Data Provenance Over Computational Graphs

Alan Ransil
Protocol Labs
alan@protocol.ai

Updated July 27, 2023

Abstract

Data-driven decision making requires trusted information from many sources to be combined using versioned procedures. Here, we introduce two protocols which when used together allow end-to-end tracing of data provenance and trust over computational pipelines. The first protocol, Transform.Storage, models compute 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 multi-layer system dynamically tracing the flow of trust 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.

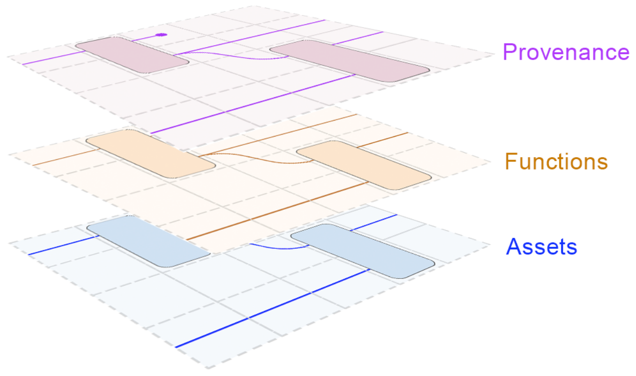

Figure 1: The three layers of a compute pipeline include underlying data assets, functions operating on those assets, and provenance establishing the origins of the lower two layers. Data types establish a correspondence between the asset and function layers, while provenance metadata operating over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

2. 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]

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 Type Structure

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.

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 term may be a member of multiple types.

Types are expressed in Transform.Storage as the CIDs of IPLD objects with the canonical attributes described in Table 1. 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.
series *(array)*	An array of the CIDs of other Types if this a series type. May be null. Must not conflict with Type.cid.
type_checking *(string)*	A value from the Type Checking Table
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.

Membership in a type is defined by the fields *cid*, *series* and *type_checking*. If the type is a series type, it is defined as an ordered series of types listed in *series*. Alternatively, if it is a primitive type, then it is defined by the data in *cid* and the method determined in *type_checking*. The value of *type_checking* can be referenced in the type checking table maintained at the Transform.Storage spec [1].

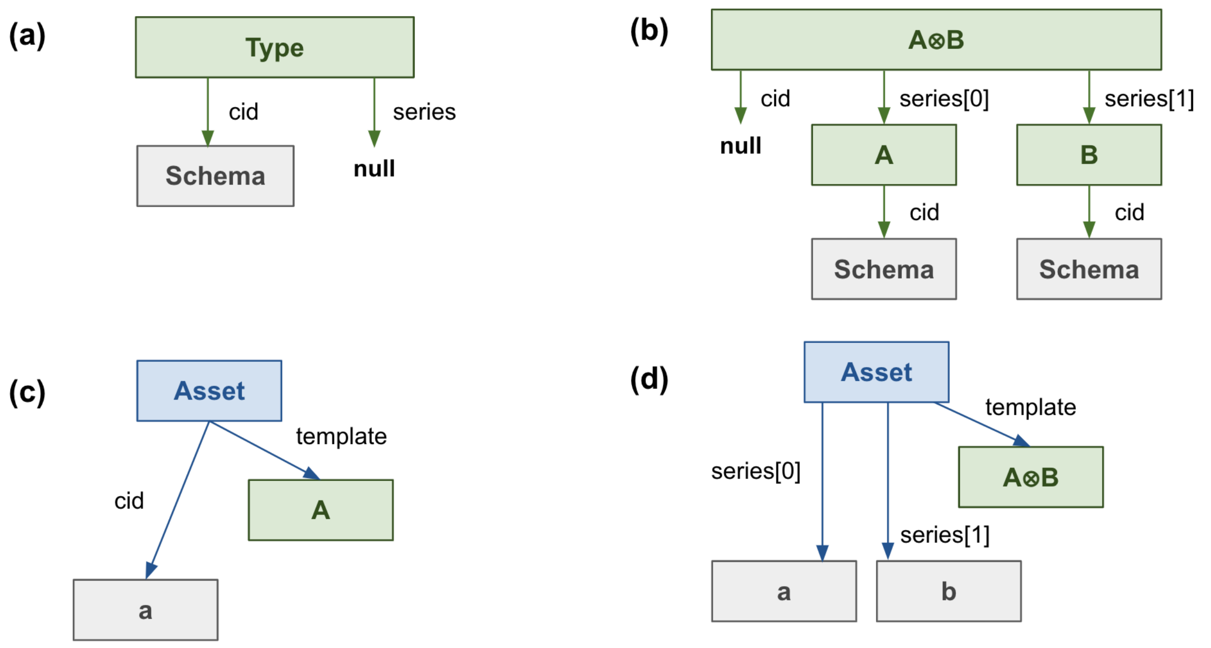

Figure 2: Block diagram showing types and assets in Transform.Storage. (a) this type is defined by a schema which is referenced in its Type.cid field. (b) the series type $A \otimes B$ is defined by two other types referenced in its Type.series field.

Pseudocode for isTerm, a function to determine whether a given dataset is a term of a given type, is provided in Algorithm 1. Because type checking for primitive types depends on looking up the function f_{Check} based on $Type.type\_checking$, the protocol can be extended to support new type systems.

Algorithm 1: isTerm Determine whether data is a term of a given type

```
   input: CID of a Type $T_{CID}$, CID of Data $D_{CID}$
 1 $T \leftarrow$ Retrieve $T_{CID}$ content
 2 $D \leftarrow$ Retrieve $D_{CID}$ content
   /* If this is a series type */
 3 if $T.series$ is not null then
 4     if Type of $D$ is not array then
 5         return False
 6     end
 7     for ( index $i$ of $T.series$ )
 8         Run isTerm recursively on T.series[i], D.series[i]
 9         if isTerm returns false then
10             return False
11         end
       /* We've checked that each element of D.series matches T.series */
12     return True
13 else
       /* If this is a primitive type */
14     Determine type checking function $f_{check}$ based on T.type_checking
15     return $f_{check}(T.CID, D)$
16 end
```

3.3. Asset

An asset is a data structure referencing both a type and a term conforming to that type. For example, the asset shown in figure **??** expresses that $a : A$. An asset contains the following canonical fields.

- **name** (str, optional): short human-readable label
- **description** (str, optional): longer human-readable label
- **cid** (CID): CID of the *data* referenced by this type. May be null.
- **series** (array): an array of the *CIDs of data* if the type represented by this asset is a series type. May be null. May not conflict with Asset.cid.
- **creator** (optional) (str): the public identifier of the creator of this asset.
- **creator_id_format** (str, required if Asset.creator is non-null): value from the Identifier Table (section 3.5) describing the type of identifier associated with the creator field.
- **protocol_name** (str): must be "transform.storage"
- **protocol_version** (str): the version of the protocol which this type is pursuant to, such as "2.0.0"

3.4. Transform.Storage Functions

Any implementation of Transform.Storage must include the following functions relating to the underlying data structures described in this section.

isMember : $(Data \otimes Type) \rightarrow Boolean$ returns *True* if the input value Data is a term of the input value Type, returns *False* otherwise.

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 through content addressing.

4. Transform.Storage Pipelines

5. Provenance Protocol

5.1. Provenance Messages

5.2. Provenance Communities

6. Transform.Storage Pipelines

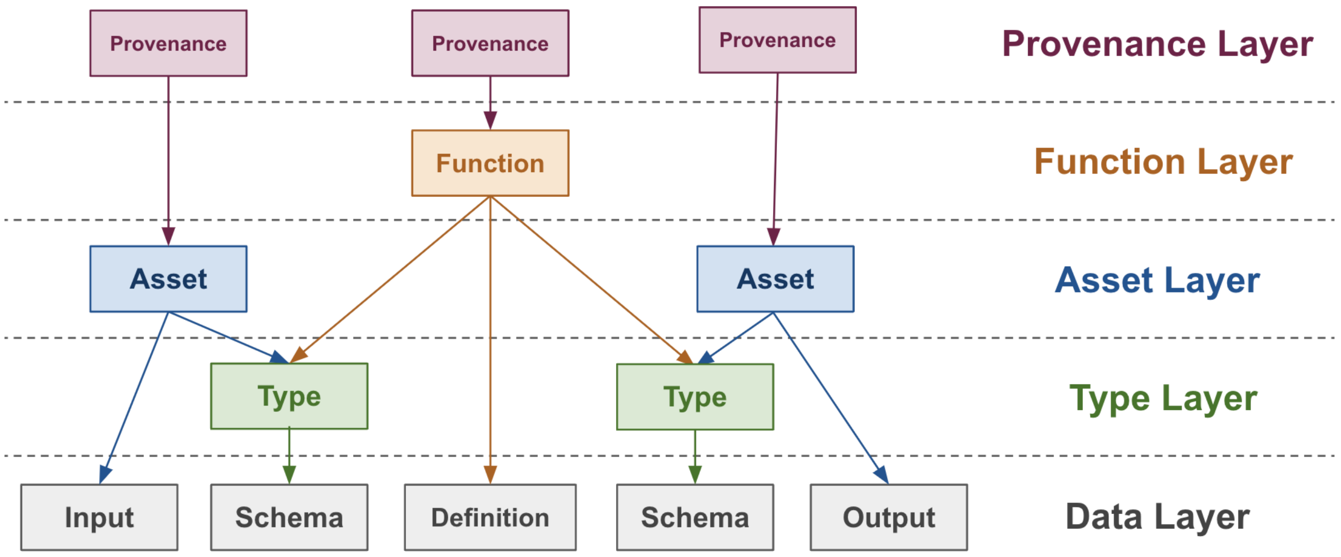

Figure 3: Diagram of a simple compute pipeline consisting of one input asset, one function, and one output asset highlighting relationships between data model components.

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.