Data Provenance Over Computational Graphs

Alan Ransil
Protocol Labs
alan@protocol.ai

Updated September 16, 2023

Abstract

Data-driven decision making requires combining trusted information from many sources. Here, we introduce two protocols which together allow end-to-end tracing of data provenance over computational pipelines. The first protocol, Transform.Storage, models pipelines as symmetric monoidal categories anchored in content-addressed data types. This allows both reproducible and non-reproducible real-world processes to be described as accessible wiring diagrams. Secondly, the Provenance Protocol uses signatures over data and code to evaluate trust with respect to a community. While the two protocols may be used independently, together they form a three-layer system dynamically tracing data provenance. To the extent that trust is compositional, this allows a rigorous examination of the flow of trust through through complex and multi-party computational processes. We explore applications of this paradigm to real-world use cases.

1. Introduction

There is a need to establish the end-to-end reasoning behind data driven decisions, and in particular reasoning incorporating data supplied by many parties. Several trends make this need increasingly urgent. First, the trend towards using public and open source data to substantiate decisions increases the utility of expressing these decisions using transparent end-to-end compute pipelines. Second, decisions are increasingly automated with IOT, smart contract and AI subsystems forming parts of the decision making pipeline. Explicitly recording this pipeline is necessary to substantiate trust in the end result. Third, generative AI heightens the need to rigorously track the provenance of data to combat misinformation. Fourth, decisions in critical areas such as environmental sustainability and AI ethics and safety rely on rapidly evolving research which carries an imperative to explicitly lay out methodologies so that they can be reproduced, challenged, improved, and rapidly applied. Fifth, the introduction of more powerful and general zero knowledge systems increases the necessity of tracing decisions end-to-end so that proven claims relying on little revealed information can be put into context.

Here we introduce two protocols that together can be used to model end-to-end decisions as modular compute pipelines, reproduce their results if sufficient information is revealed, establish trust relative to norms determined by user communities, and allow subsequent users to reuse or extend these pipelines without sacrificing verifiability. These protocols model compute pipelines in three layers as shown in figure 1. The first protocol, Transform.Storage, represents functions as modular, directional relationships between data types. At the data Asset layer, this defines pipelines as typed datasets on a two-dimensional grid along with functional maps between them. At the Function layer, Transform.Storage represents function definitions coupled with input and output types. The correspondence between these layers is determined by the types of Assets on the lower level and the input and output types of Functions at the middle level. The second protocol, the Provenance Protocol, can be used to apply provenance information to data, functions and pipelines and define bounded trust in each relative to community norms. While the two protocols can be used separately, together they can be used to automatically map the flow of trust evaluated relative to a given Provenance community.

2. Previous Work

This section does not attempt a comprehensive literature review of relevant projects. Included are brief descriptions of some of the work which inspired our approach, highlighting some critical differences in design decisions.

2.1. Decentralized Identifiers and Verifiable Credentials

The World Wide Web Consortium (W3C) developed standards for Decentralized Identifiers (DIDs) and Verifiable Credentials (VCs) to allow management of identities and credentials without a centralized registry. Following these standards, a self-sovereign DID may be generated for an individual or organization. The DID controller establishes and signs a DID Document which describes the public keys, authentication and delegation protocols, and endpoints associated with the DID. By separating the DID Document from the controller's key pair, this specification enables a persistent self-sovereign identifier that supports method upgrades and key pair rotation. [4]

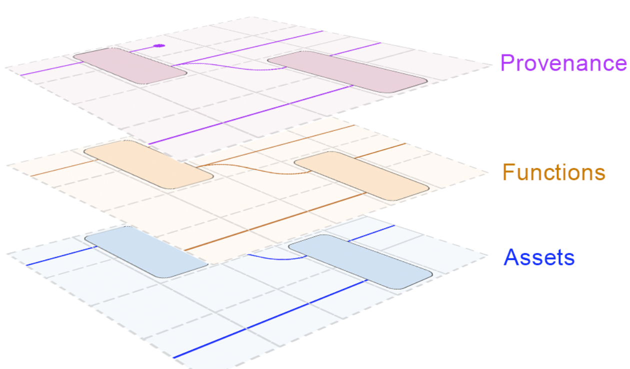


Figure 1: The three layers of a compute pipeline include underlying data assets, functions operating on those assets, and provenance establishing the origins of the lower two layers. Data types establish a correspondence between the asset and function layers, while provenance metadata operating over the lower two layers establishes a flow of trust.

2.2. IPFS

Tracing decisions end-to-end requires interoperating between work done by many individuals on many different computers and subsystems. The protocols typically used to communicate between machines, such as TCP and https, operate on the basis of location addressing. Under this design pattern, information does not have a persistent identifier and interoperability typically relies on centralized services maintaining endpoints which resolve to network locations. Users of these services typically do not control their own key pairs, which disintermediates them from their data. While the results of these design decisions on market power and centralization of economic control are widely discussed, their implications for data management are similarly profound. Because users are expected to rely on companies to maintain the integrity of their data and market incentives induce vendors to lock in users, it is difficult for developers to write code that traces data across the boundaries between software subsystems.

The Interplanetary File System (IPFS) is a protocol which allows users to address and deliver content based on the cryptographic hash of that content rather than its location on the network [2]. This design pattern, namely using a content address as a URI rather than a location address, is essential for maintaining interoperability and will therefore be used as the basis for data addressing in this work.

IPLD

2.3. Wiring Diagrams and Symmetric Monoidal Categories

Wiring diagrams are computational graphs in which data types are represented by lines and functions are represented by boxes. It has been shown that they can be interpreted rigorously as representations of symmetric monoidal categories (SMCs). [3]

A category is a mathematical structure containing objects as well as directional relationships between objects known as morphisms. Morphisms must compose, meaning that if morphisms $f : A \to B$ (morphism f is a directional relationship from object A to object B) and $g : B \to C$ both exist in a category then there must also exist $h : A \to C$ such that $h = f \fatsemi g$. A monoidal category is a category with a product operation, ie. $(A \otimes B)$.

A SMC is a monoidal category with a braiding operation $Braid_{A,B} : A \otimes B \to B \otimes A$ that is symmetric: $Braid_{A,B} \fatsemi Braid_{B,A} = Identity_{A \otimes B}$. If objects in a category are interpreted as data types and morphisms as functions mapping from one datatype to another, then the product in an SMC can be interpreted as an ordered set of two non-interacting data types.

Representing data pipelines in transform.storage as wiring diagrams is provides a formal grounding for computational diagrams that behave intuitively. The type structure makes it possible to define which functions are allowed to be composed with other functions, and by anchoring types using content addresses we are able to achieve interoperability between data storage and compute platforms without sacrificing verifiability. Furthermore, **composition and data provenance together allow a rigorous and automatic assessment of the flow of trust through a compute pipeline.** However, while types in transform.storage should be interpreted strictly composition should not be. In practice, user functions may not be defined over their entire domain (therefore, using the example from above, $h : A \to C$ such that $h = f \fatsemi g$ might not be defined for all values of type A) and may not be deterministic or even formally defined themselves. This means that while pipelines and their component functions are interpreted as morphisms within SMCs, compositionality may be broken when the computation is carried out.

2.4. Provenance Systems

Prov-o, Numbers protocol, ethereum attestations

3. Transform.Storage Data Model

Here, we describe the type structure and data primitives of transform.storage as of the current protocol version.

3.1. Data

Any data blob is a valid member of the data layer in transform.storage. It must be content addressed, allowing it to be verified regardless of the system it is stored in. While the protocol may interpret data blobs (as data conforming to a schema, a schema itself, a function, etc), the contents of data blobs are not legible to the protocol.

3.2. Type

As in type theory, a type in Transform.Storage is a collection of terms sharing a defined set of properties. These may be primitive types such as *string* and *integer*, or more complex constructed types such as that defined by a data schema. If term a is of type A, we express this membership as $a : A$. In general, a given dataset may be a term of multiple types.

Types are expressed in Transform.Storage as either (1) IPLD objects with the canonical attributes described in Table 1, or (2) logically equivalently, the CIDs of these IPLD objects. Individual implementations may add optional fields.

Type Fields	
cid *(CID)*	CID of the data defining this type; for example the CID of a schema. May be null.
type_checking *(string)*	A value from the Type Checking Table, specifying the function f_{check}
creator *(string)*	The public identifier of the creator of this type
creator_id_format *(string, required if Type.creator is non-null)*	Value from the Identifier Table (section 3.5) describing the type of identifier associated with the creator field.
protocol_name *(string)*	Must be "transform.storage"
protocol_version *(string)*	The version of the protocol which this type is pursuant to, such as "2.0.0"
name *(string, optional)*	Short human-readable label
description *(string, optional)*	Longer human-readable label

Table 1: Definitions of the canonical fields describing a Transform.Storage type.

The value of *type_checking* is cross-referenced with the type checking table maintained at the Transform.Storage spec [1], to determine the function f_{check} with the type signature f_{check}: $(\mathit{Type} \otimes \mathit{Data}) \to \{\mathit{result} : \mathit{Boolean}, \mathit{code} : \mathit{Str}\}$. Here, *result* is true if the input *Data* is a term of the input *Type* and false otherwise. The output key *code* contains an error code, or *null*.

Membership in a type is defined by the fields *cid*, and *type_checking*. A simple type is defined by the data in *cid* and the function determined in *type_checking* (Figure 2a). Alternatively, a series type is represented as an array of the CIDs of types (Figure 2b).

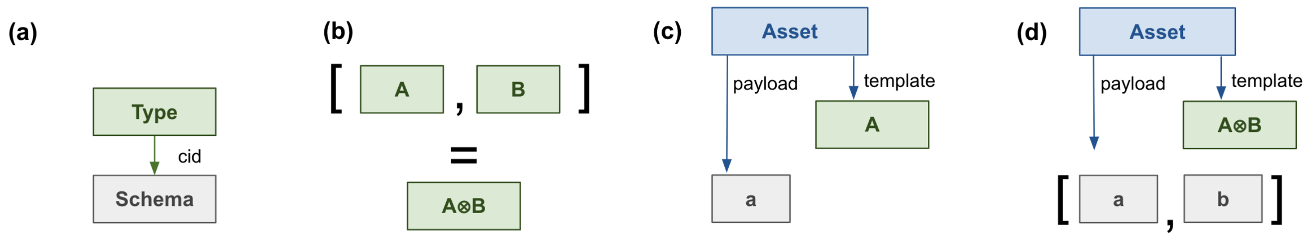


Figure 2: Block diagram showing types and assets in Transform.Storage. Blocks represent IPLD objects, (logically equivalently) CIDs of IPLD objects, or CIDs of data blobs and arrow labels represent keys of objects. Brackets represent IPLD arrays. (a) This simple type is defined by a schema which is referenced in its Type.cid field. (b) The series type $A \otimes B$ is represented as an array of the CIDs of types A and B, and is logically equivalent to the CID of that array. (c) An asset $a : A$. (d) A series asset, showing $[a, b] : A \otimes B$.

Any of the following may thus represent a type:

1. An IPLD object with the fields in table 1, representing a simple type
2. An IPLD array in which values are types, representing a series type
3. The CID of any type; logically equivalent to that type

According to the definition given, types may have complex hierarchical structures which mix objects, arrays, the CIDs of objects, and the CIDs of arrays. However, types may be represented in a normal form by obeying the following rules recursively:

- Expand every type CID: $CID_A \rightarrow A$
- Expand every CID representing an array: $[..., CID_{[A,B,C]}, ...] \rightarrow [..., [A, B, C], ...]$
- Promote the contents of every nested array: $[..., [A, B, C], ...] \rightarrow [..., A, B, C, ...]$

This will result in either a single IPLD object in the case of a simple type, or an array of IPLD objects in the case of a series type. Pseudocode for normalizeType is given in Algorithm 2.

The *height* of a type is 1 in the case of a simple type, or the *length of the normal form* of a series type.

The CID of an array of types is logically equivalent to that array, $CID_{A\otimes B} = [CID_A, CID_B]$ as shown in figure 2b. The CID of an IPLD object is also logically equivalent to that object, so that $CID_{A\otimes B} = A \otimes B$ and $[CID_A, CID_B] = [A, B]$. Note, however, that if the schema of Type C defines an array consisting of the schemas of A and B, this logical equivalence is broken because the protocol does not assess the contents of type schemas. In this situation, we have $\forall [a, b] : A \otimes B, [a, b] : C$ but $A \otimes B \neq C$.

Additionally, *True* and *null* are simple types. For every data blob d, $d : True$. Type *null* has no terms.

Every implementation of transform.storage must include the function isTerm (Algorithm 3), which determines whether a data blob is a term of a given type.

3.3. Asset

An asset is a data structure asserting that a given dataset is a term of a given type. An asset contains the canonical fields shown in table 2.

The block diagrams shown in figure 2 express (c) $a : A$, and (d) $[a, b] : A \otimes B$.

The type referenced by $Asset.template$ does not need to be in a normal form as described in section 3.2 and there are many equivalent representations of any given type. However, the data referenced by $Asset.payload$ is more restricted in that payload CIDs are not hierarchically resolved by the protocol. Thus, if $Asset.payload$ is a CID then either the normal form of $Asset.template$ must be a simple type, or the $Asset.payload$ CID must resolve to an array of data elements matching the $Asset.template$ series type. If $Asset.payload$ is an array then the normal form of $Asset.template$ must be an array with the same number of elements, or $Asset.template$ must be a simple type describing an array. Further hierarchical layers of $Asset.payload$ are not resolved. This is because, while the protocol constrains the structure of types, the contents of data blobs are exogenous to the protocol. If this were not the case, and asset payloads were allowed to contain hierarchical structures similar to series types, it would in principle be impossible for the protocol to distinguish the nested

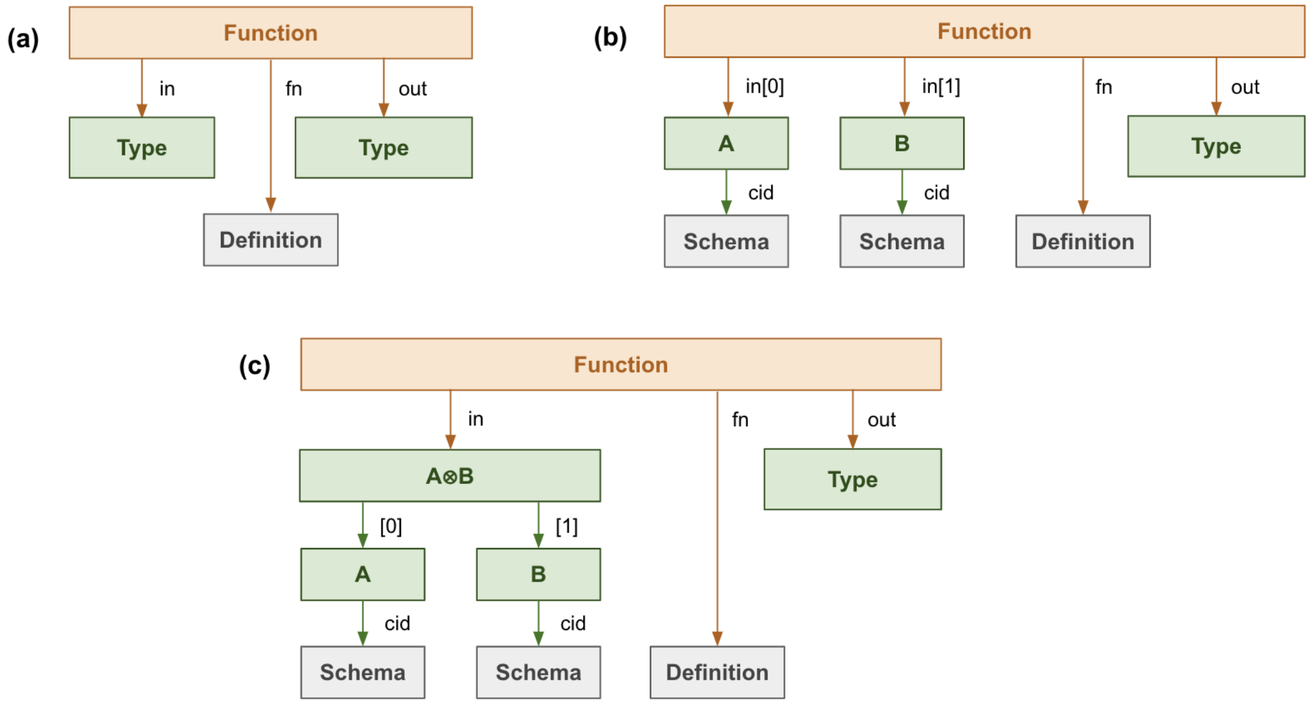


Figure 3: Block diagram showing functions in Transform.Storage. (a) A function consists of input and output types, as well as a function definition which accepts data conforming to the given input types and returns data conforming to the given output types. (b) A function accepting the series type $A \otimes B$, in which this input type is represented in normal form. (c) A function with the same series input as b, where this input is represented in non-normal form.

structure of the payload from any underlying structure of the payload data without constraining the structure of that data. This would violate non-interference (section 3.6)

Pseudocode for *isValidAsset*, a required function which determines whether or not an asset is valid, is given in Algorithm 4.

Asset Fields	
payload *(CID, array, or object)*	The CID of the data referenced by this asset, an array of data CIDs, or the raw data represented as an object or array.
template *(type)*	The transform.storage type referenced by this asset.
creator *(string)*	The public identifier of the creator of this asset

Asset Fields cont'd	
creator_id_format *(string, required if Type.creator is non-null)*	Value from the Identifier Table (section 3.5) describing the type of identifier associated with the creator field.
protocol_name *(string)*	Must be "transform.storage"
protocol_version *(string)*	The version of the protocol which this type is pursuant to, such as "2.0.0"
name *(string, optional)*	Short human-readable label
description *(string, optional)*	Longer human-readable label

Table 2: Definitions of the canonical fields describing a Transform.Storage asset.

3.4. Function

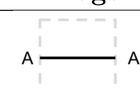
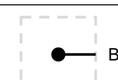
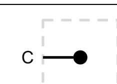
A function is an operation mapping from an input type to an output type. Functions themselves have types. If f is a function mapping from type A to type B, then $f : A \rightarrow B$ ("f is of type A to B"). A function in Transform.Storage contains information about the content addressed input and output types of that function, the function definition, and the execution environment in which the function was tested. These fields are shown in table 3. Values for *Function.execution* and corresponding values for *Function.fn* are given in the Execution Table (Table 4). The protocol includes a set of built-in functions, which correspond to fundamental operations on a computational graph, as well as the ability to externally define functions.

The height of a function is defined as $Max(Function.in.height, Function.out.height)$.

Function Fields	
execution *(string)*	Value chosen from the execution table. Necessary to interpret fn.
fn *(See Table 4)*	Function definition of type chosen from the Execution Table
reference *(string)*	Pointer (CID or URL) to human-readable *fn* code. Not used in execution.
in *(type)*	Input type. May be a CID, an IPLD object or an IPLD array.

Function Fields cont'd	
out *(type)*	Output type. May be a CID, an IPLD object or an IPLD array.
environment *(string)*	Compute environment where this function was tested, chosen from the environment table
env_params *(string)*	Parameters passed to the environment. Format depends on the value of environment.
creator *(string)*	The public identifier of the creator of this type
creator_id_format *(string, required if Type.creator is non-null)*	Value from the Identifier Table (section 3.5) describing the type of identifier associated with the creator field
protocol_name *(string)*	Must be "transform.storage"
protocol_version *(string)*	The version of the protocol which this type is pursuant to, such as "2.0.0"
name *(string, optional)*	Short human-readable label
description *(string, optional)*	Longer human-readable label

Table 3: Definitions of the canonical fields describing a Transform.Storage function.

execution	fn	Description	Image
identity	null	$f(x) = x$	
introduce	CID of a type	$f(null) = x$ in order to introduce x as a constant.	
ignore	null	$f(x) = null$ drop x from the computational graph.	

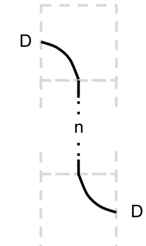
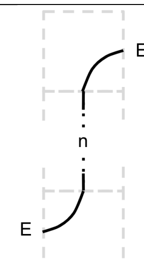
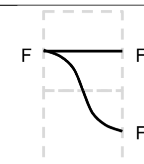
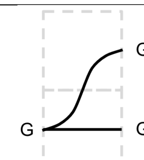
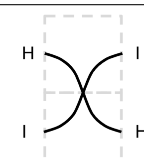
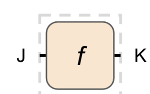
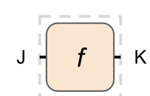
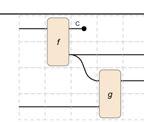
execution cont'd	fn cont'd	Description cont'd	Image cont'd
down	null	$f(D \otimes null... \otimes null) =$ $null... \otimes null \otimes D$ Shifts input down n+1 rows. Height = n+2.	
up	null	$f(null... \otimes null \otimes E) =$ $E \otimes null... \otimes null$ Shifts input down n+1 rows. Height = n+2.	
duplicate down	null	$f(F \otimes null) = F \otimes F$ Duplicates input.	
duplicate up	null	$f(null \otimes G) = G \otimes G$ Duplicates input.	
braid	null	$f(H \otimes I) = I \otimes H$ Switches the order of two inputs.	
WASM	CID of WASM function	$f(J) = K$ Runs function on input J to produce output K. J and/or K may be a series type.	
IPDR	CID of IPDR docker image	$f(J) = K$ Runs function on input J to produce output K. J and/or K may be a series type.	
pipeline	CID of a pipeline	$pipeline(L) = M$ Executes function pipeline on input.	

Table 4: Execution Table. The function definition (Table 3 includes *fn* and *execution* variables defining functions and how they are interpreted. In the case of functions built in to Transform.Storage such as *identity* or *introduce*, the *execution* variable determines the identity of the function parameterized by *fn*. In the case of an externally defined function, *fn* is a pointer to the function and *execution* parameterizes the interpretation of this function.

3.5. Identifier Table

- used to identify a creator

3.6. Non-Interference

The type structure outlined here follows the principle of non-interference: *every layer only references layers below it, never above it.* Thus, data never makes assumptions about type, type does not make assumptions about functions, and no lower level makes assumptions about provenance. This is necessary to ensure interoperability: higher levels are always extensible to include new lower-level cases as long as they can be referenced unambiguously using content addresses.

4. Transform.Storage Pipelines

5. Provenance Protocol

5.1. Provenance Messages

5.2. Provenance Communities

6. Functionality

This section contains pseudocode for functionality in implementations of transform.storage.

6.1. Transform.Storage Data Model Required Functions

Functionality required in any implementation of transform.storage. While these functions respect the logical equivalence between data structures and the CIDs of these data structures, they interoperate between the two on a 'best effort' basis. Some implementations of transform.storage may not have access to a live IPFS node (example: if the implementation itself is on-chain), they may be offline, or the required CID may not be retrievable.

In addition to the result, functions return an error code and documentation of the implementation version of the function. Functions below are labeled with the protocol version described.

6.1.1. isSimpleTypeNormalForm 2.0.0

Accepts a type. Result is a boolean which is true if the input argument is a simple type (ie. not a series type) and is in the normal form.

Protocol implementation maintains $isSimpleTypeNormalForm.version : Str$, and and array of supported $Type.protocol\_version$ values.

Algorithm 1: isSimpleTypeNormalForm :
Type → {*result* : *Boolean*, *code* : *Str*, *protocol* : "*transform.storage*", *protocol_version* : *Str*}

```
input: Type T
suffix ← {protocol: "transform.storage", protocol_version : isSimpleTypeNormalForm.version}
if T is True then
    /* True is a valid simple type */
    return {result: True, code: null, ...suffix}
else if T is False then
    /* False is a valid simple type */
    return {result: False, code: "Type T is False", ...suffix}
else if T is an object then
    if T does not have a key T.protocol with value "Transform.Storage" then
        return {result: False, code: "Type T does not use the Transform.Storage protocol", ...suffix}
    else if T does not have a key T.protocol_version then
        return {result: False, code: "Type T does not list a Transform.Storage protocol version",
          ...suffix}
    else if T.protocol_version is not supported then
        return {result: False, code: "Type T uses Transform.Storage protocol version " +
          T.protocol_version + " not supported by this implementation", ...suffix}
    else if T does not have all the canonical fields for the given protocol version (see table 1) then
        return {result: False, code: 'T does not contain required Type fields for transform.storage
          version " + T.protocol_version, ...suffix}
    /* If T is an object with the required fields of a supported transform.storage
       version, then it is a valid simple type in the normal form. */
    else
        return {result: True, code: null, ...suffix}
    end
else
    return {result: False, code: "T is not an object", ...suffix}
end
```

6.1.2. normalizeType 2.0.0

Accepts a type. Returns the normal form of that type, a boolean indicating whether normalization was successful, and an error code. This function also serves to determine whether an input argument is a valid type.

Protocol implementation maintains *normalizeType.version* : *Str*, and an array of supported versions of *isSimpleTypeNormalForm*.

Algorithm 2: normalizeType :
$Type \rightarrow \{result : Type, success : Boolean, code : Str, protocol : \text{“}transform.storage\text{”}, protocol\_version : Str\}$

```
input: Type T
suffix ← {protocol: “transform.storage”, protocol_version : normalizeType.version}
alreadySimpleNormal ← isSimpleTypeNormalForm(T)
if alreadySimpleNormal.protocol_version not supported then
    return {result: null, success: False, code: “isSimpleTypeNormalForm version” +
     alreadyNormal.protocol_version + “ not supported by normalizeType version ” +
     “normalizeType.version”, ...suffix}
else if isSimpleTypeNormalForm(T).result is True then
    /* If T is a simple type in its normal form, then it is already normalized        */
    return {result: T, success: True, code: null, ...suffix}
else if T is a CID then
    if Have IPFS node then
        /* If T is a CID and this implementation has access to an IPFS node, expand the
           CID and run recursively.                                                    */
        T_Expanded ← Retrieve T from IPFS
        return normalizeType(T_Expanded)
    else
        return {result: null, success: False, code: “Could not expand CID”, ...suffix}
    end
else if T is an array then
    /* If T is an array, then normalize each element of the array                      */
    toReturn ← [ ]
    for ( index i of T )
        elemResult ← normalizeType(T[i])
        if elemResult.success is False then
            return elemResult
        else
            /* If we successfully normalized T[i] then add to toReturn either by
               concatenating (if the normalized form is an array) or appending.        */
            if elemResult.result is an array then
                toReturn ← Concatenate toReturn and elemResult.result
            else
                toReturn ← toReturn + elemResult.result
            end
        end
    end
    return {result: toReturn, success: True, code: null, ...suffix}
else
    /* If T were a simple type, it would either be a CID or in its normal form.  If T
       were a series type, it would either be a CID or an array.  Therefore, at this
       point in the code T must not be a valid type.                                   */
    return {result: null, success: False, code: “T is not a type”, ...suffix}
end
```

6.1.3. isTerm 2.0.0

Accepts a type in its normal form, and a data blob. Returns a boolean which is true if the protocol implementation could determine that data is a valid term of the given type, and false otherwise; and an error code.

Protocol implementation maintains $isTerm.version : Str$, and array of supported $Type.protocol\_version$ values.

Algorithm 3: isTerm :
$(Type \otimes Data) \rightarrow \{result : Boolean, code : Str, protocol : \text{“}transform.storage\text{”}, protocol\_version : Str\}$

```
input: Type T, Data D
suffix ← {protocol: “transform.storage”, protocol_version : isTerm.version}
/* Check for special cases True and False */
if T is True then
    /* Every data blob D is a valid term of type True */
    return {result: True, code: null, ...suffix}
else if T is False then
    /* Type False has no terms */
    return {result: False, code: “Type T is False”, ...suffix}
/* If this is a simple type */
else if T is an object then
    /* T must be a simple type */
    if T.protocol is not “Transform.Storage” then
        return {result: False, code: “Type T does not use the Transform.Storage protocol”, ...suffix}
    end
    if T.protocol_version is not supported then
        return {result: False, code: “Type T uses Transform.Storage protocol version ” +
          T.protocol_version + “ not supported by this implementation”, ...suffix}
    end
    Using the type checking table from the version of the protocol indicated by T.protocol_version, look
     up T.type_checking to determine the type checking function f_check
    return {...f_check(T, D), ...suffix}
/* If this is a series type */
else if T is an array then
    if D is not array with the same length as T then
        return {result: False, code: “D and T length mismatch”, ...suffix}
    end
    for ( index i of T )
        /* T and D are both arrays of the same length, so run recursively on each pair of
           elements */
        if isTerm(T[i], D[i]) returns false then
            return {result: False, code: “D and T mismatch at index ” + i, ...suffix}
        end
        /* At this point in the code, each element of must T match the corresponding
           element of D */
        return {result: True, code: null, ...suffix}
    end
else
    /* T is not in the correct format */
    return {result: False, code: “T is not the normal form of a transform.storage type” , ...suffix}
end
```

6.1.4. isValidAsset 2.0.0

Accepts an asset. Result is true if the protocol implementation could determine that the asset is valid, and false otherwise. Retrieves asset payload CIDs non-recursively, as discussed in section 3.3.

Protocol implementation maintains *isValidAsset.version* : *Str*, and arrays of supported *Asset.protocol_version*, *isSimpleTypeNormalForm*, and *isTerm* values.

Algorithm 4: isValidAsset :
Asset → {*result* : *Boolean*, *code* : *Str*, *protocol* : "*transform.storage*", *protocol_version* : *Str*}

```
input: Asset A
suffix ← {protocol: "transform.storage", protocol_version : isValidAsset.version}
/* If A is the CID of an Asset and this implementation has access to an IPFS node,
   expand the CID and run recursively.                                         */
if A is a CID then
    if Have IPFS node then
        A_Expanded ← Retrieve A from IPFS
        return isValidAsset(A_Expanded)
    else
        return {result: False, code: "Could not expand CID", ...suffix}
    end
/* If A is an object, evaluate it.                                             */
if A is an object then
    if A does not have a key A.protocol with value "Transform.Storage" then
        return {result: False, code: "Asset A does not use the Transform.Storage protocol", ...suffix}
    else if A does not have a key A.protocol_version then
        return {result: False, code: "Asset A does not list a Transform.Storage protocol version",
          ...suffix}
    else if A.protocol_version is not supported then
        return {result: False, code: "Asset A uses Transform.Storage protocol version " +
          A.protocol_version + " not supported by this implementation", ...suffix}
    else if A does not have all the canonical fields for the given protocol version (see table 2) then
        return {result: False, code: "A does not contain required Asset fields for transform.storage
          version " + A.protocol_version, ...suffix}
else
    /* A is an object with the correct fields, so evaluate whether A.payload is a term
       of A.template.                                                          */
    normalizedTemplate ← normalizeType(A.template)
    if normalizedTemplate.protocol_version not supported then
        return {result: False, code: "normalizeType version" + normalizedTemplate.protocol_version +
          " not supported by isValidAsset version " + "isValidAsset.version", ...suffix}
    else if Not normalizedTemplate.success then
        return {result: False, code: normalizedTemplate.code, ...suffix}
    end
    T ← normalizedTemplate.result
```

cont'd on next page

```
...
    ...
        Initialize D
        /* If A.payload is a CID, then retrieve it (exactly once, not recursively) to get
           the data referenced by the asset.  In the edge case that the template schema
           describes a single CID, then this must be one hierarchical level down (ie.
           A.payload must be a CID of a CID).                                          */
        if A.payload is a CID then
            if Have IPFS node then
                D ← Retrieve A.payload from IPFS
            else
                return {result: False, code: "Could not expand A.payload CID", ...suffix}
            end
        /* In the case that both A.payload and T are arrays, this is a series payload.
           Expand every element of the payload if it is a CID, otherwise pass that
           element along unchanged.                                                    */
        else if A.payload is an array and T is an array then
            D ← [ ]
            for ( index i of A.payload )
                if A.payload[i] is a CID then
                    if Have IPFS node then
                        Retrieve A.payload from IPFS and append to D
                    else
                        return {result: False, code: "Could not expand A.payload[" + i + "] CID",
                          ...suffix}
                    end
                else
                    Append A.payload[i] to D
                end
            end
        /* If A.payload is not a CID and T and A.payload are not both arrays, pass
           A.payload along unchanged.                                                  */
        else
            D ← A.payload
        end
    end
    /* Evaluate whether D is a valid term of T                                         */
    validTerm ← isTerm(T, D)
    return {result: validTerm.result, code: validTerm.code, ...suffix}
/* If A is neither a CID nor an object, it is not a valid Asset.                       */
else
    return {result: False, code: "A is not an object", ...suffix}
end
```

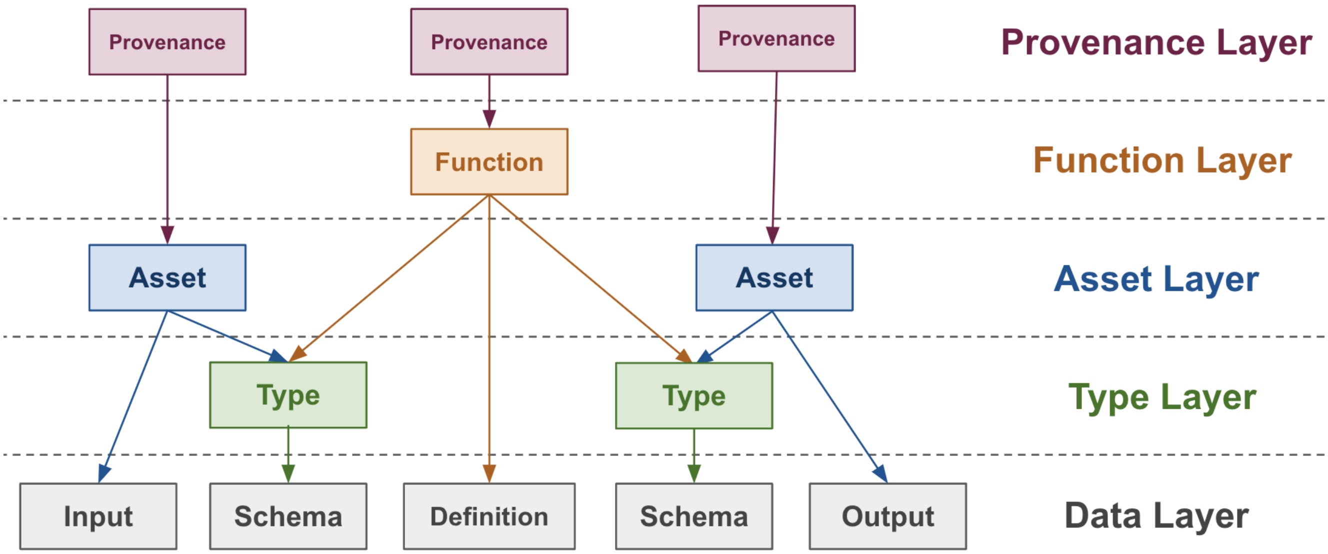


Figure 4: Diagram of a simple compute pipeline consisting of one input asset, one function, and one output asset highlighting relationships between data model components.

References

[1] Transform.storage specification, 2023. Available at `https://spec.transform.storage`.

[2] Juan Benet. Ipfs - content addressed, versioned, p2p file system. *arXiv*, 2014.

[3] Evan Patterson, David I. Spivak, and Dmitry Vagner. Wiring diagrams as normal forms for computing in symmetric monoidal categories. *Electronic Proceedings in Theoretical Computer Science*, 333:49–64, feb 2021.

[4] Manu Sporny, Dave Longley, Markus Sabadello, Drummond Reed, Orie Steele, and Christopher Allen. Decentralized identifiers (dids) v1.0, 2022.