Note for readers: this is a follow-on thread to discussions here, although it should be readable standalone.

Introduction

In the Anoma protocol, we do not want to pick a single canonical encoding scheme (serialization format) or virtual machine, both because:

- fashions in serialization schemes and virtual machines change over time, and what scheme / VM makes most sense or is most suitable depends on specific hardware and specific applications
- we do not need

to, and so avoiding doing so makes the protocol simpler and easier to understand, since we can abstract the implementation details of serialization schemes and virtual machines, and one can understand how the protocol works without understanding those details (or being misled into thinking that they're specially relevant for how the protocol works, which they aren't)

In order to facilitate interoperability between encoding schemes and virtual machines, I think we do

need to standardize:

- a type of data types

, which allows for data to be deserialized based on a known type and for inputs and outputs to functions represented in different virtual machines to be typechecked

- a multiformat for encoding schemes
- a multiformat for virtual machines (more context here)
- the functions each encoding scheme must provide, and the expected properties of those functions
- the functions each virtual machine must provide, and the expected properties of those functions

The following is written as a potential component of the specification, in order to define these concepts clearly and lay out how interfaces and interactions would work. I'm putting this down in writing in order to solicit review - particularly from @AHart @isheff @tg-x @mariari @degregat @vveiln @jonathan @graphomath - review from others welcome as well.

Data values and data types

The protocol standardises basic types and general algebraic data types. All messages sent and received within and between nodes and all parts of state must have an associated data type. Data types used by the protocol itself are fixed a priori (by this specification). Implementation languages will typically represent these types as native types in their typesystem, such that the implementation language can provide typechecking. Data types used by applications are chosen by application authors, serialised, and passed around at runtime in order to handle data appropriately.

Data types

Basic types

A basic type

is defined as either:

- a boolean (bit) type
- a ring type Z_n

of natural numbers \mathrm{mod}~n

- a finite field type \mathbb{F}_n

of order n

- a binary (binary data of unbounded length) type

type BasicType := | BooleanT | RingT Nat | FiniteFieldT Nat | BinaryT

Data types

A data type

is defined as either:

- a basic type
- a product of other algebraic data types
- a coproduct of other algebraic data types
- a function type from one data type to another data type

type DataType := | BasicT BasicType | ProductT [DataType] | CoproductT [DataType] | FunctionT DataType DataType

Basic values

A basic value

is defined as either:

- a boolean value
- a ring value n

(between 0

and n-1

)

- a finite field value \mathbb{F}_n

(a natural number n

represents the $n$th element of the finite field)

- a binary (binary data of unbounded length) value
- a function value (represented with a particular virtual machine, identified by a natural number)

type BasicValue := | BooleanV Boolean | RingV Nat | FiniteFieldV Nat | BinaryV Bytestring | FunctionV Nat Bytestring

Data value

A data value

is defined as either:

- a basic value
- a tuple of other data values (inhabitant of a product type)
- a option with an index and a data value (inhabitant of a coproduct type)

type DataValue := | BasicV BasicValue | TupleV [DataValue] | OptionV Nat DataValue

Typechecking

The typechecking relation

typecheck : DataValue -> DataType -> Boolean

can be implemented in the obvious manner. Typechecking functions would require typechecking support from particular virtual machines, which isn't covered for now, but could be added in the future in a straightforward way.

Encoding scheme

An encoding scheme

is a bijective mapping between structured data and a series of bytes, uniquely defined by the pair of serialisation and deserialisation functions.

Serialisation

The serialise

function serialises a data value into a bytestring.

type Serialise = DataValue -> Bytes

Deserialisation

The deserialise

function attempts to deserialise a bytestring into a data value of the specified type.

type Deserialise = DataType -> Bytestring -> Maybe DataValue

Properties

These functions must be inverses of each other, in that:

- deserialising a serialised value will result in Just
- serialising a deserialised value will result in the same bytestring

Furthermore, the mapping - given a type - must be bijective: fixing a given type, no two distinct bytestrings can deserialise into the same value, and no two distinct values can serialise into the same bytestring. A bijective mapping without fixing a type can be achieved simply by also serialising the type.

Multiformat

The protocol standardizes a table of encoding schemes, where each encoding scheme is associated with a unique natural number. For example:

Code	Encoding scheme
0x01	SSZ
0x02	JSON
0x03	Borsh

Nodes running the protocol then associate each number with a pair of serialise

and deserialise

functions. In order to interoperate correctly, nodes must agree on which number is associated with which encoding scheme, so this table is part of the definition of any particular protocol version, and new entries to the table, once added, cannot be changed. In general, adding new entries to the table should not break anything - a node encountering an encoding scheme it does not know simply fails.

Virtual machine

A virtual machine

is a way to represent functions, uniquely defined by three functions encode

, decode

, and evaluate

. The protocol standardizes a table of virtual machines, where each encoding scheme is associated with a unique natural number. We will refer to the virtual machine associated with n

as VM_n

, and associated functions as encode_n

, decode_n

, and evaluate_n

. Each virtual machine, in the implementation language, also comes with an opaque internal representation type VM_n.t

.

Decoding

The decoding function decode_n

attempts to decode a DataValue into an internal representation VM_n.t

. Decoding which encounters a FunctionV

associated with a different virtual machine will simply represent that as data (instead of code) in the internal representation type.

type Decode = DataValue -> Maybe VM_n.t

Encoding

The encoding function encode_n

encodes an internal representation of a function and/or data into a DataValue

. Functions in the internal representation will be serialised in some fashion and paired with the natural number associated with the virtual machine in a FunctionV

.

type Encode = VM_n.t -> DataValue

Properties

The encoding and decoding functions must be inverses of each other, in that:

- decoding an encoded value will result in Just
- encoding a decoded value will result in the original internal representation

Evaluation

The evaluation function evaluate_n

calls a function (in the internal representation) on the provided list of arguments (in the original representation). Evaluation must be deterministic. Evaluation must also meter gas

, a measure of compute and memory resource expenditure. Different virtual machines will have different gas scales. Evaluation takes a gas limit

. During VM internal execution, gas must be tracked, and evaluation must terminate if the gas limit is exceeded. Should execution complete successfully within the gas limit, the VM must return the gas actually used.

type Evaluate = VM_n.t -> [VM_n.t] -> Natural -> (Maybe VM_n.t, Natural)

Note: In the future, gas will likely change from a scalar to a vector to allow for metering compute and memory resources differently.

Provable virtual machines

A provable

virtual machine is a virtual machine with a proof type P_n

and two additional functions parameterized over P_n

, prove_n

and verify_n

.

Proving

The proving function prove_n

generates a proof for a given program (logical relation), public input, and private input.

type Prove = VM_n.t (T0 -> T1 -> boolean) -> VM_n.t (T0) -> VM_n.t (T1) -> P_n

Verification

The verification function verify_n

verifies a proof for a given program and public input.

type Verify = VM_n.t (T0 -> T1 -> boolean) -> VM_n.t (T1) -> P_n -> boolean

Properties

These functions must be correct

and complete

, in that:

- valid proofs can only be created for valid inputs (correctness

), and a valid proof can be created for any valid input (completeness

)

- i.e. verify f public proof = true

if and only if proof = prove f public' private'

where public = public'

and evaluate f [public', private'] g = (true, _)

for some sufficient gas limit g

(we could probably split evaluation into gassy and gassless versions)

Should P_n

not reveal any information about VM_n.t (T0)

, the provable virtual machine can be said to be zero-knowledge

.

Multi-functions

Multiencoding

The multiencode

function takes a DataValue

and a set of preferences, and tries to encode it in a bytestring according to those preferences. Preferences determine which encoding scheme(s) is/are chosen, and whether or not we attempt to convert between virtual machine representations (requiring compilation, which may not be supported in all cases). Multiformat codes will be included to indicate which formats are in use (see here for a longer description of how this works).

multiencode : Preferences -> DataValue -> Bytestring

Multidecoding

The multidecode

function takes a value in bytestring representation and tries to decode it into an internal representation, according to the multiformat information and the encoding schemes known by the decoding party. Attempting to decode unknown formats will result in an error.

multidecode : Bytestring -> DataType -> Maybe DataValue

Equality

Note that, in general, equality of multiencoded representations implies equality of data values, but equality of data values does not imply equality of multiencoded representations (as different encoding schemes may be used). Phrased more succinctly:

1. multiencode a = multiencode b

→ a = b

1. multiencode a /= multiencode b

does not imply a != b

Usage

In general, canonical commitments to data and code are made over the output of multiencode

.

Implication (1) guarantees that (subject to the usual cryptographic assumptions) equality of two succinct commitments (cryptographic hash function applied to the multiencoded value) implies equality of the data values so encoded and committed to.

Multiencoding is also used before storing data or sending it over the network. Any party who knows the encoding scheme table should be able to take any piece of stored or sent data and deserialise it with multidecode

.

Multievaluation

The multievaluate

function evaluates the application of a function to a value. Whenever multievaluate

encounters a FunctionV n f

, it looks up the appropriate virtual machine as specified by the natural index n

, decodes f

using decode_n

, then calls evaluate_n

, tracking and summing the gas used in subsequent evaluations.

Note: this implies a uniform gas scale, which we elide the details of for now, but would probably require e.g. benchmarking on the hardware in question.

multievaluate : DataValue -> [DataValue] -> Natural -> Maybe (DataValue, Natural)

Recursive multievaluation

What if one VM is passed a data value including a FunctionV

represented in a different VM? The VM in question can treat this code as data - in the sense that it can examine, introspect, modify it, etc. - but it cannot treat this code as code, since it doesn’t know how to interpret functions represented in another VM (in general). However, we can allow one VM to call another easily enough simply by passing a pointer to multievaluate

itself into the evaluation context. Then, when it encounters a function encoded for a different VM which it wishes to evaluate, the VM can simply call multievaluate

, tracking gas consumption as appropriate. This technique can also be used to allow for something such as a data query, where data queries are represented as functions understood by a specific, specialized VM.

Note: there’s some ABI/FFI memory layout logic to be figured out here - multievaluate

must be called with a function and arguments formatted as DataValue

s - this should be specified in detail.

Multicompilation

What if we wish to convert functions between different VM representations? We can define a multicompile

function which attempts to compile functions represented in a data value based on a set of preferences. For example, preferences could be to convert all functions to a particular VM, or to convert where conversions are known in some order of preference. Compilation will fail if unknown VM conversions are attempted.

Multicompilation depends on a known set of conversions compile_{i,j}

which convert between VM_i.t

and VM_j.t

representations of functions, which must preserve extensional equality under evaluation.

multicompile : Preferences -> DataValue -> Maybe DataValue

Equality proofs

With multicompilation, we can create evidence that a = b

, where a

and b

are arbitrary data values, including functions (where multiencode a /= multiencode b

). This evidence would consist simply of a proof that multicompile prefs a = b

for some preferences prefs

. This proof could be of varying practical forms - the verifier could simply run multicompile

themselves, the verifier could trust another’s run of multicompile

, or the verifier could check a succinct proof of computational correctness. Many details are elided here for now.

Smart multievaluation

With multicompilation and equality proofs, we can also define a version of multievaluate

which uses available evidence + preferences intelligently at evaluation time to use certain known-to-be-equivalent versions of functions (e.g. compiled versions) instead of others. Then known optimized versions of functions can be used, and even something like “JIT” compilation can happen e.g. asychronously once certain statistical thresholds are met. Optionally, this “smart multievaluate” can also

use known proofs of the results of certain evaluations (instead of repeating the evaluations), where provable VMs are involved.