number

```
initialize (self : KeyPair) :
↪  Transaction :=
  let
    dummy : Resource :=
      mkDummy@{
        npk := KeyPair.pubKey self;
        eph := false
      };
  in mkTransaction@{
    nullifierKey := KeyPair.privKey
    ↪  self;
    consumed := [dummy@Resource{
    ↪  eph := true; npk :=
    ↪  KeyPair.pubKey self }];
    created := [dummy];
    extraData := Map.empty
  };
```

```
finalize (self : KeyPair) (dummy :
↪  Resource) : Transaction :=
  mkTransaction@{
    nullifierKey := KeyPair.privKey
    ↪  self;
    consumed := [dummy];
    created := [dummy@Resource{eph
    ↪  := true}];
    extraData := Map.empty
  };
```

Listing 6. Transaction function implementations (left) initializing and (right) finalizing a dummy resource, whose constructor is shown in Listing A.4.

of times, e.g., once, which we describe in the following. This is achieved by consuming another non-ephemeral resource in the transaction and requiring its presence in the transaction nullifier set `nfs` from the logic function (branch), which should only be callable once. This way, because every resource is unique and can only be consumed once, the call cannot be repeated after settling the transaction. In practice, a dummy resource, e.g., with an empty `label` and `data` field, can be consumed, resulting in a known and unique nullifier. This nullifier is then stored as part of the resource label or hardcoded in the resource logic and checked to be present in the transaction commitment or nullifier by the resource logic. In both cases, this results in a unique resource kind. The nullifier is then referenced in the `extra` data and required to be present in the transaction by the resource logic. This pattern can be extended to any finite number of calls by specifying a set of nullifiers with the respective cardinality.

3.3. Singleton

With the previous pattern and by applying it to the initialization logic of a resource, we can create a resource kind that can only be instantiated once, which we will refer to as a *singleton*. If the resource is instantiated with a quantity greater than 1, this resource might be split into multiple resources in a subsequent transaction if the resource logic allows it. However, the total supply of the resource will remain fixed. This pattern allows us to create a counter singleton, allowing the creation of other resources with unique identifiers (UIDs) or a token with a fixed supply (see Section 3.10).

```
type ResourceRelationship :=
  mkResourceRelationship {
    origin : Commitment;
    mustBeCreated : Set Commitment;
    mustBeConsumed : Set Nullifier
  };
```

Listing 7. The definition of a message defining relationships between an origin resource and others that must be created or consumed.

3.4. Universal Annuler

Consuming a resource requires the knowledge of the nullifier key `nk` corresponding to the nullifier public key `npk` and the resource plaintext (see Appendix A.1), the former of which is usually shared only with trusted parties. A frequent requirement in applications is to have resources that anyone can nullify within the constraints of their resource logics. Instead of generating a random nullifier key and sharing it with everyone requiring it, the resource creator can select a seed with low entropy that is known by everyone, e.g., 0 (see Listing A.2) and use it to derive the nullifier public key for the resource in question. In particular, since Anoma offers composable identities, people wanting to consume universal resources can compose their identities with this key pair. For example, this pattern can be used, to build a singleton counter that can be incremented by anyone and generates UIDs.

3.5. Resource Relationships

As stated in the beginning of this section, resource logics must support compositionality, i.e., a valid transaction should not become invalid after composition with another valid transaction unless this is specifically wanted. Accordingly, resource logics must tolerate the presence of other resources being created or consumed during transaction execution time. Still, resource logics might want to express that specific other resources must be present during the transaction, either to ensure that these resources have required properties, or to check that a signature authorizes their creation or consumption. We also point out, again, that these checks must happen from a consumed resource because created resources can be replaced by malicious nodes having access to the transaction object (i.e., solvers or gossiping nodes).

Creating a link between the consumed resource conducting the check and other consumed and created resources is possible by requiring a message of a certain format to be present in the transaction's `extra` data, whose formats are shown in Listing 7.

The check proceeds as follows: The resource logic of the consumed resource r^* requiring other resources to be created or consumed as part of the transaction expects a message of known format to be in the `extra` data key-

value map with its own commitment $\texttt{cm}^* = h_{\texttt{cm}}(r^*)$ as the lookup key.[5] The lookup function is shown in Listing 8. Upon successful retrieval of the message, the logic function l^* first checks that the commitment $\texttt{cm}^*$ in the `origin` field links back to the resource r^* that l^* is associated with. Second, it checks that all the created and consumed resources referenced in the `mustBeCeated` or `mustBeConsumed` field are a subset of the commitment set `cms` or nullifier set `nfs` of the transaction, respectively. If the message cannot be found, the message does not link back to r^*, or a referenced resource is missing, the logic function returns `false`. If it passes, the resource logic can continue and either require specific properties to hold over the created resources[6] or verify a signature over the message by an authority, e.g., the owner of the consumed resource, that must be present in the `extra` data as well. This is described in the next section.

3.6. Authorization

Having a message format allowing us to express resource relationships, we can implement an authorization mechanism. For this, we require the signature of an authority to authorize the consumption of a resource and require related resources with known commitments and nullifiers to be created and consumed within the same transaction. For example, this can be a resource owner or originator of a particular resource kind, which we employ later in Section 3.10.2 to ensure that a token can only be initialized and spent by a specific public key. This proceeds as follows: In addition to the message lookup describing the resource relationships that we showed in the previous section, we also require a signature to be present under the lookup key that must be made by a public key being known by the resource logic, i.e., because it is stored in the label, data, or logic function. Upon retrieval of the message-signature pair, the resource logic can then check that the signature is valid, that the related resources are indeed created and consumed as part of the transaction, and that the message links back to the original, consumed resource whose resource logic is conducting the check. The latter is critical because it ensures that the signature cannot be replayed in a different context. These checks are combined into a function shown in Listing 9 and used later in Section 3.10.2 in the resource logics of different token types (see Listings 15 and 16).

[5] It should be noted that we would prefer to use the nullifier $\texttt{nf} = h_{\texttt{nf}}(\texttt{nk}, r^*)$ as both, the lookup key and the `origin` field type, to not unnecessarily leak information about the creation date of r^* to observers. However, this is not possible with the current Anoma node implementation, because only the plaintext r^* and commitment $\texttt{cm}^*$ are known in the local view of the resource logic function l^*, whereas the nullifier $\texttt{nf}^*$ itself and the nullifier key `nk` allowing for its computation are unavailable. This is only a temporary problem and solved when the Anoma node and consequentially Juvix can will adhere to the specifications by having the `tag` as a resource logic public input [see KG24, p. 20], which will refer to the commitment and nullifier depending on whether the resource logic function is created or consumed.

[6] The checks are limited to created resources because, again, the plaintexts of the consumed resources cannot be associated with the nullifiers due to the lack of knowledge about the nullifier keys in the local view of the resource logic.

```
lookupExtraData {Value : Type} (key : Bytes32) (tx : Transaction) : Maybe
↪  Value :=
  let
    keyValueMap : Map Bytes32 Value := anomaDecode (Transaction.extra tx);
  in Map.lookup key keyValueMap;
```

Listing 8. A function looking up a key and returning a value from the key-value map in the transaction `extra` data or nothing if the key does not exist.

```
isAuthorizedBy (signer : PublicKey) (self : Resource) (tx : Transaction) :
↪  Bool :=
  let
    cm := commitment self;
  in case
       lookupExtraData@{
         key := natToBytes32 (anomaEncode cm);
         Value := Pair ResourceRelationship Signature;
         tx
       }
     of
       | nothing := false
       | just (msg, sig) :=
         ResourceRelationship.origin msg == cm
           && anomaVerifyDetached sig msg signer
           && isSubset (ResourceRelationship.mustBeConsumed msg) (nullifierSet
           ↪  tx)
           && isSubset (ResourceRelationship.mustBeCreated msg) (commitmentSet
           ↪  tx);
```

Listing 9. A function checking if the transaction `extra` data map contains a message-signature pair of type `Pair ResourceRelationship Signature` (see Listing 7) with the resource commitment as a lookup key. Upon successful lookup, it is checked that the consumed resource whose commitment was used for the lookup is referenced as in the message's `origin` field, that the signer public key being passed as an input argument has signed the message, and that the `mustBeCreated` commitments and `mustBeConsumed` nullfiers are subsets of the commimtment set `cms` and nullifier set `nfs`, respectively–otherwise, the function returns nothing.

3.7. Property Checks

Instead of requiring a signature over the message to be present, resource logics can check that the referenced, created or consumed resources have specific properties. Similar to the signature check and after a lookup of the resource relationship message as described in Section 3.5, we check that the related resources are indeed created and consumed as part of the transaction and that the message links back to the original, consumed resource whose resource logic is conducting the check. Instead of checking a signature, we obtain the resource plaintexts associated with the `mustBeCreated` commitments and `mustBeConsumed` nullifiers being available from the resource logic private inputs [see KG24, p. 20], which allows us to check their properties. This can be employed to ensure that state is transitioned correctly, e.g., that a counter value is incremented by exactly one.

3.8. Roles

The authorization mechanisms from the previous sections can be used to implement roles, e.g., by storing allowed public keys in the resource's plaintext. The keys can either be stored in the `data` field, where they do not affect the resource's fungibility, in the `label` field, or inside the resource logic function object, where they affect the resource `kind`. The keys can then be used within the resource logic for signature-based checking, e.g., to verify that a role holder has authorized the consumption and creation of related resources or property-based checking, e.g., to verify that the role data has been correctly moved over from consumed to created resources.

3.9. Intents

Intents are unbalanced transactions describing preferred state transitions and are necessary if a sender cannot settle the transaction unilaterally, thus requiring one or multiple counterparties. Matching intents can be composed with each other and add the mutually missing created or consumed resources to the transaction to form a balanced and valid transaction, the process of which is called *intent solving*.

Intents can be realized in two approaches. One approach is to consume the exact resources one is willing to give and create the exact resources one wants to receive. The latter resources must be specified in detail, including the exact nonce and randomness seed. However, this approach does not allow expressing optionality (e.g., "I want either a `5 A` resource or `3 B` resource.") and variability (e.g., "I want at least `5 A` or more."). It is also harder for solvers to match these intents due to the narrow solution space, thus rendering this approach impractical.

In the second approach, the sender creates an ephemeral resource expressing predicates over the state transition via its resource logic function, e.g.,

```
type Asset :=
  mkAsset {
    quantity : Nat;
    kind : Kind
  };

type Quantifier :=
  | Any
  | All;

type QuantifiedAssets :=
  mkQuantifiedAssets {
    quantifier : Quantifier;
    assets : List Asset
  };
```

```
syntax operator of_ additive;

syntax alias of_ := mkAsset;

any (as : List Asset) :
↪ QuantifiedAssets :=
  mkQuantifiedAssets@{
    quantifier := Any;
    assets := as
  };

all (as : List Asset) :
↪ QuantifiedAssets :=
  mkQuantifiedAssets@{
    quantifier := All;
    assets := as
  };
```

Listing 10. Listings showing (left) asset-related type definitions and (right) and operators, aliases, and functions constituting syntactic sugar for an intent DSL.

conditionally requiring resources with specific properties to be part of the transaction or ensuring time constraints. This ephemeral resource, which we refer to as an *intent resource*, must be consumed by a solver identity (that can be specified through the nullifier public key `npk`) within the same transaction [see Figure 2, bottom, and KG24, pp. 25 ff.]. In contrast to the first approach, the predicates allow for expressing optionality and variability, including weighted preferences for different resource kinds, [see Har23, pp. 22 ff.] and general constraints, e.g., with respect to time. This is demonstrated later in Section 3.10.3, where we present the implementation of a simple token swap.

Because application users cannot be expected to code such predicates themselves, application interfaces (see Figure 1) must facilitate intent resource authoring through transaction and projection functions and by employing domain-specific languages.

3.9.1. Domain-Specific Languages

Intent domain-specific languages (DSLs) provide a user-friendly way to translate an intent into an intent resource with the desired predicates encoded in the resource logic function. For example, to encode optionality, an intent DSL can define a `QuantifiedAssets` type consisting of a list of assets and a quantifier, as shown in Listing 10. An asset encodes a desired resource kind and quantity, whereas the quantifier expresses if `Any` or `All` of the listed assets are required in the transactions. This allows formulating Alice's intent from above as the following expression: `any [5 of_ A; 3 of_ B]`. Later, in Section 3.10.3, we employ the DSL to write a `swap` transaction function that

creates a swap intent resource (see Listings 25 and 26).

3.10. Token

With the patterns discussed, we can implement reusable and composable resource logic, transaction, and projection functions for tokens, which we describe in the following.

Three roles exist in the context of a token: The token *originator* can initialize new token resources and characterize the token kind. A token *owner* can consume existing tokens and create new ones with the same or a different owner. A token *receiver* receives created tokens and is specified by the owner of the consumed token.

Additionally, we define a token supply type that can either be *unbound*, *capped*, or *fixed*. For a token with an unbound supply, the originator can *initialize* token resources an unlimited number of times, thus inflating the supply of this token kind. The token owner can *finalize* token resources and, therefore, deflate the supply. For a token with a capped supply, the total quantity is capped by a limiting value.[7] Lastly, for a token with a fixed supply, the originator can only *initialize once* with a given quantity and *never finalize*, thus resulting in a constant total quantity of this resource kind. Note that non-fungible tokens (NFTs) can be realized as tokens with a fixed supply and a quantity of 1.

Independently of the token supply type, a token can be *transferable*. This means that an owner can consume the resource and create corresponding resources for different receivers (including the owner itself) as the owners. Three elementary operations, *transfer*, *split*, and *merge*, can be distinguished. A transfer consumes a token resource and creates one of the same kind owned by a receiver (who can be the current owner). A split consumes a token resource and creates a set of tokens of the same kind with corresponding sub-quantities and corresponding receiving public keys as the new owners. The sub-quantities of the created tokens must add up to the quantity of the consumed resource. A merge consumes a set of multiple token resources of the same kind and creates a token of the same kind owned by a receiver (who can be the current owner). The quantity of the created token must be the sum of all consumed token quantities. For non-transferable tokens, sometimes referred to as soulbound tokens (SBTs), none of the above operations is possible. Next, we present the implementation of the token.

3.10.1. Resource

A token resource with a specific quantity is constructed, as shown in Listing 11, from a specific token label (`tokenLabel : Label`), owned by an external

[7] This supply type is not implemented yet.

```
mkToken (quantity : Nat) (tokenLabel : Label) (npk : PublicKey) {eph : Bool :=
↪  false} : Resource :=
  Resource.mk@{
    logic := tokenLogic (Label.supply tokenLabel);
    label := anomaEncode (tokenLabel);
    quantity;
    data := 0;
    eph;
    npk;
    nonce := rand;
    rseed := rand
  };
```

Listing 11. The token resource constructor.

```
type Label :=
  mkLabel {
    name : String;
    symbol : String;
    decimals : Nat;
    supply : Supply;
    transferable : Bool;
    originator : PublicKey
  };
```

Listing 12. The token label type definition.

```
type FixingNullifier := mk {nf :
↪  Nullifier};

type Supply :=
  | Unbound
  | Capped
  | Fixed FixingNullifier;
```

Listing 13. The token supply type definition.

identity being assigned as the nullifier public key (`npk : PublicKey`). By default, the constructed resource is non-ephemeral.

The token label is shown in Listing 12 and has six data fields: The name (`name : String`), the symbol (`symbol : String`), the decimals(`decimals : Nat`) allowing token quantities to be represented as floating-point values in user interfaces (UIs), the supply-type (`supply : Supply`) encoding the different supply types and shown in Listing 13, the transferability specifier (`transferable : Bool`) denoting the transferability of the token, and the originator public key (`originator : PublicKey`) expressing who has initialized the resource. The `Fixed` supply type shown in Listing 13 is special because it contains an additional `FixingNullifier` field containing the nullifier. Because the nullifier becomes part of the resource label, this results in a unique resource kind, as discussed in Section 3.2.

```
tokenLogic (supply : Supply) : Resource -> Transaction -> Bool :=
  case supply of
    | Unbound := unboundSupplyLogic
    | Capped := cappedSupplyLogic
    | Fixed f := fixedSupplyLogic (FixingNullifier.nf f);
```

Listing 14. The token logic function implementation covering different supply cases.

3.10.2. Resource Logic

The resource logic function is shown in Listing 14, obtained after partial function application depending on the supply type provided in the label. The unbound and fixed case logic functions are shown in Listings 15 and 16, whereas the capped supply logic has not been implemented yet.

For non-ephemeral resources, the fixed and supply case logic functions are identical. In the creation case, both return `true`. In the consumption case, the validity is determined by the same transfer logic shown in Listing 17. The transfer logic checks that the token is transferable as per its label and that the owner has authorized related, consumed, and created resources via a message and signature (see Section 3.6 and Listing 9 therein).

In the case of ephemeral resources, the unbound and fixed supply cases differ. To initialize tokens with unbound supply, ephemeral resources can be consumed if the originator, whose public key is encoded in the token label, authorizes the consumption and creation of related resources. To finalize tokens with unbound supply, ephemeral resources can be created in which case associated logic function will always return `true`.

To initialize tokens with a fixed supply, an ephemeral resource can be consumed once if two conditions are met. First, the originator, whose public key is encoded in the token label, must authorizes the consumption and creation of related resources. Second, the fixing nullifier, which is encoded in the token label as well, must be present in the transaction object. Token with fixed supply can never be finalized. This is because the logic function will always return `false` for created, ephemeral resources.

Lastly, if the resource's lifecycle status is unknown, e.g., because of an unexpected bug, the resource logic must return `false` regardless of its ephemerality.

This resource logic covers the entire token model described initially. In the following, we present the transaction functions constituting the application write interface.

3.10.3. Transaction Functions

We start by introducing the function to initialize and finalize tokens (often called to as minting and burning).

```
unboundSupplyLogic (self : Resource) (tx : Transaction) : Bool :=
  case lifecycle self tx, ephemerality self of
    | Consumed, Ephemeral := unboundSupplyInitializationLogic self tx
    | Consumed, NonEphemeral := transferLogic self tx
    | Created, Ephemeral := true
    | Created, NonEphemeral := true
    | Unknown, _ := false;

unboundSupplyInitializationLogic (self : Resource) (tx : Transaction) : Bool
↪  :=
  isAuthorizedBy (getOriginator self) self tx;
```

Listing 15. The token logic function for tokens with unbound supply together with the dedicated initialization logic function. The transfer logic function is shown in Listing 17.

```
fixedSupplyLogic (nf : Nullifier) (self : Resource) (tx : Transaction) : Bool
↪  :=
  case lifecycle self tx, ephemerality self of
    | Consumed, Ephemeral := fixedSupplyInitializationLogic nf self tx
    | Consumed, NonEphemeral := transferLogic self tx
    | Created, Ephemeral := false
    | Created, NonEphemeral := true
    | Unknown, _ := false;

fixedSupplyInitializationLogic (nf : Nullifier) (self : Resource) (tx :
↪  Transaction) : Bool :=
  isNullifierPresent nf tx && isAuthorizedBy (getOriginator self) self tx;
```

Listing 16. The token logic function for tokens with fixed supply together with the dedicated initialization logic function. The transfer logic function is shown in Listing 17.

```
transferLogic (self : Resource) (tx : Transaction) : Bool :=
  isTransferable self && isAuthorizedBy (Resource.npk self) self tx;
```

Listing 17. The token transfer logic function.

Initialize. As inputs, the `initialize` transaction function as in Listing 18 receives the key pair (`self : KeyPair`) of the sender, a label (`label : Label`), a quantity (`quantity : Nat`) to be created, and a receiver (`receiver : PublicKey`) becoming the owner of the initialized resource. It returns a `Result` type that either resolves into a `TokenError` on failure or a `Transaction` object on success. The function initializes a token by constructing a non-ephemeral resource to be created being owned by the receiver with the provided label and quantity and a corresponding, ephemeral token resource to be consumed being owned by the sender, thus balancing the transaction. Next, it creates an extra data map containing the message describing the resource relationship Section 3.5 and signature with the consumed, ephemeral resource commitment in the `origin` field and the created, non-ephemeral token resource in the `mustBeCreated` commitment set. Subsequently, the consumed and created resources as well as the prepared extra data are used to populate the transaction object. In case the token to be initialized has an unbound supply, the transaction is returned. If the supply is capped, which is currently not supported, the function returns an error. If the supply is fixed and a fixing nullifier must be part of the consumed transaction object (see Section 3.2), the function currently expects a dummy resource plaintext (see Listing A.4) to be part of the node's key-value storage with the nullifier as the lookup key. If it finds the value, it calls the `finalize` transaction function of the dummy (see Listing 6), producing a second transaction object containing the nullifier and related data that can then be composed with the first one (see Listing A.1) and returned. If not, the function silently fails.[8]

Finalize. The `finalize` transaction function shown in Listing 19 receives the sender key pair (`self : KeyPair`) and the plaintext of the resource to be finalized. The function first checks if the sender is the current owner of the passed token resource and returns an error if the check does not pass. If it passes, it checks the supply type. Tokens with capped and fixed supply can not be finalized, as this is not implemented yet, and not be burned, respectively. For tokens with unbound supply, the transaction function creates an ephemeral copy of the token resource, where the nullifier public key (i.e., the owner) is set to the zero public key, thus balancing the transaction. After creation of the associated extra data expressing the resource relationship, the transaction object is returned.

Transfer. The `transfer` transaction function shown in Listing 20 receives the key pair (`self : KeyPair`) of the sender, the resource plaintext (`token : Resource`) of the token resource to be transferred, and a receiver (`receiver : PublicKey`). First, the function checks if the token is transferable and if

[8] This will be improved in a future Anoma node version.

```
initialize (self : KeyPair) (label : Label) (quantity : Nat) (receiver :
↪  PublicKey)
: Result TokenError Transaction :=
  let
    myself : PublicKey := KeyPair.pubKey self;
    nk : PrivateKey := KeyPair.privKey self;
    newToken : Resource := mkToken@{quantity; tokenLabel := label; npk :=
    ↪  receiver};
    ephToken := newToken@Resource{npk := myself; eph := true};
    tx : Transaction := mkTxWithExtraData@{nk; consumed := [ephToken]; created
    ↪  := [newToken]};
   in
     case Label.supply label of
       | Unbound := ok tx
       | Capped := throw mkError@{msg := "Tokens with capped supply are not
       ↪  supported yet."}
       | (Fixed f) := ok
          compose@{
            tx1 := tx;
            tx2 := Dummy.finalize@{self; dummy := anomaGet(FixingNullifier.nf
            ↪  f)}
          };
```

Listing 18. The transaction function initializing a token resource with a given amount being owned by a receiver.

```
finalize (self : KeyPair) (token : Resource)
: Result TokenError Transaction :=
  let
    myself : PublicKey := KeyPair.pubKey self;
    nk : PrivateKey := KeyPair.privKey self;
    owner : PublicKey := Resource.npk token;
    ephToken := token@Resource{eph := true; npk := Zero.pubKey};
  in if
    | owner /= myself := throw mkUnauthorizedError@{expected := myself; actual
    ↪  := owner}
    | else :=
        case getSupply token of
          | Unbound := ok mkTxWithExtraData@{nk; consumed := [token]; created
          ↪  := [ephToken]}
          | Capped := throw mkError@{msg := "Tokens with capped supply are not
          ↪  supported yet."}
          | (Fixed _) := throw mkError@{msg := "Tokens with fixed supply
          ↪  cannot be burned."};
```

Listing 19. The transaction function finalizing a token resource. The function requires the caller to be the owner and handles the different token supply types.

```
transfer (self : KeyPair) (token : Resource) (receiver : PublicKey)
: Result TokenError Transaction  :=
  let
    myself : PublicKey := KeyPair.pubKey self;
    nk : PrivateKey := KeyPair.privKey self;
    owner : PublicKey := Resource.npk token;
    newToken := token@Resource{npk := receiver};
  in if
    | not (isTransferable token) := throw mkNonTransferableError
    | owner /= myself := throw mkUnauthorizedError@{expected := myself; actual
    ↪  := owner}
    | else := ok mkTxWithExtraData@{nk; consumed := [token]; created :=
    ↪  [newToken]};
```

Listing 20. The transaction function transferring a token resource to another owner.

the sender is the current owner of the passed token resource and returns respective errors otherwise. Next, it constructs a token with the same label and quantity as the input token but with the receiver's public key as the owner. Lastly, after again preparing the authorization extra data, a transaction object is constructed consuming the input token and creating the tokens for the receivers.

Split. The `split` transaction function shown in Listing 21 receives the key pair (`self : KeyPair`) of the sender, the resource plaintext (`token : Resource`) of the token resource to be split, and finally, a list of quantities and receivers `quantitiesAndReceivers : List (Pair Nat PublicKey)`. As for the `transfer` function, the `split` function first checks that the input token is transferable and owned by the sender. Additionally, it checks that the input quantities sum up to the input token quantity. If these checks pass, the function creates a list of tokens with the same label as the input token and the specified quantities and receivers. After populating the extra data, the transaction object is returned.

Merge. The `merge` transaction function shown in Listing 22 receives the key pair (`self : KeyPair`) of the sender, a list of resource plaintexts (`tokens : List Resource`) of the token resources to be merged, and finally, a receiver (`receiver : PublicKey`). First, the function checks that the first token is transferable as encoded by its label and that all the labels are identical. Moreover, it checks that all input tokens have the same resource logic and are all owned by the sender. Next, the function constructs a token with the same label as the input tokens, the specified receiver as the owner, and its quantity equal to the total input token quantities. Lastly, a transaction object is constructed, consuming the input tokens and creating the token for the receiver, where the extra data contains a map entry for each consumed token as the lookup key.

```
split (self : KeyPair) (token : Resource) (quantitiesAndReceivers : List (Pair
↪  Nat PublicKey))
: Result TokenError Transaction :=
  let
    myself : PublicKey := KeyPair.pubKey self;
    nk : PrivateKey := KeyPair.privKey self;
    owner : PublicKey := Resource.npk token;
    label : Label := anomaDecode (Resource.label token);
    sum : Nat := for (acc := 0) ((quantity, _) in quantitiesAndReceivers)
    ↪  {quantity + acc};
    balance := Resource.quantity token;
    created : List Resource := map ((quantity, receiver) in
    ↪  quantitiesAndReceivers) {
        mkToken@{quantity; tokenLabel := label; npk := receiver}
      };
  in if
    | not (isTransferable token) := throw mkNonTransferableError
    | owner /= myself := throw mkUnauthorizedError@{expected := myself; actual
    ↪  := owner}
    | balance /= sum := throw mkInsufficientQuantityError@{limit := balance;
    ↪  actual := sum}
    | else := ok mkTxWithExtraData@{nk; consumed := [token]; created};
```

Listing 21. The transaction function splitting a token resource with a given quantity into multiple sub-quantities owned by different receivers.

This is because every consumed token resource logic will be conducting the validity check and require a `ResourceRelationship` message and associated owner signature to be present.

Send. The `send` transaction function shown in Listing 23 is a convenience function allowing the sending of the total or partial quantity of a token to a receiver, thus providing an abstraction over the `transfer` and `split` functions. It receives the key pair (`self` : `KeyPair`) of the sender, the resource plaintext (`token` : `Resource`) of the token resource to be consumed, a quantity (`quantity` : `Nat`) to be sent, and a receiver (`receiver` : `PublicKey`). The function compares the available quantity from the input token quantity with the requested quantity to be sent. If the available quantity is less than the requested quantity, an error is returned. If the quantities are equal, the transfer function is called. Lastly, if the available quantity is greater than the requested quantity, two tokens are created via the `split` function, where the first has the requested quantity as its quantity and is owned by the receiver and the second has the difference to the available quantity as its quantity and is owned by the sender.

```
merge (self : KeyPair) (tokens : List Resource) (receiver : PublicKey)
: Result TokenError Transaction :=
  case tokens of
    | nil := throw mkInsufficientElementsError@{limit := 1; actual := 0}
    | (t :: _) :=
    let
      myself : PublicKey := KeyPair.pubKey self;
      nk : PrivateKey := KeyPair.privKey self;
      kind : Kind := anomaKind t;
      totalQuantity : Nat := for (acc := 0) (t in tokens) {Resource.quantity t
      ↪  + acc};
      merged := t@Resource{quantity := totalQuantity; npk := receiver};
    in if
        | not (isTransferable t) := throw mkNonTransferableError
        | else :=
          case find ((/=) myself) (map Resource.npk tokens) of
            | just notMyself := throw mkUnauthorizedError@{expected := myself;
            ↪  actual := notMyself}
            | nothing :=
              case find ((/=) kind) (map anomaKind tokens) of
                | just otherKind := throw mkInvalidKindError@{expected := kind;
                ↪  actual := otherKind}
                | nothing := ok mkTxWithExtraData@{nk; consumed := tokens;
                ↪  created := [merged]};
```

Listing 22. The transaction function merging multiple token resources into one, if all input resources are owned by the caller, have the same label,and have the same logic.

```
send (self : KeyPair) (token : Resource) (quantity : Nat) (receiver :
↪  PublicKey)
: Result TokenError Transaction  :=
  let
    myself : PublicKey := KeyPair.pubKey self;
    balance : Nat := Resource.quantity token;
  in case (compare balance quantity) of
      | LT := throw mkInsufficientQuantityError@{limit := balance; actual :=
      ↪  quantity}
      | EQ := transfer self token receiver
      | GT :=
        let
          difference : Nat := toNat (intSubNat balance quantity);
        in
          split self token [(quantity, receiver); (difference, myself)];
```

Listing 23. A convenience transaction function to send a token amount. After checking the available and requested amount, the function either calls `transfer`, `split`, or throws and error.

```
swap (self : KeyPair) (token : Resource) (want : QuantifiedAssets) (solver :
↪  PublicKey)
: Result TokenError Transaction :=
  let
    myself : PublicKey := KeyPair.pubKey self;
    nk : PrivateKey := KeyPair.privKey self;
    owner : PublicKey := Resource.npk token;
  in if
    | not (isTransferable token) := throw mkNonTransferableError
    | owner /= myself := throw mkUnauthorizedError@{expected := myself; actual
    ↪  := owner}
    | else := ok
      mkTxWithExtraData@{
        nk;
        consumed := [token];
        created := [mkSwapIntent@{want; receiver := myself; solver}]
      };
```

Listing 24. The transaction function swapping a token resource for a list of quantifier assets.

Swap. All the transaction functions presented so far can be settled by the sender without requiring a second party. The `swap` transaction function shown in Listing 24, however, includes an intent resource (see Section 3.9) being shown in Listings 25 and 26 and, thus, needs a counterparty to be settled. It receives the key pair (`self : KeyPair`) of the sender, the plaintext (`token : Resource`) of the resource to be consumed in the swap, a list of quantified assets (`want : QuantifiedAssets`, see Section 3.9) to receive, and a receiver (`receiver : PublicKey`).

Like the `transfer` function, it checks if the passed token is transferable and if the sender is the current owner and returns respective errors otherwise. If the checks pass, the token is consumed, a swap intent resource is created, and the authorization extra data is populated with the token and intent resource being referenced in the `origin` and `mustBeCreated` fields, respectively. This prevents the intent from being removed from the transaction object. The intent resource itself, shown in Listing 25, is created from a wanted list of quantified assets (`want : QuantifiedAssets`) and the public keys of the receiver and solver (`receiver solver : PublicKey`). The `want` asset list and the receiver public key are used in the resource logic Listing 26 to specify the properties of the created resources that the sender desires to receive. If the intent is created and ephemeral, it is checked if the assets specified in the `want : QuantifiedAssets` record are part of the commitment set of the transaction object with the receiver as the owner. Here, two cases must be distinguished. For the `Any` quantifier, only one asset must be present for the intent to be satisfied. For the `All` quantifier, all assets from the list must be present. Given this information, the solver identity being specified as

```
mkSwapIntent (want : QuantifiedAssets) (receiver solver : PublicKey) :
↪  Resource :=
  Resource.mk@{
    logic := swapIntentLogic want receiver;
    label := anomaEncode "TokenSwapIntent";
    quantity := 1;
    data := 0;
    eph := true;
    npk := solver;
    nonce := rand;
    rseed := 0
  };
```

Listing 25. The swap intent resource constructor.

```
swapIntentLogic (want : QuantifiedAssets) (receiver : PublicKey) (self :
↪  Resource) (tx : Transaction)
: Bool :=
  case lifecycle self tx, ephemerality self of
    | Created, Ephemeral :=
      let
        created := map commitmentResource (Transaction.commitments tx);
      in case QuantifiedAssets.quantifier want of {
           | Any :=
             any (asset in QuantifiedAssets.assets want)
               includesAsset@{ asset; receiver; resources := created}
           | All :=
             all (asset in QuantifiedAssets.assets want)
               includesAsset@{ asset; receiver; resources := created}
         }
    | Consumed, Ephemeral := true
    | _, _ := false;
```

Listing 26. The swap intent logic function implementation.

the annuler in the intent resource constructor (which can be a composed or universal identity; see Section 3.4) can try to find a matching, unbalanced transaction object from a counterparty. If a match is found, the transaction objects of the two or more parties can be composed (see Listing A.1), and the created intent resources can be consumed by the solver to produce a balanced and valid transaction. Note that the intent resource logic returns `true` on consumption as long as the consumed intent resource is ephemeral. For non-ephemeral resources or one's with unknown lifecycle status, the logic will always return false.

The elementary transaction functions presented are reusable and can be composed with each other. For example, a resource can be initialized and split, or multiple resources can be merged and sent within an atomic transaction.

```
balance (resources : Set Resource) : Nat :=
  for (sum := 0) (r in Set.toList resources) {sum + Resource.quantity r};
```

Listing 27. The projection function calculating the balance as the total quantity of from a set of resources. This assumes that the passed resources have the same kind and are owned by the same account.

```
fetchOwnedResources (kind : Kind) (account : PublicKey) : Set Resource :=
  let
    ownedResources : List Commitment := anomaGet (kind, account);
  in Set.fromList (map commitmentResource ownedResources);
```

Listing 28. A function fetching owned resources from the nodes local key-value store. This assumes that a key-value map of public keys and commitments exists and is maintained for each resource `kind` by the node.

Next, we discuss projection functions for tokens.

3.10.4. Projection Functions

The `balance` projection function shown in Listing 27 receives a set of owned token resource (`resources : Set Resource`) that are assumed to be of the same kind and owned by the same account as the input and returns the balance as the sum over all resource quantities. For now, the set of resources can be obtained via a helper function shown in Listing 27 that receives the resource kind (`kind : Kind`) of the resource to look up, and the account (`account : PublicKey`) to calculate the balance for. The kind and account constitute a lookup key in Anoma's key-value store to the commitment set of resources of the specified kind being owned by the account. Currently, we assume the set to be up-to-date. In the future, data indexing and availability services can provide this data and will be discussed in Section 2.4.1.

4. Applications

With the patterns and primitives discussed in the last section, we can now describe the first applications.

4.1. Kudos

Kudos is an accounting primitive embodying trust relationships between identities[compare Goe22, LPNS23]. As such, it allows identities to create their own tokens and specify their issuance and transfer conditions. Use cases, for example, are traffic accounting for peer-to-peer routing, trust or reputation systems, accounting and swaps for bandwidth, storage or compute resources. Trust and reputation systems could use non-transferable kudos, for example, to express user ratings of interactions with a service. In contrast, routing and resource accounting and exchanges could utilize transferable

```
mkKudoLabel (originator : PublicKey) : Label :=
  mkLabel@{
    name := "Kudos";
    symbol := "KDS";
    decimals := 18;
    supply := Unbound;
    transferable := true;
    originator
  };
```

Listing 29. The constructor of an example Kudos label (see Listing 12) for a transferable Kudos token with unbound supply.

kudos to discern which specific bandwidth or storage is being recorded or traded.

With the token primitives provided in Section 3.10, in particular the `initialize` transaction function discussed in Section 3.10.3, identities can create new kudos denominations by encoding the desired properties in the token label (see Listing 12) and utilizing the token logic from Listing 14. An example of a Kudos label is shown in Listing 29.

Through the supply type (see Listing 13), identities can limit the total supply or create Kudos with unbound supply relying on trust relationships. By issuing non-transferable Kudos, SBTs can be created representing commitments, credentials, and affiliations [OWB22]. Transaction functions such as `merge`, `split`, `transfer`, `send`, and `swap`, transferable Kudos can be distributed and accumulated as credit in exchange for goods and services. Lastly, Kudos with capped or unbound supply can be destroyed using the `finalize` function, e.g., for supply correction.

5. Concluding remarks

In this report, we described the general architecture of applications, an initial set of application design patterns and primitives, and Kudos as the first application. In upcoming versions of this report, these will be refined, and new applications will be added, namely:

- Multichat: A distributed messaging protocol without a central server operator.
- A simplified Proof-of-Stake protocol assigning voting power to validators.
- Public Signal: An intent-centric crowd-funding protocol being supply- but also demand-side driven.

6. Acknowledgements

We would like to thank Christopher Goes for his guidance on shaping the report's scope and emphasis and the time he took for discussions on authorization mechanisms and application service commitments, Paul Cadman for all the help and discussions, Lukasz Czajka and Jan Mas Rovira for their work on the Juvix compiler, and Yulia Khalniyazova for her clarifications and answers related to the ARM specifications. This work was supported by the Anoma Foundation.

References

CHMR24. Paul Cadman, Michael Andreee Heuer, and Jan Mas Rovira. A Juvix library for writing Anoma applications v0.4.0, 2024. URL: https://github.com/anoma/juvix-anoma-stdlib/releases/tag/v0.4.0. (cit. on p. 11.)

CMRCPC24. Paul Cadman, Jan Mas Rovira, Lukasz Czajka, and Jonathan Prieto-Cubides. Immutable container types for Juvix v0.13.0, 2024. URL: https://github.com/anoma/juvix-containers/releases/tag/v0.13.0. (cit. on p. 11.)

CPCMRC24a. Paul Cadman, Jonathan Prieto-Cubides, Jan Mas Rovira, and Lukasz Czajka. Juvix: A Language for Intent-Centric and Declarative Decentralized Applications, 2024. URL: https://docs.juvix.org/0.6.3/README.html. (cit. on pp. 3 and 11.)

CPCMRC24b. Paul Cadman, Jonathan Prieto-Cubides, Jan Mas Rovira, and Lukasz Czajka. The Juvix standard library v0.5.0, 2024. URL: https://github.com/anoma/juvix-stdlib/releases/tag/v0.5.0. (cit. on p. 11.)

Cza23. Lukasz Czajka. The Core language of Juvix. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8268849, doi:10.5281/zenodo.8268850. (cit. on pp. 3 and 11.)

Dan15. Quynh H. Dang. Secure Hash Standard. Technical Report August, National Institute of Standards and Technology, Gaithersburg, MD, Jul 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, doi:10.6028/NIST.FIPS.180-4. (cit. on p. A-2.)

Goe22. Christopher Goes. Towards heterotopia - the prerequisite cultural and technological substrate for a return to a world of scale-free credit money, 2022. URL: https://pluranimity.org/2022/09/26/towards-heterotopia/. (cit. on p. 30.)

Goe24. Christopher Goes. Anoma as the universal intent machine for Ethereum, 2024. URL: https://ethresear.ch/t/rfc-draft-anoma-as-the-universal-intent-machine-for-ethereum/19109. (cit. on p. 3.)

GYB23. Christopher Goes, Awa Sun Yin, and Adrian Brink. Anoma: a unified architecture for full-stack decentralised applications. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8279841, doi:10.5281/zenodo.8279842. (cit. on p. 3.)

Har23. Anthony Hart. Constraint Satisfaction Problems: A Survey for Anoma. *Anoma Research Topics*, Oct 2023. URL: https://doi.org/10.5281/zenodo.10019112, doi:10.5281/zenodo.10019113. (cit. on p. 18.)

HKS24. Tobias Heindel, Aleksandr Karbyshev, and Isaac Sheff. Heterogeneous Narwhal and Paxos. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.10498998, doi:10.5281/zenodo.10498999. (cit. on p. 3.)

HR24. Anthony Hart and D Reusche. Intent Machines. *Anoma Research Topics*, Feb 2024. URL: https://doi.org/10.5281/zenodo.10498992, doi:10.5281/zenodo.10654543. (cit. on pp. 3 and 4.)

KG24. Yulia Khalniyazova and Christopher Goes. Anoma Resource Machine Specification. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.10498990, doi:10.5281/zenodo.10689620. (cit. on pp. 3, 4, 10, 15, 17, 18, A-1, A-2, A-3, and A-4.)

KS24. Aleksandr Karbyshev and Isaac Sheff. Heterogeneous Paxos 2.0: the Specs. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.12572557, doi:10.5281/zenodo.12572558. (cit. on p. 3.)

LPNS23. Andrew Lewis-Pye, Oded Naor, and Ehud Shapiro. Grassroots Flash: A Payment System for Grassroots Cryptocurrencies, 2023. URL: https://arxiv.org/abs/2309.13191, arXiv:2309.13191, doi:10.48550/arXiv.2309.13191. (cit. on p. 30.)

OWB22. Puja Ohlhaver, Eric Glen Weyl, and Vitalik Buterin. Decentralized Society: Finding Web3's Soul. *SSRN Electron. J.*, May 2022. URL: http://dx.doi.org/10.2139/ssrn.4105763, doi:10.2139/ssrn.4105763. (cit. on p. 31.)

She24. Isaac Sheff. Cross-Chain Integrity with Controller Labels and Endorsement. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.10498996, doi:10.5281/zenodo.10498997. (cit. on pp. 3 and A-3.)

SWRM24. Isaac Sheff, Xinwen Wang, Robbert Van Renesse, and Andrew C. Myers. Heterogeneous Paxos. *Leibniz Int. Proc. Informatics, LIPIcs*, 184(Opodis):1–16, Jun 2024. doi:10.4230/LIPIcs.OPODIS.2020.5. (cit. on p. 3.)

Wik24. Wikipedia contributors. Application software — Wikipedia, The Free Encyclopedia, 2024. URL: https://en.wikipedia.org/w/index.php?title=Application_software&oldid=1238738001. (cit. on p. 2.)

A. Appendix

A.1. The Resource Object

In this section, we briefly describe the definitions related to resource objects. For more details, see the ARM specifications [KG24, pp. 5 ff.].

We start with introducing notation. When referring to the resource in its entirety as a unit of state, we will use the symbol r and call it the resource *plaintext*. When referring to it as a data structure and one of its components, we will use the notion $r.$ followed by the component name in monospaced font.

In the following, we give an overview of the primary data components and secondary, computable components. The most characteristic resource components are its quantity, label, and logic function. The resource *quantity* (`quantity : Nat`) is a natural number, including zero, and expresses how many objects this resource describes. The resource *label* (`label : Nat`[9]) specifies the fungibility domain–or, in other words, what this resource describes. The resource *logic function* expresses the predicates under which the resource can be created and consumed. Instead of storing the logic function representation directly in the resource plaintext, only the *resource logic hash* (`logic : Nat`[10]) is stored. The former can be looked up in the node's key-value storage. This

[9] The `label` type will be changed to `String`.

[10] The `logic` type will be changed to `List Byte`. Currently, the full logic function is stored directly as a `Nat` and not its hash.

saves storage space in the commitment accumulator and provides information-flow control because the function implementation can be revealed only to selected parties.

Both, the *label* and *logic* component are used to compute the resource *kind*

$$r.\texttt{kind} = h_{\texttt{kind}}(r.\texttt{logic}, r.\texttt{label}) \tag{A.1}$$

where $h_{\texttt{kind}}$ is the Secure Hash Algorithm 256 (SHA-256) [Dan15] hash function. Altogether, this allows, for example, to define a `5 Apple` resource, where `5` is the quantity and `Apple` is the kind, which also encodes the predicates under which the apple can be created and consumed.

The ARM ensures that all resource logics are satisfied and that the quantities of created and consumed resources of the same kind balance[see KG24, p. 8].

Besides these characteristic components, more fields are present in the resource plaintext. One of them is the *ephemerality* flag (`eph : Bool`). If set, the ARM does not check the resource's existence in the commitment accumulator on consumption or creation, which is essential for resource initialization and finalization (see Section 3.1), respectively. This satisfies the balance checker when a resource with a certain quantity is created for the first time or permanently deleted. Moreover, it allows a resource to present only during the transaction, which enables intents (see Section 3.9).

Another one is the *data* field (`data : Nat`[11]), which allows arbitrary data (or a reference to a data BLOB) to be stored in the resource's plaintext, e.g., to indicate who owns it or when it was created. This data does not affect the resource fungibility.

In case the resource logic requires a source of randomness, a *randomness seed* (`rseed : Nat`) is stored. Because resource plaintexts are visible to transaction observers (i.e., other nodes and solvers) in the transparent case and knowledge of the seed would allow predicting numbers generated by pseudorandom number generators, this data field is only relevant for the shielded case. Additionally, resources carry a *nonce* (`nonce : Nat`) ensuring that resources are unique, even if all other data components in the plaintext are the same.

Given the plaintext r, the *commitment* (`cm : Nat`[12]) of a resource is computed as

$$\texttt{cm} = h_{\texttt{cm}}(r) \tag{A.2}$$

where $h_{\texttt{cm}}$ is again the SHA-256 hash function. The plaintext and commitment of a resource remain constant over the complete lifecycle from creation to consumption.

[11] The `data` type will be changed to `List Byte` and the field will be renamed to `value`.

[12] The `cm` type will be changed to `List Byte`.

To consume a resource, a *nullifier* (`nf : Nat`[13])

$$r.\mathsf{nf} = h_{\mathsf{nf}}(\mathsf{nk}, r) \tag{A.3}$$

must be revealed being computed from the resource plaintext r and a *nullifier key* `nk`, where h_{nf} is again the SHA-256 hash function. The nullifier key `nk` can be randomly generated and shared with selected parties, thus allowing them to consume the resource in agreement with the associated resource logic. The correspondence between the revealed nullifier `nf` and the resource to be consumed is achieved by storing a *nullifier public key* (`npk : Nat`[14])

$$r.\mathsf{npk} = h_{\mathsf{npk}}(\mathsf{nk}) \tag{A.4}$$

in the resource plaintext being derived from the nullifier key `nk`.

A.2. Transaction Definitions

In this section, we briefly describe the definitions related to transaction objects. For more details, see the ARM specifications [KG24, pp. 11 ff.].
A transaction contains a set of *commitments* (`commitments : List Commitment`[15]) and a set of *nullifiers* (`nullifiers : List Nullifier`[16]) of the resources being created and consumed, respectively. This includes ephemeral resources as well.

Additionally, a transaction contains a set of commitment tree state *roots* (`roots : List Nat`[17]) that are used to prove the membership of resources in the commitment tree, which are skipped for ephemeral resources. Multiple roots can exist because transactions can be relayed and incrementally composed by solvers before settlement. If other unrelated transactions have been settled meanwhile, this can result in different state roots. It is recommended to use fresher state roots for membership proofs to not leak information about a resource's age[KG24, pp. 7 ff.]. In case of conflicting state roots, i.e., double-spends, the controller must detect and prevent this, which is described in [She24, pp. 11 ff.].
Another data component to consider is the list of *resource deltas* (`delta : List DeltaComponent`[18]), where

$$r.\Delta = h_{\Delta}(\mathsf{kind}, r.\mathsf{quantity}) \tag{A.5}$$

[13] The `nf` type will be changed to `List Byte`.
[14] The `npk` type will be changed to `List Byte`.
[15] The `commitments` type will be changed to `Set Commitment`.
[16] The `nullifiers` type will be changed to `Set Nullifier`.
[17] The `roots` type will be changed to `Set Root`, where `Root` will be of `List Byte` type.
[18] The `delta` type will be changed to `Set DeltaComponents`

where h_Δ is the Pedersen commitment scheme being additively homomorphic, i.e.,

$$r_1.\Delta + r_2.\Delta = h_\Delta(\texttt{kind}, r_1.\texttt{quantity} + r_2.\texttt{quantity}) \tag{A.6}$$

and kind-distinct for different kinds

$$h_\Delta(r_1.\texttt{kind}, r_1.\texttt{quantity}) + h_\Delta(r_2.\texttt{kind}, r_2.\texttt{quantity}) = h_\Delta(\texttt{kind}, q_{\texttt{kind}}) \tag{A.7}$$

so that it is computationally infeasible to compute `kind` and `quantity`. [19]

The *transaction balance* Δ_{tx} is a computable component representing the total quantity change of the transaction and being derived from the resource deltas as follows

$$tx.\Delta = \sum_j r_{i_j}.\Delta - \sum_j r_{o_j}.\Delta = \sum_j h_\Delta(\texttt{kind}, q_{\texttt{kind}}), \tag{A.8}$$

where r_i and r_o are the created and consumed resources, respectively, and the kind-distinctness property allows resources of all kinds to be added without the need to distinguish between them. Δ_{tx} has to compute to 0 for a transaction to be balanced.

In this context, the transaction contains a set of *proof records* (`proofs : List Proof`) [KG24, pp. 13 ff.] proving that the resource logics are satisfied and the transaction balance Δ_{tx} is correctly derived from the resource deltas and commits to a balancing value of 0. Moreover, it contains a set of *compliance proofs* (`complianceProofs : List ComplianceProof`) (with `ComplianceProof : Type := Pair Nat Nat;`) proving that the transaction complies with the ARM, i.e., that each resource was consumed strictly after it has been created and that the commitments and nullifiers are derived correctly.
Another important component , particularly for applications, is *extra data* (`extra : Nat`[20]), which is a map and can contain arbitrary information to be read in the resource logics accompanied by a deletion criterion (see Section 2.4.1).
Moreover, it contains a *preference function* (`preference : Nat`[21]) taking a transaction object as an input and signalling how preferable the state transition, which is relevant in the context of intent-solving. Lastly, it contains an *information-flow-control predicate* (`IFCPredicate : Transaction -> ExternalIdentity -> Bool`) specifying the transaction visibility for certain identities, which has not yet been implemented in the Anoma node client and Juvix.

The composition function implementation used in the current Anoma node version is shown in Listing A.1.

[19] Currently, an element of `delta` is defined as

```
type DeltaComponent := mk {denom : Nat; sign : Bool; amount : Nat};
```

thus deviating from this definition.

[20] The `extra` type should be changed to `Map (List Byte) (List Byte)`.

[21] The `preference` type should be changed to `Transaction -> Float`, where the output is normalised on the interval $[0, 1]$.

A.3. Code Listings

```
compose (tx1 tx2 : Transaction) : Transaction :=
  Transaction.mk@{
    roots := Transaction.roots tx1 ++ Transaction.roots tx2;
    commitments := Transaction.commitments tx1 ++ Transaction.commitments tx2;
    nullifiers := Transaction.nullifiers tx1 ++ Transaction.nullifiers tx2;
    proofs := Transaction.proofs tx1 ++ Transaction.proofs tx2;
    complianceProofs := Transaction.complianceProofs tx1 ++
    ↪  Transaction.complianceProofs tx2;
    delta := Transaction.delta tx1 ++ Transaction.delta tx2;
    extra :=
      let
        kvList1 : List (Pair Bytes32 Bytes) := Map.toList (anomaDecode
        ↪  (Transaction.extra tx1));
        kvList2 : List (Pair Bytes32 Bytes) := Map.toList (anomaDecode
        ↪  (Transaction.extra tx2));
      in anomaEncode (Map.fromList (kvList1 ++ kvList2));
    preference := 0
  };
```

Listing A.1. The transaction composition function implementation.

```
module Universal;
  keyPair : KeyPair :=
    KeyPair.mk@{
      pubKey := PublicKey.mk
      ↪  0x29da598ba148c03aa643e21d77153265730d6f2ad0a8a3622da4b6cebc276a3b;
      privKey :=
        PrivateKey.mk
          0x29da598ba148c03aa643e21d77153265730d6f2ad0a8a3622da4b6cebc276a3b
          ↪  0000000000000000000000000000000000000000000000000000000000000000
```

Listing A.2. A universal keypair derived from the seed value 0 in little-endian byte order.

```
--- Signs a message with a private key and returns the signed message.
builtin anoma-sign
axiom anomaSign : {Message : Type} -> Message -> PrivateKey -> SignedMessage;

--- Signs a message with a private key and returns the signature.
builtin anoma-sign-detached
axiom anomaSignDetached : {Message : Type} -> Message -> PrivateKey ->
↪  Signature;


--- Verifies a signed message against a public key.
builtin anoma-verify
axiom anomaVerify : SignedMessage -> PublicKey -> Bool;

--- Verifies a signature against a message and public key.
builtin anoma-verify-detached
axiom anomaVerifyDetached : {Message : Type} -> Signature -> Message ->
↪  PublicKey -> Bool;
```

Listing A.3. Builtins for signing and verifying messages.

```
mkDummy (npk : PublicKey) (eph : Bool) : Resource :=
  Resource.mk@{
    logic := alwaysTrueLogic;
    label := 0;
    quantity := 1;
    data := 0;
    eph;
    npk;
    nonce := rand;
    rseed := rand
```

Listing A.4. The constructor of a dummy resource.