Authenticated Data Structures as Functors in Isabelle/HOL

Andreas Lochbihler
Digital Asset, Zurich, Switzerland
andreas.lochbihler@digitalasset.com,mail@andreas-lochbihler.de

Ognjen Marić
Digital Asset, Zurich, Switzerland
ognjen.maric@digitalasset.com

Abstract

Authenticated data structures allow several systems to convince each other that they are referring to the same data structure, even if each of them knows only a part of the data structure. Using inclusion proofs, knowledgeable systems can selectively share their knowledge with other systems and the latter can verify the authenticity of what is being shared.

In this paper, we show how to modularly define authenticated data structures, their inclusion proofs, and operations thereon as datatypes in Isabelle/HOL, using a shallow embedding. Modularity allows us to construct complicated trees from reusable building blocks, which we call Merkle functors. Merkle functors include sums, products, and function spaces and are closed under composition and least fixpoints.

As a practical application, we model the hierarchical transactions of Canton, a practical interoperability protocol for distributed ledgers, as authenticated data structures. This is a first step towards formalizing the Canton protocol and verifying its integrity and security guarantees.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Higher order logic; Theory of computation → Cryptographic primitives

Keywords and phrases Merkle tree, functor, distributed ledger, datatypes, higher-order logic

Supplement Material The formalization is available in the Archive of Formal Proofs [15].

1 Introduction

An authenticated data structure (ADS) allows several systems to use succinct digests to convince each other that they are referring to the same data structure, even if each of them knows only a part of the data structure. This has two main benefits. First, it saves storage and bandwidth, as the systems only have to store parts of the entire structure that they are interested in, and exchange just digests instead of the whole structure. This has been exploited for a wide range of applications, e.g., logs in Certificate Transparency and the blockchain structure and lightweight clients in Bitcoin. Second, ADSs allow parts of the structure to be kept confidential to a subset of the systems involved in processing the structure. For example, distributed ledger technology (DLT) promises to keep multiple organizations synchronized about the state of their joint business workflows. Synchronization requires transactions, i.e., atomic changes to the shared state. Yet organizations often do not want to share all the changes with all involved parties. Some DLT protocols such as the Canton interoperability protocol [6] and Corda [7] leverage ADSs to provide both transactionality and varying levels of confidentiality. The formalization of Canton was the starting point for this work.

Merkle trees [17] are the prime example of an ADS. The original Merkle tree is a binary tree with data at the leaves, where every node is assigned a hash (serving as the digest) using a cryptographic hash function h: a leaf with data d has hash $h\ d$ and an inner node

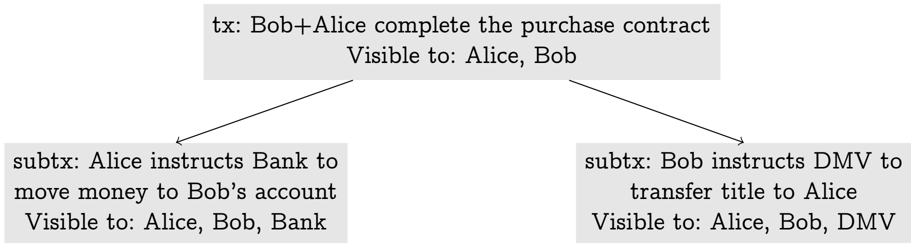

Figure 1 A hierarchical Canton transaction. DMV is the department of motor vehicles.

has the hash $h\ (h_l,\ h_r)$ where h_l and h_r denote the hashes of the two children. If the hash of the root is known to all systems, then one system can convince another that a certain leaf stores data d. If π is the path from the root to the leaf, the inclusion proof consists of the sibling hashes of the nodes on the path. Given such an inclusion proof, the other system can recompute the hashes of the nodes on the path and check that the result matches the common root hash. This shows that the leaf indeed stores the given data if the hash function is collision-resistant. Moreover, the other system learns only hashes (of hashes) of the other data in the tree. So if h is preimage-resistant, then the inclusion proof does not leak information about the rest of the tree, provided that the hashed data contains sufficient entropy. This idea generalizes to finite tree data structures in general [18].

In this work, we consider authenticated data structures, which generalize Merkle trees to arbitrary shape, and we show how to modularly define them as datatypes in Isabelle/HOL. Modularity allows us to construct complicated trees from small reusable building blocks, for which properties are easy to prove. To that end, we consider authenticated data structures as functors and equip them with appropriate operations and their specifications. We show that this class of functors includes sums, products, and function spaces, and is closed under composition and least fixpoints. Concrete functors are defined as algebraic datatypes using Isabelle/HOL's datatype package [1]. This shallow embedding makes it possible to use Isabelle's rich infrastructure for datatypes.

As a practical application, we define ADSs over the hierarchical transactions [3] in the Canton protocol. To see an example of such a transaction, suppose that Alice wants to sell a car title to Bob. Transactionality allows Alice and Bob to exchange the title and the money atomically, which reduces their counterparty risks. Figure 1 shows the corresponding Canton transaction[1] for exchanging the money and the title. The transaction is generated from a smart contract that implements the purchase agreement. Such smart contracts can be conveniently written in the functional programming language DAML [8], which is built on the same hierarchical transactions as Canton.

Canton's hierarchical transactions offer three advantages over conventional flat transactions found in other DLT solutions. First, complex transactions can be composed from smaller building blocks. In the example above, the atomic swap transaction composes two transactions: the money transfer and the title transfer. Second, if a participant is involved only in a subtransaction, then the participant learns the contents of just this subtransaction, but not of other parts. In the example, the Bank only sees the money transfer, but not what Alice bought; similarly, the department of motor vehicles (DMV) does not see the amount the car was sold for. This also improves scalability as everyone must process only

[1] Here and elsewhere in the paper, we take significant liberties in the presentation of Canton and focus on parts relevant for the construction of ADSs and for reasoning about them.

the data they are involved in. Third, they include mandatory authorization checks, which are enforced even in the presence of Byzantine parties. Authorization flows from top to bottom to enable delegation.

This hierarchy, enriched with some additional data, is encoded in ADSs and the protocol exchanges inclusion proofs for such trees. More details will be given throughout the paper. For now, it suffices to summarize the resulting requirements on the formalization:

1. Hashes allow for checking whether two inclusion proofs refer to the same ADS. This allows Canton to commit the example transaction atomically at all participants, even if the Bank and the DMV see only a part of it.
2. Inclusion proofs allow us to prove inclusion for multiple leaves at the same time. Canton sends such inclusion proofs to save bandwidth. Note that conventional inclusion proofs are only for a single leaf.
3. Multiple inclusion proofs can be merged into one if they refer to the same ADS. This is because Canton merges inclusion proofs only if they have the same set of recipients. This reduces the load on the sender because it can multi-cast the same inclusion proof to all recipients. Merging also simplifies the recipients' job: for example, Alice will receive inclusion proofs for the entire transaction as well as both sub-transactions in Figure 1. Merging them leaves her with just a single data structure representing the entire transaction.

Our main contribution is a modular construction principle for ADSs as HOL datatypes, i.e., functors. We also derive a variant of the ADSs that models inclusion proofs. To that end, we introduce the class of Merkle functors, which are equipped with operations for hashing and merging as required above. Our construction is modular in the sense that the class of Merkle functors includes sums, products, and function spaces, and is closed under composition and least fixpoints. Accordingly, the construction works for any inductive datatype (sums of products and exponentials). Moreover, we show that the theory is suitable for constructing concrete real-world instances such as Canton's transaction trees. The construction lives in the symbolic models, i.e., we assume that no hash collisions occur. Our Isabelle/HOL formalization is available in the Archive of Formal Proofs [15].

The rest of the paper is structured as follows. In Section 2, we describe our abstract interface for ADSs. Section 3 shows how to construct such interfaces for tree-like structures in a modular fashion. Section 4 demonstrates how to create inclusion proofs for general rose trees and Canton transactions in particular. We discuss the related work in Section 5 and conclude in Section 6.

2 Inclusion Proofs for Authenticated Data Structures

We now present the operations and abstract interfaces for ADSs, motivated by their application to Canton. Figure 2 shows a suitable a Canton-based deployment, where the Bank and the DMV handle payments and car titles. The participants communicate with each other using the Canton protocol. Unlike in most other DLT solutions, business data resides with Canton primarily at the participants' nodes that share the data only on a need-to-know basis [5]. Canton participants run a two-phase commit protocol to atomically update the system state using transactions. The protocol is run over a Canton domain, which is operated by a third party. The domain acts as the commit coordinator. While the participants may be Byzantine, the domain is assumed to be honest-but-curious. That is, it is trusted to correctly execute the protocol, but it should not learn the contents of a transaction (e.g.,

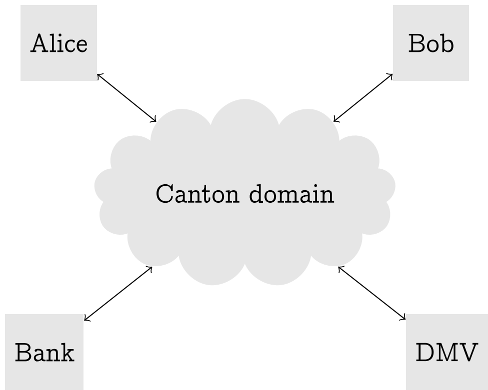

Figure 2 Example topology of a Canton-based distributed ledger

how much Alice pays to Bob). Instead, it should only learn the minimal metadata that allows the protocol to tolerate Byzantine participants. Consequently, Canton sends business data through the domain only in encrypted or hashed form.

This motivates the *transaction tree* structure that Canton uses. The structure for the example transaction from Figure 1 is shown in Figure 3. Each (sub)-transaction of Figure 1 is turned into a *view* in Figure 3, which consists of the view *data* and view *metadata*. For example, the node labeled by 1 in Figure 3 is the view corresponding to the top-level transaction in Figure 1. Its two children that are leaves contain the view's data and metadata. The metadata contains the information about who is affected by the view (here, Alice and Bob) and should therefore participate in the two-phase commit. The metadata is shared with Alice, Bob and the domain. The view data contains the confidential data with the actual state updates, and is shared only with Alice and Bob. This view also has two *subviews*, which correspond to the sub-transactions in Figure 1 as expected. A view can have an arbitrary number of subviews; the views labeled by 1.1 and 1.2 have no subviews, for example.

Additionally, the entire transaction is also equipped with metadata describing transaction-wide parameters, common to all views. Some of it is visible to all the involved participants, but not the domain, and some of it is visible to both the dmain and the participants. The leaf children of the tree's root node store this metadata. Formally, the transaction tree can be modelled by the following datatypes, for some types *common-metadata*, *participant-metadata*, *view-metadata*, and *view-data* whose contents are not relevant for this paper.

datatype *view* =
  View ‹*view-metadata*› ‹*view-data*› (*subviews*: ‹*view list*›)
datatype *transaction* =
  Transaction ‹*common-metadata*› ‹*participant-metadata*› (*views*: ‹*view list*›)

In Figure 3, the *Transaction* and *View* constructors become the inner nodes (black circles) and the data sits at the leaves (grey rectangles).

An ADS over this structure allows the participants and the domain use the root hash to refer to a transaction, and be sure that they are all referring to the same transaction tree. When constructing root hashes, it is useful to think of ADSs with multiple roots (i.e., forests) rather than just a single root like in a Merkle tree. For example, consider how the root node of a binary Merkle tree is constructed from two children. The two children themselves are Merkle trees, so we already have a forest of Merkle trees. More precisely, this forest has the shape of a pair. By adding the root node, we combine the whole forest into a larger Merkle

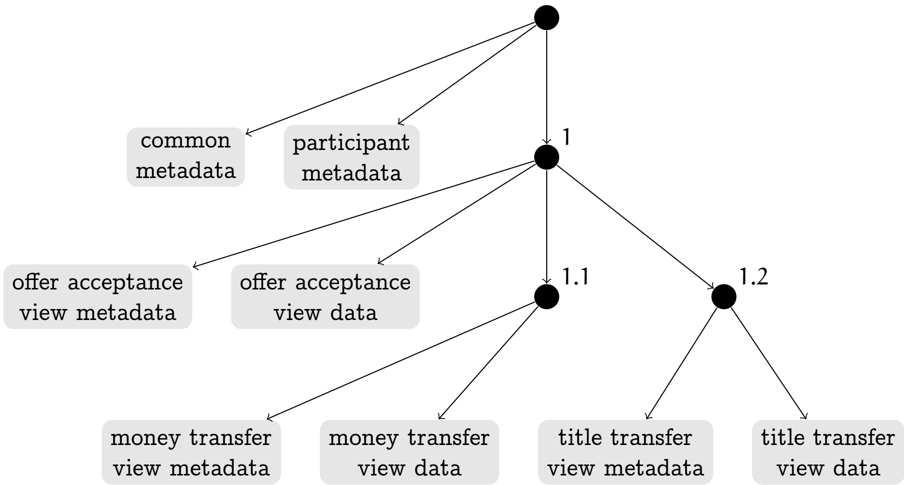

Figure 3 Simplified Canton transaction tree for car title sale of Figure 1

tree. By the construction of Merkle trees, the new root hash is computed solely from the root hashes of the two child trees. Note that the concrete hash operation depends on the shape of the forest (a pair in this case). The new root is again a degenerate forest of a single tree with a single root hash. This view underlies our modular construction principle in Section 3.

In our construction, we will use the following conventions.

1. Raw data to be arranged in an ADSs is written as usual, e.g., $'a$, $'a$ *list*.
2. Hashes and forests of hashes carry a subscript $_\mathrm{h}$ as in $'a_\mathrm{h}$. We leave hashes for now abstract as type variables and define them only in Section 3. Since the root hash identifies an ADS, we represent ADSs by their hashes.

Taking a root hash can make communication more efficient, but it is not enough for our purposes. For example, Bank does not know the contents of view 1.2 or even who is involved in view 1.2; the domain hides the latter. The views that are visible to a participant are called the participant's *projection* of the transaction. Canton aims to achieve the following integrity guarantee [3]: There exists a shared ledger that adheres to the underlying DAML smart contracts such that its projection to each honest participant consists exactly of the updates that have passed the participant's local checks. Achieving this guarantee for the Bank hinges on the Bank's ability to ensure that the view 1.1 is really included in the transaction tree. Thus, we also have to be able to prove that a substructure is included in a root hash.

Inclusion proofs are therefore the main workhorse in our formalization and the focus of this paper. We will denote the type of inclusion proofs over the source type with a subscript $_\mathrm{m}$, e.g., $'a_\mathrm{m}$, $('a_\mathrm{m},\ 'a_\mathrm{h})\ \mathit{tree}_\mathrm{m}$. We need two operations on inclusion proofs:

1. Computing the (forest of) root hashes (which identifies the ADS to which the inclusion proof corresponds).
2. Merging two inclusion proof with the same root hash.

Accordingly, we introduce two type synonyms for these operations:

type_synonym $('a_\mathrm{m},\ 'a_\mathrm{h})\ \mathit{hash} = \langle 'a_\mathrm{m} \Rightarrow\ 'a_\mathrm{h}\rangle$
type_synonym $'a_\mathrm{m}\ \mathit{merge} = \langle 'a_\mathrm{m} \Rightarrow\ 'a_\mathrm{m} \Rightarrow\ 'a_\mathrm{m}\ \mathit{option}\rangle$

The merge operation returns *None* iff the inclusion proofs have different (forests of) root hashes. We require that merging is idempotent, commutative, and associative. The locale *merkle-interface* below captures these properties. Associativity is expressed using the monadic ($\gg=$) on the *option* type. The merge operation makes inclusion proofs with the same hash into a semi-lattice. We fix the induced order as another parameter *bo* of the locale, where an inclusion proof is smaller than another if it reveals less. In that case, we say that the smaller is a *blinding* of the larger inclusion proof.

type_synonym $'a_m$ *blinding-of* $= \langle 'a_m \Rightarrow 'a_m \Rightarrow bool \rangle$
locale *merkle-interface* =
  fixes $h :: \langle ('a_m, 'a_h)\ hash \rangle$
    and $bo :: \langle 'a_m\ blinding\text{-}of \rangle$
    and $m :: \langle 'a_m\ merge \rangle$
  assumes *merge-respects-hashes*: $\langle h\ a = h\ b \longleftrightarrow (\exists\, ab.\ m\ a\ b = Some\ ab) \rangle$
    and *idem*: $\langle m\ a\ a = Some\ a \rangle$
    and *commute*: $\langle m\ a\ b = m\ b\ a \rangle$
    and *assoc*: $\langle m\ a\ b \gg= m\ c = m\ b\ c \gg= m\ a \rangle$
    and *bo-def*: $\langle bo\ a\ b \longleftrightarrow m\ a\ b = Some\ b \rangle$

As expected for a semi-lattice, merging computes the least upper bound in the blinding relation:

$$(m\ a\ b = Some\ ab) = (bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall\, u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u))$$

Also, the equivalence closure of the blinding relation gives the equivalence classes of the inclusion proofs under the hash function: *equivclp bo* = *vimage2p h h* $(=)$ where *equivclp* R denotes the equivalence closure of R and *vimage2p* $f\ g\ R = (\lambda x\ y.\ R\ (f\ x)\ (g\ y))$ the preimage of a relation under a pair of functions.

Our interface does not provide generic operations to build inclusion proofs for subtrees of tree-shaped data. This is because the construction depends on the exact shape of the tree. In Section 4, we will show how to create such proofs for the general shape of rose trees and Canton transactions in particular, using standard functional programming techniques.

3 Modularly Constructing Forests of Authenticated Data Structures

In this section, we develop the theory to modularly construct ADSs and their inclusion proofs as HOL datatypes, including the operations for *merkle-interface*. We first introduce the concept of a blindable position (Section 3.1), which models a node in an ADS, and show how we obtain ADSs for Canton's transaction trees by introducing blindable positions in the right spots of the datatype definitions (Section 3.2).

The specification *merkle-interface* is not inductive and therefore not preserved by datatype constructions. We therefore generalize the specification and show that the generalization is preserved under composition of functors and least fixpoints (Section 3.4). Finally, we show that sums, products and function spaces preserve the generalization (Section 3.5).

3.1 Blindable position

A blindable position represents a node (inner node or leaf) in an ADS. Every node in an ADS comes with its root hash. In this work, we model such hashes symbolically. That is, we assume that no hash collisions occur, i.e., the hash function from values to the type of hashes is injective. We do not assume surjectivity though: some hashes do not correspond to any value. We model such values as garbage coming from a countable set (the naturals). A countable set is large enough given that ADS are always finite in practice (since one cannot compute a hash of infinite amounts of data).

type_synonym *garbage* = ‹*nat*›
datatype $'a_h$ $\mathit{blindable}_h$ = *Content* ‹$'a_h$› | *Garbage* ‹*garbage*›

Since the hash function is injective, we can identify the values $'a$ with a subset of the hashes, namely those of form *Content* _. Accordingly, we could also have written $'a$ $\mathit{blindable}_h$ instead of $'a_h$ $\mathit{blindable}_h$. However, as an ADS contains hashes of hashes, it is more accurate to use $'a_h$ here.

For example, a degenerate Merkle tree with a single leaf, which stores some data x, has the root hash *Content* x. What does an inclusion proof for this tree look like? It can take two forms:

1. The inclusion proof proves inclusion of x, i.e., the leaf is not blinded. The inclusion proof thus contains x.
2. The inclusion proof does not prove inclusion of x, i.e., the leaf is blinded. So the inclusion proof contains only the hash of x.

In the second case, the recipients of such an inclusion proof cannot verify that the hash is meaningful (unless they already know the contents). So the hash could also be garbage. The following datatype formalizes these cases.

datatype ($'a_m$, $'a_h$) $\mathit{blindable}_m$ = *Unblinded* ‹$'a_m$› | *Blinded* ‹$'a_h$ $\mathit{blindable}_h$›

In general, inclusion proofs are nested, e.g., if a Merkle tree leaf contains another Merkle tree as data. We therefore use the inclusion proof type variable $'a_m$ instead of $'a$ for values, similar to $'a_h$ in $\mathit{blindable}_h$,

Note that our hashes are typed. Accordingly, the formalization cannot confuse hashes of ADSs that store *int*s in their leaves with hashes of ADSs that store some other data, say *string*. In the real world, this could happen as hashes are usually just bitstrings. However, for reasoning about inclusion proofs, the garbage hashes adequately model such confusion possibilities: If security best practices are followed, type flaw attacks lead to different hashes unless a hash collision occurs. So the hash of the *int* Leaf would be interpreted as garbage in the type of hashes for the ADS of *string*s. This is adequate for inclusion proofs because we care about the contents of a hash only if the position is unblinded, i.e., of shape *Content*.

Having introduced the types for blindable positions, we now define the corresponding operations and show that they satisfy the specification *merkle-interface*. The hash operation converts an inclusion proof into the root hash of the tree. We define it in two steps:

(i) *hash-blindable'* assumes that there are no nested inclusion proofs, i.e., $'a_m = 'a_h$.
(ii) *hash-blindable* generalizes *hash-blindable'* to nested inclusion proofs. It first converting nested inclusion proofs to their root hashes using the hash function that is given as a parameter. Here, $\mathit{map\text{-}blinddable}_m$ is the mapper generated by the **datatype** command.

primrec *hash-blindable'* :: ‹(($'a_h$, $'a_h$) $\mathit{blindable}_m$, $'a_h$ $\mathit{blindable}_h$) *hash*› **where**
‹*hash-blindable'* (*Unblinded* x) = *Content* x›
| ‹*hash-blindable'* (*Blinded* x) = x›

definition *hash-blindable*
:: ‹($'a_m$, $'a_h$) *hash* ⇒ (($'a_m$, $'a_h$) $\mathit{blindable}_m$, $'a_h$ $\mathit{blindable}_h$) *hash*› **where**
‹*hash-blindable* h = *hash-blindable'* ∘ $\mathit{map\text{-}blindable}_m$ h *id*›

Next, we define the blinding order *blinding-of-blindable*. Like *hash-blindable*, it is parametrized by the hash function and the blinding order for the nested inclusion proofs.

The first clause lifts the blinding order in case the inclusion proof unblinds the contents. The second clause, when the position on the left is blinded, checks that both positions have the same hash.

```
context fixes h :: ‹('a_m, 'a_h) hash› and bo :: ‹'a_m blinding-of› begin
inductive blinding-of-blindable :: ‹('a_m, 'a_h) blindable_m blinding-of› where
  ‹blinding-of-blindable (Unblinded x) (Unblinded y)› if ‹bo x y›
| ‹blinding-of-blindable (Blinded x) t› if ‹hash-blindable h t = x›
end
```

Merging of blindable positions works similarly. If both positions are unblinded, *merge-blindable* tries to merge the contents. If both are blinded, it succeeds iff the hashes are the same. Otherwise, it checks that the hashes are the same and, if so, returns the unblinded version.

```
context fixes h :: ‹('a_m, 'a_h) hash› and m :: ‹'a_m merge› begin
fun merge-blindable :: ‹('a_m, 'a_h) blindable_m merge› where
  ‹merge-blindable (Unblinded x) (Unblinded y) = map-option Unblinded (m x y)›
| ‹merge-blindable (Blinded t) (Blinded u) =
    (if t = u then Some (Blinded u) else None)›
| ‹merge-blindable (Blinded x) (Unblinded y) =
    (if x = Content (h y) then Some (Unblinded y) else None)›
| ‹merge-blindable (Unblinded y) (Blinded x) =
    (if x = Content (h y) then Some (Unblinded y) else None)›
end
```

It is straightforward to show that these definitions preserve the specification *merkle-interface*. That is, if the operations for nested inclusion proofs satisfy *merkle-interface*, then so do the operations for $\mathit{blindable}_{\mathrm{m}}$.

```
lemma merkle-blindable:
  ‹merkle-interface
     (hash-blindable h)
     (blinding-of-blindable h bo)
     (merge-blindable h m)›
  if ‹merkle-interface h bo m›
```

3.2 Example: Canton transaction trees

We now illustrate how to use $\mathit{blindable}_{\mathrm{h}}$ and $\mathit{blindable}_{\mathrm{m}}$ to define the ADSs and inclusion proofs for the Canton transaction trees from Section 2. As shown in Figure 3, the transaction tree contains a node for the transaction tree as a whole, every view, and every leaf (*common-metadata*, *participant-metadata view-metadata*, and *view-data*). Yet, the datatype declarations do not contain the information what should become a separate node in the ADS. To make the construction systematic, we consider the blindable positions to be marked in the datatype with the type constructor *blindable*.

```
type_synonym 'a blindable = ‹'a›
```

So we pretend in this section as if *view*s and *transaction*s were defined as follows:

```
datatype view = View
  ‹((view-metadata blindable × view-data blindable) × view list) blindable›
datatype transaction = Transaction
```

⟨((*common-metadata blindable* × *participant-metadata blindable*) × *view list*) *blindable*⟩

To define the hashes and inclusion proofs, we simply replace each type constructor τ with its counterparts τ_h and τ_m. For views, this looks as follows. Here $\times_h$, $\times_m$, $list_h$, and $list_m$ are type synonyms for × and *list*; Section 3.5 introduces them formally. We abuse notation by writing *view-metadata*$_h$ *view-metadata*$_m$ for the blindable position of *view-metadata*.

type_synonym *view-metadata*$_h$ = ⟨*view-metadata blindable*$_h$⟩
type_synonym *view-data*$_h$ = ⟨*view-data blindable*$_h$⟩
datatype *view*$_h$ = *View*$_h$ ⟨((*view-metadata*$_h$ $\times_h$ *view-data*$_h$) $\times_h$ *view*$_h$ *list*$_h$) *blindable*$_h$⟩

type_synonym *view-metadata*$_m$ = ⟨(*view-metadata*, *view-metadata*) *blindable*$_m$⟩
type_synonym *view-data*$_m$ = ⟨(*view-data*, *view-data*) *blindable*$_m$⟩

datatype *view*$_m$ = *View*$_m$
⟨((*view-metadata*$_m$ $\times_m$ *view-data*$_m$) $\times_m$ *view*$_m$ *list*$_m$,
(*view-metadata*$_h$ $\times_h$ *view-data*$_h$) $\times_h$ *view*$_h$ *list*$_h$) *blindable*$_m$⟩

These types nest hashes and inclusion proofs: A view node, e.g., nests hashes and inclusion proofs for the metadata, the data, and all the subviews. In particular, the *view*$_h$ and *view*$_m$ datatypes recurse through the *blindable*$_h$ and *blindable*$_m$ type constructors. This works because *blindable*$_h$ and *blindable*$_m$ are bounded natural functors (BNFs) [21]. In fact, this transformation works for any datatype declaration thanks to the compositionality of BNFs. The construction for transaction trees is accordingly:

type_synonym *common-metadata*$_h$ = ⟨*common-metadata blindable*$_h$⟩
type_synonym *common-metadata*$_m$ =
⟨(*common-metadata*, *common-metadata*) *blindable*$_m$⟩

type_synonym *participant-metadata*$_h$ = ⟨*participant-metadata blindable*$_h$⟩
type_synonym *participant-metadata*$_m$ =
⟨(*participant-metadata*, *participant-metadata*) *blindable*$_m$⟩

datatype *transaction*$_h$ = *Transaction*$_h$
⟨((*common-metadata*$_h$ $\times_h$ *participant-metadata*$_h$) $\times_h$ *view*$_h$ *list*$_h$) *blindable*$_h$⟩
datatype *transaction*$_m$ = *Transaction*$_m$
⟨((*common-metadata*$_m$ $\times_m$ *participant-metadata*$_m$) $\times_m$ *view*$_m$ *list*$_m$,
(*common-metadata*$_h$ $\times_h$ *participant-metadata*$_h$) $\times_h$ *view*$_h$ *list*$_h$) *blindable*$_m$⟩

3.3 Composition

Having defined the types of ADSs, we next must define the operations on ADSs and prove that they satisfy *merkle-interface*. Doing so directly is possible, but prohibitively cumbersome. Instead, we modularize the proofs following the structure of the types. We can derive preservation lemmas for all involved type constructors analogous to *merkle-blindable*.

The preservation lemmas are compositional by construction: if $'a_h\ \tau_h / 'a_m\ \tau_m$ and $'b_h\ \sigma_h / 'b_m\ \sigma_m$ preserve *merkle-interface*, then so does their composition $'a_h\ \tau_h\ \sigma_h / 'a_m\ \tau_m\ \sigma_m$. Moreover, every nullary functor also satisfies *merkle-interface* with the discrete ordering (=).

definition *merge-discrete* :: ⟨$'a$ *merge*⟩ **where**

‹*merge-discrete x y* = (*if x* = *y then Some y else None*)›
lemma *merkle-discrete*: ‹*merkle-interface id* (=) *merge-discrete*›

For *view-data*, for example, we compose the corresponding discrete functor with a blindable position.

abbreviation *hash-view-data* :: ‹($\mathit{view\text{-}data}_{\mathrm{m}}$, $\mathit{view\text{-}data}_{\mathrm{h}}$) *hash*› **where**
‹*hash-view-data* ≡ *hash-blindable id*›
abbreviation *blinding-of-view-data* :: ‹$\mathit{view\text{-}data}_{\mathrm{m}}$ *blinding-of*› **where**
‹*blinding-of-view-data* ≡ *blinding-of-blindable id* (=)›
abbreviation *merge-view-data* :: ‹$\mathit{view\text{-}data}_{\mathrm{m}}$ *merge*› **where**
‹*merge-view-data* ≡ *merge-blindable id merge-discrete*›

lemma *merkle-view-data*:
‹*merkle-interface hash-view-data blinding-of-view-data merge-view-data*›
by(*rule merkle-blindable*)(*rule merkle-discrete*)

If we do the same for *view-metadata* and consider the pair *view-metadata* × *view-data*, composition immediately gives us the following (the operations for products will be introduced in Section 3.5).

lemma ‹*merkle-interface*
(*hash-prod hash-view-metadata hash-view-data*)
(*blinding-of-prod blinding-of-view-metadata blinding-of-view-data*)
(*merge-prod merge-view-metadata merge-view-data*)›

3.4 Inductive generalization for least fixpoints

The *view* datatype is the least fixpoint of the functor

$$'a\ F = ((\mathit{view\text{-}metadata\ blindable} \times \mathit{view\text{-}data\ blindable}) \times {'a}\ \mathit{list})\ \mathit{blindable}$$

and so are $\mathit{view}_{\mathrm{h}}$ and $\mathit{view}_{\mathrm{m}}$ of analogous functors F_{h} and F_{m}. Composition gives us a preservation theorem for F, but we need more for least fixpoints.

In fact, *merkle-interface* is not inductive, so least fixpoints need not preserve it. The problem is the following: In the inductive preservation proof, we get the induction hypothesis only for smaller values. We therefore cannot use F's preservation theorem because *merkle-interface* requires the conditions to hold on *all* values, not just the smaller ones. So we must generalize *merkle-interface* to make it inductive.

In our first attempt with a direct generalization, the proofs about the merge operation turned out to be rather cumbersome. The associativity law in particular required many case distinctions due to the *option*s. We therefore present a different approach where the focus is on the blinding relation and merge is merely characterized as the join. We abstractly derive commutativity, idempotence, and associativity for merge once and for all from the ordering properties and merge's characterization. This leads to simpler proofs where all case distinctions dealt with by Isabelle's proof automation.

Our generalization splits *merkle-interface* into three locales (Figure 4):

1. The locale *blinding-respects-hashes* splits off the first assumption of *merkle-interface*. No relativization is needed here because the (inductive) blinding order *bo* occurs only once and in a negative position. The preservation proof can therefore use rule induction rather than structural induction.

```
locale blinding-respects-hashes =
  fixes h :: ‹('a_m, 'a_h) hash›
    and bo :: ‹'a_m blinding-of›
  assumes hash: ‹bo ⩽ vimage2p h h (=)›

locale blinding-of-on = blinding-respects-hashes ‹h› ‹bo› for A h bo +
  assumes refl: ‹x ∈ A ⟹ bo x x›
    and trans: ‹⟦ bo x y; bo y z; x ∈ A ⟧ ⟹ bo x z›
    and antisym: ‹⟦ bo x y; bo y x; x ∈ A ⟧ ⟹ x = y›

locale merge-on = blinding-of-on ‹UNIV› ‹h› ‹bo› for A h bo m +
  assumes join: ‹⟦ h x = h y; x ∈ A ⟧
      ⟹ ∃z. m x y = Some z ∧ bo x z ∧ bo y z ∧ (∀u. bo x u ⟶ bo y u ⟶ bo z u)›
    and undefined: ‹⟦ h x ≠ h y; x ∈ A ⟧ ⟹ m x y = None›
```

Figure 4 Inductive generalization of *merkle-interface*

2. The locale *blinding-of-on* formalizes the order properties of the blinding relation *bo* (reflexivity, transitivity, antisymmetry). It fixes a set A in addition to the Merkle operations; the inductive proof for fixpoints instantiates A with the set of smaller terms for which the properties hold by the induction hypothesis. Accordingly, one of the variables in the properties is restricted to A. (Since the induction proof will be structural, it suffices to restrict one variable instead of all.) Unlike *hash* for *blinding-respects-hashes*, transitivity and antisymmetry cannot be shown by rule induction even though *bo* occurs as an assumption, because *bo* occurs multiple times, but rule induction acts only on one. Accordingly, F's preservation theorem does not apply to the induction hypothesis because it assumes that all occurrences are the same.[2]
3. The locale *merge-on* augments *blinding-of-on* with the characterization for merge as the join. While *merge-on*'s assumptions are again restricted by A, the restriction is removed on the assumptions of the parent locale *blinding-of-on* by setting A to the type universe *UNIV*.
   This change is crucial and the reason for introducing three locales: When we prove *join* for the least fixpoint, we can (and must) use that *bo* is an order *everywhere*. This is because *join* uses *bo* with many different arguments, in particular the result z of the merge. In a unified locale, we would have to prove that z stays within the set A, which incurred a lot more proof effort.

In the unrestricted case, *merge-on* is equivalent to *merkle-interface*:

lemma ‹*merkle-interface h bo m* ⟷ *merge-on UNIV h bo m*›

We are now ready to define the class of Merkle functors. For readability, we only spell out the case of unary functors. The generalization to n-ary functors is as expected.

▶ **Definition 1** (Merkle functor). *A unary BNF* F_h *and binary BNF* F_m *constitute a unary Merkle functor if there exist operations hash-F′* :: $(('a_h, 'a_h)\ F_m, 'a_h\ F_h)$ *hash*

[2] Alternatively, we could have generalized the property such that different blinding relations are allowed. Preservation of transitivity becomes preservation of relation composition and antisymmetry transforms into preservation of intersections. For reflexivity, we would still have needed to the set A however.

and $\mathit{blinding\text{-}of\text{-}F} :: ('a_{\mathrm{m}}, 'a_{\mathrm{h}})\ \mathit{hash} \Rightarrow 'a_{\mathrm{m}}\ \mathit{blinding\text{-}of} \Rightarrow ('a_{\mathrm{m}}, 'a_{\mathrm{h}})\ F_{\mathrm{m}}\ \mathit{blinding\text{-}of}$ *and* $\mathit{merge\text{-}F} :: ('a_{\mathrm{m}}, 'a_{\mathrm{h}})\ \mathit{hash} \Rightarrow 'a_{\mathrm{m}}\ \mathit{merge} \Rightarrow ('a_{\mathrm{m}}, 'a_{\mathrm{h}})\ F_{\mathrm{m}}\ \mathit{merge}$ *with the following properties*

Monotonicity $$\frac{bo \leqslant bo'}{\mathit{blinding\text{-}of\text{-}F}\ h\ bo \leqslant \mathit{blinding\text{-}of\text{-}F}\ h\ bo'}$$

Congruence $$\frac{\forall\, a{\in}A.\ \forall\, b.\ m\ a\ b = m'\ a\ b}{\forall\, x \in \{y.\ \mathit{set}_1\text{-}F_{\mathrm{m}}\ y \subseteq A\}.\ \forall\, b.\ \mathit{merge\text{-}F}\ h\ m\ x\ y = \mathit{merge\text{-}F}\ h\ m'\ x\ y}$$

Hashes $$\frac{\mathit{blinding\text{-}respects\text{-}hashes}\ h\ bo}{\mathit{blinding\text{-}respects\text{-}hashes}\ (\mathit{hash\text{-}F}\ h)\ (\mathit{blinding\text{-}of\text{-}F}\ h\ bo)}$$

Blinding order $$\frac{\mathit{blinding\text{-}of\text{-}on}\ A\ h\ bo}{\mathit{blinding\text{-}of\text{-}on}\ \{x.\ \mathit{set}_1\text{-}F_{\mathrm{m}}\ x \subseteq A\}\ (\mathit{hash\text{-}F}\ h)\ (\mathit{blinding\text{-}of\text{-}F}\ h\ bo)}$$

Merge $$\frac{\mathit{merge\text{-}on}\ A\ h\ bo\ m}{\mathit{merge\text{-}on}\ \{x.\ \mathit{set}_1\text{-}F_{\mathrm{m}}\ x \subseteq A\}\ (\mathit{hash\text{-}F}\ h)\ (\mathit{blinding\text{-}of\text{-}F}\ h\ bo)\ (\mathit{merge\text{-}F}\ h\ m)}$$

where $\mathit{hash\text{-}F}\ h = \mathit{hash\text{-}F}' \circ \mathit{map\text{-}F}_{\mathrm{m}}\ h\ \mathit{id}$.

Every Merkle functor preserves *merkle-interface*: set $A = \mathit{UNIV}$ in the merge property and use the above equivalence between *merkle-interface* and *merge-on*.

We are now ready to state and prove the main theoretical contribution of this paper.

▶ **Theorem 2.** *Merkle functors of arbitrary arity are closed under composition and least fixpoints.*

Proof. Closure under composition is obvious from the shape of the properties and the fact that BNFs are closed under composition.

For closure under least fixpoints, we consider a functor F and its least fixpoint T through one of F's arguments. say **datatype** $T = T$ *"T F"*, and similarly for T_{h} and T_{m}. The operations are defined as follows, where we omit all Merkle operation parameters for type parameters that are not affected.

- The hash operation $\mathit{hash\text{-}T}'$ is defined by primitive recursion:

$$\mathit{hash\text{-}T}'\ (T_{\mathrm{m}}\ x) = T_{\mathrm{h}}\ (\mathit{hash\text{-}F}'\ (\mathit{map\text{-}F}_{\mathrm{m}}\ \mathit{hash\text{-}T}'\ x)).$$

- The blinding order *blinding-of-T* is defined inductively by the following rule:

$$\frac{\mathit{blinding\text{-}of\text{-}F}\ \mathit{hash\text{-}T}\ \mathit{blinding\text{-}of\text{-}T}\ x\ y}{\mathit{blinding\text{-}of\text{-}T}\ (T_{\mathrm{m}}\ x)\ (T_{\mathrm{m}}\ y)}$$

  Monotonicity ensures that *blinding-of-T* is well-defined.
- Merge *merge-T* is defined by well-founded recursion (over the subterm relation on T_{m}):

$$\mathit{merge\text{-}T}\ (T_{\mathrm{m}}\ x)\ (T_{\mathrm{m}}\ y) = \mathit{map\text{-}option}\ T_{\mathrm{m}}\ (\mathit{merge\text{-}F}\ \mathit{hash\text{-}T}\ \mathit{merge\text{-}T}\ x\ y)$$

  Congruence ensures that *merge-F* calls *merge-T* recursively only on smaller arguments. We have not been able to define *merge-T* with primitive recursion, which allows pattern matches only on one argument, not two. Our attempts with **primrec** failed because the recursive call occurs under *merge-F*, which is not F_{m}'s mapper. The usual trick of using parametricity theorems to extract the recursive calls into $\mathit{map\text{-}F}_{\mathrm{m}}$ did not work because the parametricity theorem for *merge-F* is too weak. It is also not clear how it could be strengthened without excluding important examples of Merkle functors such as *blindable*.

Well-founded recursion works well, except that Isabelle has no automatic parametricity inference for well-founded recursion. We therefore manually proved the parametricity theorems that the transfer package needs.

Monotonicity and preservation of *blinding-respects-hashes* are proven by rule induction on *blinding-of-T*. Congruence, *blinding-of-on*, and *merge-on* are shown by structural induction on the argument that is constrained by A. ◀

It is not possible to formalize this theorem abstractly in Isabelle/HOL because it is not possible to abstract over type constructors. Instead, we have axiomatized a binary Merkle functor using the **bnf_axiomatization** command and carried out the construction and proofs for least fixpoints and composition. This approach is similar to how Blanchette et al. have formalized the theory of bounded natural functors [2]. The axiomatization also illustrates how the definition and proofs generalize to several functors with type arguments. Moreover, all the example ADS constructions in Section 3.6 merely adapt these proofs to the concrete functors at hand.

3.5 Concrete Merkle functors

We now present concrete Merkle functors. They show that the class of Merkle functors is sufficiently large to be of interest. In particular, it contains all inductive datatypes (least fixpoints of sums of products). We have formalized all of the following.

- The discrete functor from Section 3.3 with hash operation id and the discrete blinding order $(=)$ is a nullary Merkle functor.
- Blindable positions $\mathit{blindable}_{\mathrm{h}}$ and $\mathit{blindable}_{\mathrm{m}}$ are a unary Merkle functor.[3]
- Sums and products are binary Merkle functors. We set $'a_{\mathrm{h}} \times_{\mathrm{h}} {}'b_{\mathrm{h}} = {}'a_{\mathrm{h}} \times {}'b_{\mathrm{h}}$ and $'a_{\mathrm{m}} \times_{\mathrm{m}} {}'b_{\mathrm{m}} = {}'a_{\mathrm{m}} \times {}'b_{\mathrm{m}}$ and similarly for $+_{\mathrm{h}}$ and $+_{\mathrm{m}}$. Formally, $\times_{\mathrm{m}}$ and $+_{\mathrm{m}}$ should take four type arguments. However, as sums and products do not themselves contain blindable positions, the type arguments $'a_{\mathrm{h}}$ and $'b_{\mathrm{h}}$ are ignored and we therefore omit them. The hash operations *hash-prod* and *hash-sum* are the mappers *map-prod* and *map-sum*, respectively. The blinding orders *blinding-of-prod* and *blinding-of-sum* are the relators *rel-prod* and *rel-sum*. The merge operations are defined as follows:

  $$\begin{aligned}&\mathit{merge\text{-}prod}\ \mathit{ma}\ \mathit{mb}\ (x,\ y)\ (x',\ y') =\\ &\mathit{ma}\ x\ x' \ggg (\lambda x''.\ \mathit{map\text{-}option}\ (\mathit{Pair}\ x'')\ (\mathit{mb}\ y\ y'))\end{aligned}$$

  $$\begin{aligned}&\mathit{merge\text{-}sum}\ \mathit{ma}\ \mathit{mb}\ (\mathit{Inl}\ x)\ (\mathit{Inl}\ y) = \mathit{map\text{-}option}\ \mathit{Inl}\ (\mathit{ma}\ x\ y)\\ &\mathit{merge\text{-}sum}\ \mathit{ma}\ \mathit{mb}\ (\mathit{Inr}\ x)\ (\mathit{Inr}\ y) = \mathit{map\text{-}option}\ \mathit{Inr}\ (\mathit{mb}\ x\ y)\\ &\mathit{merge\text{-}sum}\ \mathit{ma}\ \mathit{mb}\ (\mathit{Inr}\ v)\ (\mathit{Inl}\ va) = \mathit{None}\\ &\mathit{merge\text{-}sum}\ \mathit{ma}\ \mathit{mb}\ (\mathit{Inl}\ va)\ (\mathit{Inr}\ v) = \mathit{None}\end{aligned}$$

- The function space $'a \Rightarrow {}'b$ is a unary Merkle functor in the codomain. (Like for sums and products, $'a \Rightarrow_{\mathrm{h}} {}'b_{\mathrm{h}} = {}'a \Rightarrow {}'b_{\mathrm{h}}$ and $'a \Rightarrow_{\mathrm{m}} {}'b_{\mathrm{m}} = {}'a \Rightarrow {}'b_{\mathrm{m}}$ and we omit the ignored $'b_{\mathrm{h}}$.) Hashing is function composition and the blinding order is pointwise. Merge is defined by

[3] The proof of transitivity preservation requires that the blinding order bo on $'a_{\mathrm{m}}$ respects hashes everywhere, not only on A. This is the reason why we have split the locale *blinding-respects-hashes* from *blinding-of-on*.

$\textit{merge-fun}\ m\ f\ g =$
$(\textit{if}\ \forall x.\ m\ (f\ x)\ (g\ x) \neq \textit{None then Some}\ (\lambda x.\ \textit{the}\ (m\ (f\ x)\ (g\ x)))\ \textit{else None})$

Proving the Merkle properties requires choice.

3.6 Case study: Merkle rose trees and Canton's transactions

Thanks to Theorem 2, all datatypes built from the Merkle functors in the previous section are also Merkle functors. We now show the elegance and expressiveness of Merkle functors using three datatypes: lists, rose trees and Canton transaction, where each builds on the previous ones.

Lists are isomorphic to the datatype

datatype $'a\ \textit{list}' = \textit{List}'\ \langle \textit{unit} + {'a} \times {'a}\ \textit{list}' \rangle$

and therefore also a Merkle functor. We have carried out this construction as *list* occurs in Canton transaction trees. Like sums, products, and function spaces, *list*s do not contain blindable positions directly, so $\textit{list}_{\mathrm{h}} = \textit{list}_{\mathrm{m}} = \textit{list}$. Hashing and the blinding order are the mapper and the relator. Initially, we tried to prove *merkle-interface* for *list*s directly, but the proofs about merge quickly got out of control. We therefore carried out the fixpoint construction of Theorem 2 for *list'* and transferred the definitions and theorems to *list* using the transfer package [13].

Rose trees are then given by the datatype

datatype $'a\ \textit{rose-tree} = \textit{Tree}\ \langle({'a} \times {'a}\ \textit{rose-tree list})\ \textit{blindable}\rangle$

Applying our constrution, we obtain Merkle rose trees as

datatype $'a_{\mathrm{h}}\ \textit{rose-tree}_{\mathrm{h}} = \textit{Tree}_{\mathrm{h}}\ \langle({'a_{\mathrm{h}}} \times_{\mathrm{h}} {'a_{\mathrm{h}}}\ \textit{rose-tree}_{\mathrm{h}}\ \textit{list}_{\mathrm{h}})\ \textit{blindable}_{\mathrm{h}}\rangle$
datatype $({'a_{\mathrm{m}}}, {'a_{\mathrm{h}}})\ \textit{rose-tree}_{\mathrm{m}} = \textit{Tree}_{\mathrm{m}}$
$\langle({'a_{\mathrm{m}}} \times_{\mathrm{m}} ({'a_{\mathrm{m}}}, {'a_{\mathrm{h}}})\ \textit{rose-tree}_{\mathrm{m}}\ \textit{list}_{\mathrm{m}}, {'a_{\mathrm{h}}} \times_{\mathrm{h}} {'a_{\mathrm{h}}}\ \textit{rose-tree}_{\mathrm{h}}\ \textit{list}_{\mathrm{h}})\ \textit{blindable}_{\mathrm{m}}\rangle$

with the corresponding operations and their properties.

From here, it is only a small step to transactions in Canton. Views are Merkle rose trees where the data at the nodes is instantiated with the Merkle functor corresponding to *view-metadata blindable* × *view-data blindable*. Then, transactions compose the Merkle functor for *common-metadata blindable* × *participant-metadata blindable* × - *list* with views. We have lifted our machinery from these raw Merkle functors to the datatypes $\textit{view}_{\mathrm{m}}$ and $\textit{transaction}_{\mathrm{m}}$ using the lifting and transfer packages [13].

4 Creating Inclusion Proofs

So far, given a tree-like data type $'t$, we showed how to systematically construct the corresponding type of ADSs $'t_{\mathrm{h}}$ and their inclusion proofs, $'t_{\mathrm{m}}$. To make use of this construction in practice, we must also be able to create values of type $'t_{\mathrm{m}}$ from values of type $'t$. As in the case of our composition and fixpoint theorem, HOL's lack of abstraction over type constructors makes it impossible to express this process in HOL in its full generality. Instead, we show how it works on rose trees, as these are the most general type of tree in terms of branching. The construction can be easily adapted for other kinds of trees.

There are three basic operations:

- Hashing *hash-source-tree* returns the root hash for a source tree.

- Embedding *embed-source-tree* returns the inclusion proof that proves inclusion of the whole tree.
- Fully blinding *blind-source-tree* returns the inclusion proof that proves no inclusion at all.

Hashing and fully blinding conceptually do the same thing, but their return types ($'a_{\mathrm{h}}$ *rose-tree*$_{\mathrm{h}}$ and $('a_{\mathrm{m}},\ 'a_{\mathrm{h}})$ *rose-tree*$_{\mathrm{m}}$) differ. As rose trees are parameterized by their node label type, hashing, embedding and fully blinding take parameters which hash or embed the node labels. The expected properties hold: the embedded and fully blinded versions of the same source tree have the same hash, namely the hashing of the source tree, and the former is a blinding of the latter.

The more interesting operations concern creating an inclusion proof for a subtree of a tree. For example, with Canton's hierarchical transactions, we would like to prove that a subtransaction is really part of the entire transaction. Such a proof consists of the subtree itself, together with a path connecting the tree's root to the subtree's root. As noticed by Seefried [20], this corresponds to a zipper [12] focused on the subtree. This enables simple manipulation of such proofs in a functional programming style, well-suited to HOL. The zippers for rose trees are captured by the following types.

type_synonym $'a$ *path-elem* $= \langle 'a \times\ 'a$ *rose-tree list* $\times\ 'a$ *rose-tree list*$\rangle$
type_synonym $'a$ *path* $= \langle 'a$ *path-elem list*$\rangle$
type_synonym $'a$ *zipper* $= \langle 'a$ *path* $\times\ 'a$ *rose-tree*$\rangle$

Given a zipper that focuses on a node, we define the operations that turn rose trees into zippers and vice-versa

$$\textit{tree-of-zipper}\ ([],\ t) = t$$
$$\textit{tree-of-zipper}\ ((a,\ l,\ r) \cdot z,\ t) = \textit{tree-of-zipper}\ (z,\ \textit{Tree}\ (a,\ l\ @\ t \cdot r))$$

$$\textit{zipper-of-tree}\ t \equiv ([],\ t)$$

The zippers for inclusion proofs have the exact same shape, except that all the type constructors are subscripted by $_{\mathrm{m}}$ and have another type parameter capturing the type of hashes (e.g., $('a,\ 'a_{\mathrm{h}})$ *zipper*$_{\mathrm{m}}$). Like for source trees, we define operations that blind and embed a path respectively, and define operations that convert between Merkle rose trees and their zippers. As expected, given the same source zipper, blinded and embedding its path yield a Merkle rose tree with the same hash. Furthermore, reconstructing a Merkle rose tree constructed by embedding a source zipper gives the same result as first reconstructing the source zipper, and then embedding it into a Merkle rose tree. Finally, we show that reconstruction of trees from zippers respects the blinding relation if the Merkle operations on the labels satisfy *merkle-interface*:

$$\textit{blinding-of-tree}\ h\ bo\ (\textit{tree-of-zipper}_{\mathrm{m}}\ (p,\ t))\ (\textit{tree-of-zipper}_{\mathrm{m}}\ (p,\ t')) =$$
$$\textit{blinding-of-tree}\ h\ bo\ t\ t'$$

Inclusion proofs derived from zippers prove inclusion of a single subtree of the rose tree. When we want to create an inclusion proof for several subtrees, we create an inclusion proof for each individual subtree and then merge them into one. To that end, we have defined the function *zippers-rose-tree* that enumerates the inclusion proof zippers for all nodes of a rose tree.

For Canton's transactions, we have lifted the zippers and their theorems from rose trees to *views*. We define the projection of the inclusion-proof embedded view for one participant P as follows:

1. Enumerate all zippers for the views in the transaction using the lifted version of *zippers-rose-tree*.
2. Each such zipper gives us direct access to the view's metadata. Use the metadata to determine whether P is a recipient of the view. If not, filter out the zipper.
3. Convert the zippers into inclusion proofs for the view and compose each of them with the transaction metadata inclusion proof.
4. Merge all these inclusion proofs into one.

This gives an inclusion proof for the recipient's projection of the transaction. At the end of the two-phase commit protocol, the domains's commit message contains an inclusion proof of the view common data for all the views that the participant should have received. By comparing this inclusion proof against the projection using *blinding-of-transaction*, the participant can decide whether it has received all views it was supposed to receive. (Conversely, checking that it does not receive extraneous views is simple as it can be read from the view metadata.)

5 Related Work

Miller et al. developed a lambda calculus with authentication primitives for generic tree structures [18]. The calculus was formalized in Isabelle/HOL by Brun and Traytel [4]. In the calculus, the programmer annotates the structures with authentication tags. Given a value of such a structure, and a function operating on it, their presented method automatically creates a correctness proof accompanying a result. The proof allows a verifier that holds only a digest of values with authentication tags (but not the values themselves) to check the function's result for correctness. The proof is a stream of inclusion proofs, one for each tagged value that the function operates on. Merging of inclusion proofs is not considered, although the streams can be optimized by sharing. Unlike Brun and Traytel [4] who use a deep embedding with the Nominal library, our embedding is shallow. Furthermore, our ADSs can provide inclusion proofs for multiple sub-structures simultaneously. However, we do not aim to derive correctness proofs for functions on the data structures.

White [22] designed a cryptographic ledger with lightweight proofs of transaction validity and formalized the design in Coq. The ledger is a function from assets to addresses. Transactions move assets between addresses and transform one ledger into another. The transactions' plausibility can be proved by checking that the assets existed in the old ledger and that the assets in the new ledger were moved to the correct place. Ledgers are represented by a tree, where leaves list assets and a tree path encodes an address. A Merkle structure over the tree and Merkle inclusion proofs of the assets' movement relieve the verifiers from having to know the entire ledger. A merge operation allows a single Merkle tree to provide several inclusion proofs. The Coq development is tailored to the use case: the Merkle trees are binary and the leaves are restricted to fixed single type (either asset lists or sentinels that mark empty subtrees). Our generic development can be instantiated to cover this structure.

Yu et al. [23] use Merkle constructions on different binary trees to implement logs with inclusion and exclusion proofs. The constructions are proved correct using a pen-and-paper approach. The proved properties are then used in the Tamarin verification tool to analyze a security protocol.

Ogawa et al [19] formalize binary Merkle trees as used in a timestamping protocol. They automatically verify parts of the protocol using the Mona theorem prover.

Seefried [20] observed that inclusion proofs in a Merkle tree correspond to the derivative of the type, i.e., a Huet-style zipper [12], where the subtrees in zipper context have been replaced by the Merkle root hashes. McBride showed that zippers represent one-hole contexts [16]. In this analogy, our inclusion proofs correspond to contexts with arbitrarily many holes where the subtrees without holes have been replaced by the corresponding hashes. These many-hole zippers must not be confused with Kiselyov's zippers [14] and Hinze and Jeuring's webs [11], which are derived from the traversal operation rather than the data structure .

6 Conclusion and Future Work

We have presented a modular construction principle for authenticated data structures over HOL datatypes (i.e., functors) that have a tree-like shape, and basic operations over these structures. The class of supported functors includes sums, products, and functions, and is closed under composition and least fixpoints. The supported operations are root hash computations and merging of inclusion proofs. We showed how to instantiate the construction to rose trees, as well as to a real-world structure used in Canton, a Byzantine fault tolerant atomic commit protocol.

The ongoing formalization of the Canton protocol will continue to test our abstractions and trigger further improvements. As noted earlier, ADSs cannot only improve storage efficiency, but also provide confidentiality. For example, Canton uses them to keep parts of a transaction confidential to a subset of the transaction's participants. However, reasoning about confidentiality is not straightforward. As hashing is injective, we can simply write $inv\ h\ x$ in HOL to obtain the pre-image of a hash x. In fact, our current model does not even distinguish between the authenticated data structure and its root hash because of this. A sound confidentiality analysis must therefore restrict the adversary using an appropriate calculus, e.g., a Dolev-Yao style deduction relation [9].

In a system, if a source substructure S is unblinded somewhere in an inclusion proof ip, then the confidentiality analysis of the structure should unblind all occurrences of *Blinded* (*Content* S), in ip, regardless of the position where they occur. Our blinding orders and the merge operation do not do this. For example, consider a binary Merkle tree of two leaves that both store a value x. So both leaves have the same hash, and the recipient of an inclusion proof for one leaf detects that the other leaf has the same hash. So they can deduce that the other leaf also contains the value x. Yet, in our blinding order, the inclusion proof for one leaf is strictly smaller than an inclusion proof for both leafs. For proving Canton's integrity guarantees, this is not a problem because confidentiality is not a concern. Moreover, all leaves in the transaction tree contain nonces and the domain checks that all hashes in its inclusion proof are distinct. So the lack of unblinding might not be a problem for reasoning about confidentiality in Canton, even though a proper treatment would simplify the soundness argument.

A related issue is that our modular approach does not apply to commutative structures, such as multisets. The conceptual problem is that the issue with substructures and confidentiality also appears when merging inclusion proofs for commutative structures. One option is consider Merkle functors as quotients with respect to a normalization function that collects all unblinding information and propagates the unblinding across the whole inclusion proof. The normalized inclusion proofs then serve as the canonical representatives. We have not yet worked out whether such a construction can still be modular and whether the quotients are still BNFs [10].

Moreover, our representation of hashes as terms makes hashing injective. While this is "morally equivalent" to standard cryptographic assumptions, an alternative (followed by [4]) would be to prove results about authentication as a disjunction: either the result holds, or a hash collision was found. The advantage of such a statement would be that hash collisions become explicit, which simplifies the soundness argument for the formalization. As is, nothing that prevents us from conceptually "evaluating" the hash function on arbitrarily many inputs, which would not be cryptographically sound. To make hash collisions explicit, we must make hashes explicit, i.e., use a type like *bitstring*s instead of terms. This can be done as additional step.

typedecl *bitstring*
class *encode* =
  fixes *encode* :: ‹$'a \Rightarrow$ *bitstring*›
  assumes *enj-encode*: ‹*inj encode*›

Encoding functions must be defined for all types used as arguments to *blindable*. For *blindable* itself, we then define the actual hash operation as follows.

primrec *root-hash* :: ‹($'a_{\mathrm{h}}$ :: *encode*) $blindable_{\mathrm{h}}$ $\Rightarrow$ *bitstring*› **where**
  ‹*root-hash* (*Garbage garbage*) = *encode-garbage garbage*›
| ‹*root-hash* (*Content x*) = *encode x*›

This can be lifted to forests using the functorial structure of Merkle functors, similar to how $hash\text{-}F\ h = hash\text{-}F' \circ map\text{-}F_{\mathrm{m}}\ h\ id$ first hashes the elements of F using h and then applies the actual function $hash\text{-}F'$. We do not expect problems with extending our constructions to such a model, but it is unclear how severely the indirection through *bitstring*s impacts our proofs, in particular the Canton formalization.

References

1 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP 2014)*, volume 8558 of *LNCS (LNAI)*, pages 93–110. Springer, 2014. `doi:10.1007/978-3-319-08970-6_7`.

2 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Operations on bounded natural functors. *Archive of Formal Proofs*, 2017. `http://isa-afp.org/entries/BNF_Operations.html`, Formal proof development.

3 Sören Bleikertz, Andreas Lochbihler, Ognjen Marić, Simon Meier, Matthias Schmalz, and Ratko G. Veprek. A structured semantic domain for smart contracts. Computer Security Foundations poster session (CSF 2019), `https://www.canton.io/publications/csf2019-abstract.pdf`, 2019.

4 Matthias Brun and Dmitriy Traytel. Generic authenticated data structures, formally. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *Interactive Theorem Proving (ITP 2019)*, volume 141 of *LIPIcs*, pages 10:1—10:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ITP.2019.10`.

5 Canton: A private, scalable, and composable smart contract platform. `https://www.canton.io/publications/canton-whitepaper.pdf`, 2019.

6 Canton: Global synchronization beyond blockchain. `https://www.canton.io/`, 2020.

7 Corda: Open source blockchain platform for business. `https://www.corda.net/`, 2020.

8 Digital Asset. Daml programming language. `https://daml.com`, 2020.

9 D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

10 Basil Fürer, Andreas Lochbihler, Joshua Schneider, and Dmitriy Traytel. Quotients of bounded natural functors. In *IJCAR 2020*, LNCS. Springer, 2020. To appear.

11 Ralf Hinze and Johan Jeuring. Weaving a web. *J. Funct. Program.*, 11(6):681—689, 2001. `doi:10.1017/S0956796801004129`.

12 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549—554, 1997. `doi:10.1017/S0956796897002864`.

13 Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs*, volume 8307 of *LNCS*, pages 131—-146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

14 Oleg Kiselyov. Zippers with several holes. http://okmij.org/ftp/Haskell/Zipper2.lhs, 2011.

15 Andreas Lochbihler and Ognjen Marić. Authenticated data structures as functors. *Archive of Formal Proofs*, April 2020. http://isa-afp.org/entries/ADS_Functor.html.

16 Conor McBride. The derivative of a regular type is its type of one-hole contexts, 2001.

17 Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology (CRYPTO 1987)*, volume 293 of *LNCS*, pages 369–378. Springer, 1987. doi:10.1007/3-540-48184-2_32.

18 Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Principles of Programming Languages (POPL 2014)*, pages 411—423. Association for Computing Machinery, 2014. doi:10.1145/2535838.2535851.

19 Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. Proving properties of incremental Merkle trees. In Robert Nieuwenhuis, editor, *Automated Deduction (CADE 2005)*, volume 3632 of *LNCS*, pages 424–440. Springer, 2005. doi:10.1007/11532231_31.

20 Sean Seefried. Automatically generating tamper resistant data structures. Functional Programming Sydney, http://code.ouroborus.net/fp-syd/past/2017/2017-04-Seefried-Merkle.pdf, 2017.

21 Dmitry Traytel, Andrei Popescu, and Jasmin C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *Logic in Computer Science (LICS 2012)*, pages 596—-605. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.75.

22 Bill White. A theory for lightweight cryptocurrency ledgers. https://github.com/input-output-hk/qeditas-ledgertheory, 2015.

23 Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: a new formalized PKI with no trusted parties. *The Computer Journal*, 59(11):1695–1713, November 2016. arXiv: 1408.1023. URL: http://arxiv.org/abs/1408.1023, doi:10.1093/comjnl/bxw039.