Authenticated Data Structures as Functors in Isabelle/HOL

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

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

Abstract

Merkle trees are ubiquitous in blockchains and other distributed ledger technologies (DLTs). They guarantee that the involved systems are referring to the same binary tree, even if each of them knows only the cryptographic hash of the root. Inclusion proofs allow knowledgeable systems to share subtrees with other systems and the latter can verify the subtrees' authenticity. Often, blockchains and DLTs use data structures more complicated than binary trees; *authenticated data structures* generalize Merkle trees to such structures.

We show how to formally define and reason about authenticated data structures, their inclusion proofs, and operations thereon as datatypes in Isabelle/HOL. The construction lives in the symbolic model, i.e., we assume that no hash collisions occur. Our approach is modular and 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

Digital Object Identifier 10.4230/OASIcs.FMBC.2020.5

Related Version An extended version is available at `www.canton.io/publications/iw2020.pdf`.

Supplementary Material The formalization is available in the Archive of Formal Proofs [18].

1 Introduction

Authenticated data structures (ADSs) allow systems to use succinct digests to ensure that they are referring to the same data structure, even if each system knows only a part of the data structure. The benefits are twofold. First, this saves storage and bandwidth: the systems can store only the structure's parts that are relevant for them, and transmit just digests, not the whole structure. Blockchains use ADSs for this reason, both in the core design and in various optimizations (e.g., Bitcoin's lightweight clients). Second, ADSs can keep parts of the structure confidential to the subset of the systems involved in processing the structure. For example, distributed ledger technology (DLT) promises to keep multiple organizations synchronized on their shared business data. Synchronization requires transactions, i.e., atomic changes to the shared state. Yet organizations often do not want to share their full state with all involved parties. Some DLT protocols such as the Canton interoperability protocol [7] and Corda [8] leverage ADSs to provide both transactions and varying levels of confidentiality. Formal reasoning about blockchains and DLTs thus often requires mechanised theories of ADSs. In fact, the formalization of Canton was the starting point for this work.

2nd Workshop on Formal Methods for Blockchains (FMBC 2020).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 5; pp. 5:1–5:14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

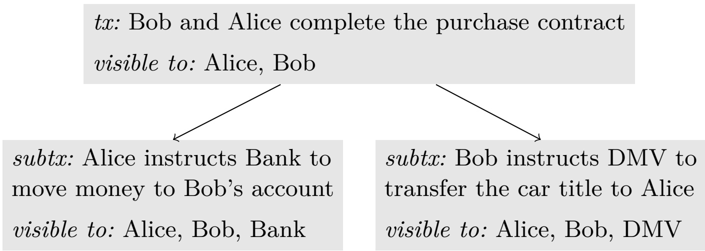

Figure 1 A hierarchical Canton transaction. DMV is the department of motor vehicles.

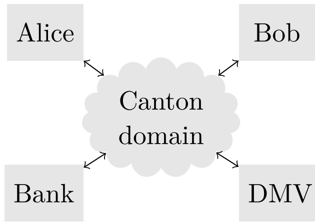

Figure 2 Example topology of a Canton-based distributed ledger

Merkle trees [20] are the prime example of an ADS. They are binary trees of digests, i.e., cryptographic hashes. Leaves contain data hashes, and inner nodes combine their children's hashes using a hash function h. An *inclusion proof*, also known as a Merkle proof, shows that a tree t includes a subtree st. It consists of the roots of t and st and the siblings of nodes on the path between these roots. The proof is valid if the hash of every node on the path is h of the children's hashes. It is sound, i.e., does prove inclusion, if h is collision-resistant. It keeps the rest of the tree confidential if h is preimage-resistant and the hashed data contains sufficient entropy.

ADSs [21] generalize these ideas to arbitrary finite tree data structures, whose hierarchies can conveniently encode more complex relationships between data. Our main example are the hierarchical transactions [4] in the Canton protocol. Suppose that Alice wants to sell a car title to Bob. Figure 1 shows the corresponding Canton transaction for exchanging the money and the title. (We take significant liberties in the presentation of Canton in this paper and focus on parts relevant for the construction of ADSs and for reasoning about them.) The transaction is generated from a smart contract (written in the DAML [10] programming language) implementing the purchase agreement.

The transactions' hierarchical nature benefits Canton in three crucial ways. First, complex transactions can be composed from simpler building blocks, which are transactions themselves. The purchase transaction above composes two such sub-transactions: the money transfer and the title transfer. Second, participants learn only the contents of subtransactions they are involved in. Above, the Bank only sees the money transfer, but not what Alice bought; similarly, the DMV does not learn the car's price. This also improves scalability, as everyone processes only the subtransactions they are involved in. Third, the hierarchy enables correct delegation in Canton's built-in authorization logic even in a Byzantine setting. Canton encodes this hierarchy, enriched with some additional data, in ADSs, and exchanges inclusion proofs for subtransactions. We give more details throughout the paper, but summarize the resulting requirements on the formalization here:

1. It must support ADS digests, to check that two inclusion proofs refer to the same ADS. This allows the example transaction to commit atomically, even if the Bank and the DMV see only a part of it.
2. Proofs must enable proving inclusion for multiple subtrees simultaneously, not just single subtree as standard. Canton uses such inclusion multi-proofs to save bandwidth.
3. Inclusion proofs refering to the same ADS must be mergeable into one multi-proof. In the example of Figure 1, Alice receives inclusion proofs for the entire transaction as well as both sub-transactions, and merges them to a single data structure, the entire transaction.

In this work, we show how to modularly define ADSs as datatypes in Isabelle/HOL. The modular approach is our main theoretical contribution. It 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 so-called *Merkle functors* and equip them with appropriate operations and their specifications. The class of Merkle functors includes sums, products, and function spaces, and is closed under composition and least fixpoints. Hence, the construction works for any inductive datatype (sums of products and exponentials). Concrete functors are defined as algebraic datatypes using Isabelle/HOL's datatype package [3]. This shallow embedding is a significant practical benefit, as it enables the use of Isabelle's rich reasoning infrastructure for datatypes. The construction lives in the symbolic model, i.e., we assume that no hash collisions occur. Finally, we show that the theory is suitable for constructing concrete real-world instances such as Canton's transaction trees. Our formalization is available in the Isabelle AFP [18].

The rest of the paper is structured as follows. In Section 2, we provide the background on Canton and use it to motivate 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 Operations on Authenticated Data Structures

We now present the interfaces for ADSs, motivated by their application to Canton. Figure 2 shows a suitable Canton-based deployment for our example transaction. The participants transact using Canton, a distributed commit protocol similar to a two-phase commit protocol. The protocol is run over a Canton *domain* operated by a third party that 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., how much Alice pays to Bob). Unlike in most other DLT solutions, participants share business data only on a need-to-know basis [6]. In particular, the domain receives business data only in encrypted form or as a digest. The domain may only learn the metadata that allows the protocol to tolerate Byzantine participants.

These privacy requirements motivate the hierarchical transactions that Canton uses, which are encoded in *transaction trees*. The tree 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 first two children contain the view's data and metadata. The metadata lists who is affected by the view and should therefore participate in the commit protocol (here, Alice and Bob), and 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. Views can have an arbitrary number of subviews; e.g., the views labeled by 1.1 and 1.2 have no subviews.

Additionally, the two leaf children of the tree root store metadata describing transaction-wide parameters that apply to all views. The first is visible to the domain and the participants involved in the transaction; the second only to the latter. 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 irrelevant for this paper.

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

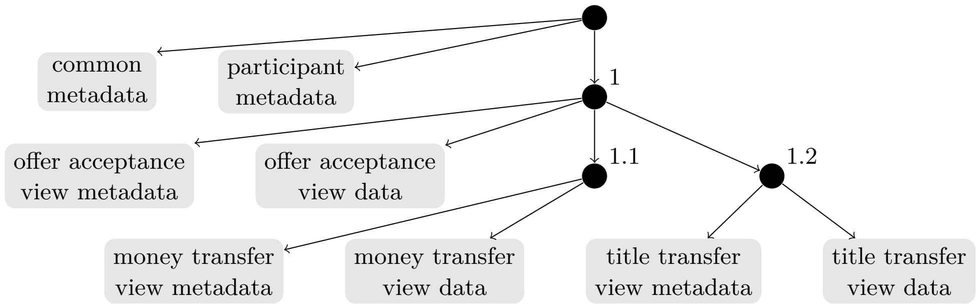

Figure 3 Simplified Canton transaction tree for car title sale of Figure 1

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

The participants and the domain can use a root hash of an ADS over a *Transaction* to ensure that they are all referring to the same transaction tree. When constructing ADS hashes, we need to consider ADSs with multiple roots (i.e., forests) rather than just a single root like in a Merkle tree. For example, computing the hash of an inner node in a Merkle tree requires taking a hash over both of its children, i.e., over the forest constructed from its two children. 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 this paper, we use the following Isabelle notations: Type variables $'a$, $'b$ are prefixed by $'$ like in Standard ML. Type constructors like *list* are usually written postfix as in *string list*. Exceptions are the function space $\Rightarrow$, sums $+$, and products $\times$, all written infix. The notation $t :: \tau$ denotes that the term t has the type τ. In our construction, we will use the following decorations. Raw data to be arranged in an ADS is written as usual, e.g., $'a$, $'a$ *list*. Hashes and forests of hashes carry a subscript $_h$ as in $'a_h$. 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.

A root hash makes communication more efficient, but we require more. For example, the Bank does not know the contents or participants of view 1.2; the domain hides the latter. Still, the Bank must ensure that the view 1.1 is really included in the transaction tree. In general, the views visible to a participant are called the participant's *projection* of the transaction. Canton aims to achieve the following integrity guarantee [4]: 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. This requires the ability 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 denote the type of inclusion proofs over the source type with the subscript $_m$, e.g., $'a_m$, $('a_m, 'a_h)\ tree_m$. We need two operations on inclusion proofs:

1. Computing the (forest of) root hashes of an inclusion proof, in order to identify the ADS to which the inclusion proof corresponds.
2. Merging two inclusion proofs with the same root hash.

Accordingly, we introduce two type synonyms for these operations:

type_synonym $('a_m, 'a_h)\ hash = \langle 'a_m \Rightarrow 'a_h \rangle$
type_synonym $'a_m\ merge = \langle 'a_m \Rightarrow 'a_m \Rightarrow 'a_m\ option \rangle$

We model the merge operation as a partial function using the *option* that returns *None* iff the inclusion proofs have different (forests of) root hashes. We require that merging is idempotent, commutative, and associative. The merge operation makes inclusion proofs with the same hash into a semi-lattice, where the induced order treats an inclusion proof as 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* $= \langle 'a_m \Rightarrow 'a_m \Rightarrow bool \rangle$

▶ **Definition 1.** *A* Merkle interface *consists of three operations* $h :: ('a_m, 'a_h)$ *hash and* $m :: 'a_m$ *merge and* $bo :: 'a_m$ *blinding-of with the following properties:*

1. *Merge respects hashes, i.e.,* $(h\ a = h\ b) = (\exists\, ab.\ m\ a\ b = Some\ ab)$.
2. *Merge is idempotent, i.e.,* $m\ a\ a = Some\ a$.
3. *Merge is commutative, i.e.,* $m\ a\ b = m\ b\ a$.
4. *Merge is associative, i.e.,* $m\ a\ b \gg= m\ c = m\ b\ c \gg= m\ a$, *where* $(\gg=)$ *is the monadic bind on the option type.*
5. *Blinding is induced by merge, i.e.,* $bo\ a\ b = (m\ a\ b = Some\ b)$.

So merge is 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.

Isabelle/HOL's term language is not expressive enough to automatically create the ADS and inclusion proof types of arbitrary tree-shaped data, define the interface's operation, or build inclusion proofs for subtrees of tree-shaped data. Instead, in the next two sections, we show how to systematically construct these types and operations.

3 Modularly Constructing Forests of Authenticated Data Structures

In this section, we develop the theory to modularly construct ADSs, their inclusion proofs as HOL datatypes, and Merkle interfaces over them. We start with the concept of a blindable position (Section 3.1), which models an ADS node, 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).

We have shown how the Merkle interface specification is preserved by type composition (Section 3.3). It is, however, not inductive and therefore not preserved by datatype constructions. We thus generalize it and show that functor composition and least fixpoint preserve the generalization (Section 3.4). Finally, we show that sums, products and function spaces preserve the generalization (Section 3.5) and compose these preservation results to obtain the Merkle interface properties for Canton transactions (Section 3.6).

3.1 Blindable position

A *blindable position* represents a node (inner node or leaf) in an ADS. Recall that "blinding" allows an inclusion proof to hide the node contents by using just the root hash of the node. In this work, we model such hashes symbolically, that is, as injective functions, and assume that no hash collisions occur. 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). This suffices as digests contain only a finite amount of information.

datatype $'a_h$ *blindable*$_h$ = *Content* ⟨$'a_h$⟩ | *Garbage* ⟨*nat*⟩

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 $'a$ *blindable*$_h$ instead of $'a_h$ *blindable*$_h$. However, as an ADS contains hashes of hashes, $'a_h$ is more accurate 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. Either it reveals x, i.e., the leaf is not blinded, or it does not reveal x, i.e., the leaf is blinded. The following datatype formalizes these cases.

datatype ($'a_m$, $'a_h$) *blindable*$_m$ = *Unblinded* ⟨$'a_m$⟩ | *Blinded* ⟨$'a_h$ *blindable*$_h$⟩

Similar to *blindable*$_h$, inclusion proofs may be 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 $'a$. In the second case, the hash could be garbage, so we use $'a_h$.

Note that our *blindable*$_h$ hashes are typed: hashes of those ADSs that store *int*s and those that store *string*s in their leaves always differ. In the real world, they can be equal as hashes are just bitstrings. However, for systems which follow security best practices, type flaw attacks lead to different hashes unless a hash collision occurs. Garbage hashes adequately model such confusion possibilities: a hash of the *int* Leaf would be treated 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 and thus 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 *hash-blindable* :: ($'a_m$, $'a_h$) *hash* ⇒ (($'a_m$, $'a_h$) *blindable*$_m$, $'a_h$ *blindable*$_h$) *hash* converts an inclusion proof into the root hash of the tree. It is parameterized by a hash function h_a that converts nested inclusion proofs $'a_m$ into their root hashes $'a_h$. Its definition is straightforward: for unblinded nodes, apply h_a, and for blinded nodes, just take the contained hash. Similarly, the blinding order *blinding-of-blindable* :: ($'a_m$, $'a_h$) *hash* ⇒ $'a_m$ *blinding-of* ⇒ ($'a_m$, $'a_h$) *blindable*$_m$ *blinding-of* is parametrized by the hash h_a and the blinding order bo_a for the nested inclusion proofs, as well as the blindable inclusion proofs to be compared. If both of the compared inclusion proofs unblind the contents, then we compare the contents using bo_a. Otherwise, the first argument is a blinding of the second one only if it is blinded, and if its hash matches the hash of the second argument. Merging of blindable positions is also similar. 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. It is straightforward to show the following lemma.

▶ **Lemma 2.** *If h_a, bo_a, and m_a jointly form a Merkle interface, then so do hash-blindable h_a, blinding-of-blindable h_a bo_a, and merge-blindable h_a m_a.*

3.2 Example: Canton transaction trees

We now illustrate how to use *blindable*$_h$ and *blindable*$_m$ 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 start from an isomorphic representation of *view* and *transaction*, where we mark the blindable positions with the type constructor *blindable*, which is just the identity functor:

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 $\textit{view-metadata}_h$ and $\textit{view-metadata}_m$ for the blindable position of *view-metadata*.

type_synonym $\textit{view-metadata}_h$ = ‹*view-metadata* $\textit{blindable}_h$›
type_synonym $\textit{view-data}_h$ = ‹*view-data* $\textit{blindable}_h$›
datatype $\textit{view}_h$ = $\textit{View}_h$ ‹(($\textit{view-metadata}_h \times_h \textit{view-data}_h) \times_h \textit{view}_h\ \textit{list}_h$) $\textit{blindable}_h$›
type_synonym $\textit{view-metadata}_m$ = ‹(*view-metadata*, *view-metadata*) $\textit{blindable}_m$›
type_synonym $\textit{view-data}_m$ = ‹(*view-data*, *view-data*) $\textit{blindable}_m$›
datatype $\textit{view}_m$ = $\textit{View}_m$
‹(($\textit{view-metadata}_m \times_m \textit{view-data}_m) \times_m \textit{view}_m\ \textit{list}_m$,
($\textit{view-metadata}_h \times_h \textit{view-data}_h) \times_h \textit{view}_h\ \textit{list}_h$) $\textit{blindable}_m$›

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 $\textit{view}_h$ and $\textit{view}_m$ datatypes recurse through the $\textit{blindable}_h$ and $\textit{blindable}_m$ type constructors. This works because $\textit{blindable}_h$ and $\textit{blindable}_m$ are bounded natural functors (BNFs) [3]. In fact, this transformation works for any datatype declaration thanks to the compositionality of BNFs. The construction for transaction trees is similar.

3.3 Composition

Having defined the types of ADSs, we next must define the operations on ADSs and prove that they form a Merkle interface. Doing so directly is possible, but prohibitively complex. 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 $'a_h\ \tau_h/('a_m,\ 'a_h)\ \tau_m$ and $'b_h\ \sigma_h/('b_m,\ 'b_h)\ \sigma_m$ satisfy *merkle-interface*, then so does their composition $'a_h\ \tau_h\ \sigma_h/(('a_m,\ 'a_h)\ \tau_m,\ 'a_h\ \tau_h)\ \sigma_m$. For example, we can define the instance for blindable nodes of type *view-data* compositionally. First, we exploit the fact that every nullary functor satisfies *merkle-interface* with the discrete ordering (=), hash *id* and *merge* defined only for equal operands. Second, we compose *view-data*, viewed as a nullary functor with *blindable*. For example, we define:

abbreviation *hash-view-data* :: ‹($\textit{view-data}_m$, $\textit{view-data}_h$) *hash*› **where**
‹*hash-view-data* ≡ *hash-blindable id*›

We perform the same constructions on *view-metadata*, and then use composition for the pair *view-metadata* × *view-data*, to get the following (the operations for products will be introduced in Section 3.5).

▶ **Lemma 3.** *The following three operations form a 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

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

and so are $view_h$ and $view_m$ of analogous functors F_h and F_m. Composition gives us a preservation theorem for F, but we need another one for least fixpoints.

Yet, the Merkle interface specification is not inductive and thus not preserved by fixpoints. We now generalize it. Simultaneously, we make the generalization more amenable to Isabelle's proof automation by focusing on the blinding order and characterizing merge as its join. Our generalization splits the Merkle interface into three:

1. The interface *blinding-respects-hashes* assumes that $bo \le \textit{vimage2p}\ h\ h\ (=)$ where $(\le)$ denotes inclusion on binary predicates.
2. The interface *blinding-of-on* formalizes the order properties of the blinding relation bo: Reflexivity $bo\ x\ x$, transitivity $bo\ x\ y \Longrightarrow bo\ y\ z \Longrightarrow bo\ x\ z$, and antisymmetry $bo\ x\ y \Longrightarrow bo\ y\ x \Longrightarrow x = y$ hold for all $x \in A$ and all y, z: The restriction of x to the set A makes the statement inductive, as A can be instantiated to the set of smaller values in structural induction proofs.
3. The interface *merge-on* extends *blinding-of-on* applied to the type's universal set $UNIV$ with the characterization of merge as the join, but now again restricted by a set A. In the unrestricted case $A = UNIV$, *merge-on* is equivalent to the Merkle interface.

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 4** (Merkle functor). *A unary BNF F_h and binary BNF F_m constitute a unary Merkle functor if there exist operations:*

- $\textit{hash}'_F :: (({'a_h}, {'a_h})\ F_m, {'a_h}\ F_h)\ \textit{hash}$ *and*
- $\textit{blinding-of}_F :: ({'a_m}, {'a_h})\ \textit{hash} \Rightarrow {'a_m}\ \textit{blinding-of} \Rightarrow ({'a_m}, {'a_h})\ F_m\ \textit{blinding-of}$ *and*
- $\textit{merge}_F :: ({'a_m}, {'a_h})\ \textit{hash} \Rightarrow {'a_m}\ \textit{merge} \Rightarrow ({'a_m}, {'a_h})\ F_m\ \textit{merge}$

with the following properties

$$\textit{Monotonicity}\qquad \frac{bo \le bo'}{\textit{blinding-of}_F\ h\ bo \le \textit{blinding-of}_F\ h\ bo'}$$

$$\textit{Congruence}\qquad \frac{\forall\, a{\in}A.\ \forall\, b.\ m\ a\ b = m'\ a\ b}{\forall\, x \in \{y.\ \textit{set}_1\text{-}F_m\ y \subseteq A\}.\ \forall\, b.\ \textit{merge}_F\ h\ m\ x\ y = \textit{merge}_F\ h\ m'\ x\ y}$$

$$\textit{Hashes}\qquad \frac{\textit{blinding-respects-hashes}\ h\ bo}{\textit{blinding-respects-hashes}\ (\textit{hash}_F\ h)\ (\textit{blinding-of}_F\ h\ bo)}$$

$$\textit{Blinding order}\qquad \frac{\textit{blinding-of-on}\ A\ h\ bo}{\textit{blinding-of-on}\ \{x.\ \textit{set}_1\text{-}F_m\ x \subseteq A\}\ (\textit{hash}_F\ h)\ (\textit{blinding-of}_F\ h\ bo)}$$

$$\textit{Merge}\qquad \frac{\textit{merge-on}\ A\ h\ bo\ m}{\textit{merge-on}\ \{x.\ \textit{set}_1\text{-}F_m\ x \subseteq A\}\ (\textit{hash}_F\ h)\ (\textit{blinding-of}_F\ h\ bo)\ (\textit{merge}_F\ h\ m)}$$

where $\textit{hash}_F\ h = \textit{hash}'_F \circ \textit{map-}F_m\ h\ id$ for the BNF mapper $\textit{map-}F_m$, and where the BNF setter $\textit{set}_1\text{-}F_m\ x$ returns all atoms of type ${'a_m}$ in $x :: ({'a_m}, {'a_h})\ F_m$.

Every Merkle functor preserves the Merkle interface specification: set $A = UNIV$ in the merge property and use the equivalence between the Merkle interface and *merge-on*. With this, we now state the main theoretical contribution of this paper.

▶ **Theorem 5.** *Merkle functors of arbitrary arity are closed under composition and least fixpoints.*

Proof. (Sketch) 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** $T = T \langle T\ F \rangle$, 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 *hash-T′* is defined by primitive recursion:

$$\textit{hash-}T'\ (T_m\ x) = T_h\ (\textit{hash-}F'\ (\textit{map-}F_m\ \textit{hash-}T'\ x)).$$

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

$$\frac{\textit{blinding-of-}F\ \textit{hash-}T\ \textit{blinding-of-}T\ x\ y}{\textit{blinding-of-}T\ (T_m\ x)\ (T_m\ y)}$$

  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:

$$\textit{merge-}T\ (T_m\ x)\ (T_m\ y) = \textit{map-option}\ T_m\ (\textit{merge-}F\ \textit{hash-}T\ \textit{merge-}T\ x\ y)$$

  Congruence ensures that *merge-F* calls *merge-T* recursively only on smaller arguments.

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. ◀

Isabelle/HOL lacks the abstraction over type constructors necessary to formalize this theorem. As our approach also translates to theorem provers with more expressive type systems (e.g., Lean, Coq), the theorem could be formalized there. For Isabelle/HOL, we adopt an approach similar to Blanchette et al. [3]. We axiomatize a binary Merkle functor and carry out the construction and proofs for least fixpoints and composition, illustrating how the definition and proofs generalize to functors with several type arguments. The example ADS constructions in Section 3.6 then 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 $\textit{blindable}_h$ and $\textit{blindable}_m$ are a unary Merkle functor.
- Sums and products are binary Merkle functors. We set $\times_h = \times_m = \times$ and $+_h = +_m = +$. 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 operation *merge-of-prod* attempts to merge each component separately, while *merge-of-sum* can only merge left and left, or right and right values. (Formally, $\times_m$ and $+_m$ should take four type arguments. However, as sums and products do not themselves contain blindable positions, the type arguments $'a_h$ and $'b_h$ are ignored in inclusion proofs and we therefore omit them.)

- The function space $'a \Rightarrow 'b$ is a unary Merkle functor in the codomain. Like for sums and products, $\Rightarrow_h = \Rightarrow_m = \Rightarrow$ and no additional type arguments are added. Hashing is function composition and the blinding order is pointwise.

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

Theorem 5 shows that all datatypes built from the Merkle functors in the previous section are Merkle functors. We apply the construction sketched in the proof to concrete datatypes that build on top of each other. For example, lists, rose trees [24], and Canton transactions are all Merkle functors. We prove that *'a list* is a Merkle functor with the help of an isomorphic data type that is the least fixpoint $\mu X.\ 1 + {'a} \times X$ and following the fixpoint construction of Theorem 5. We transfer the definitions and theorems to *list* using the transfer package [16]. Rose trees are then given by the datatype

datatype $'a$ *rose-tree* $=$ *Tree* ‹($'a \times 'a$ *rose-tree list*) *blindable*›

Applying the construction gives us Merkle rose trees with the corresponding operations and their properties.

datatype $'a_h$ *rose-tree*$_h$ $=$ *Tree*$_h$ ‹($'a_h \times_h 'a_h$ *rose-tree*$_h$ *list*$_h$) *blindable*$_h$›
datatype $('a_m,\ 'a_h)$ *rose-tree*$_m$ $=$ *Tree*$_m$
‹($'a_m \times_m ('a_m,\ 'a_h)$ *rose-tree*$_m$ *list*$_m$, $'a_h \times_h 'a_h$ *rose-tree*$_h$ *list*$_h$) *blindable*$_m$›

From here, it is only a small step to transactions in Canton. Views are isomorphic to Merkle rose trees where the data at the nodes is instantiated, i.e., composed, with the Merkle functor corresponding to *view-metadata blindable* $\times$ *view-data blindable*. Then, transactions compose the Merkle functor for *common-metadata blindable* $\times$ *participant-metadata blindable* $\times$ *- list* with views. We have lifted our machinery from these raw Merkle functors to the datatypes *view*$_m$ and *transaction*$_m$ using the lifting and transfer packages [16].

4 Creating Inclusion Proofs

So far, given a tree-like data type $'t$, we showed how to systematically construct the corresponding type of ADSs $'t_h$ and their inclusion proofs $'t_m$. To make use of this construction in practice, we must also be able to create values of type $'t_m$ 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 sketch 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:

- Digesting, *hash-source-tree*, returns the root hash for a rose 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 (the root is blinded).

Digesting and fully blinding conceptually do the same thing, but their return types ($'a_h$ *rose-tree*$_h$ and $('a_m,\ 'a_h)$ *rose-tree*$_m$) differ. As rose trees are parameterized by their node label type, digesting, embedding, and fully blinding take parameters which digest, embed, or fully blind the node labels. The expected properties hold: the embedded and fully blinded versions of the same rose tree have the same hash, namely the digest of the rose 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 [23], this corresponds to a zipper [15] focused on the subtree. This connection 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* = ⟨*'a* × *'a rose-tree list* × *'a rose-tree list*⟩
type_synonym *'a path* = ⟨*'a path-elem list*⟩
type_synonym *'a zipper* = ⟨*'a path* × *'a rose-tree*⟩

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 Merkle rose trees, i.e., inclusion proofs for rose trees, have the exact same shape, except that all the type constructors are subscripted by $_m$ and have another type parameter capturing the type of hashes (e.g., $('a_m,\ 'a_h)\ \textit{zipper}_m$). Like for rose trees, we define operations that blind and embed a path respectively. This way, zippers on rose trees can be turned into zippers on Merkle rose trees. As expected, starting with a rose tree zipper, blinding and embedding its path yields a Merkle rose tree with the same hash. Furthermore, reconstructing a Merkle rose tree from an embedded rose tree zipper gives the same result as first reconstructing the rose tree 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*:

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

Inclusion proofs derived from zippers prove inclusion of a single subtree of the rose tree. The general case of several subtrees can be reduced to the single-subtree case using merging. 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 operations to turn a rose tree into a zipper focused on the root and into zippers into its subtrees. Then, the function *zippers-rose-tree* enumerates the inclusion proof zippers for all nodes of a rose tree using those two operations. This allows us to easily model the messages that the initiator of a transaction sends in the first phase of Canton's commit protocol. The initiator constructs all zippers for the views in the transaction tree, and then turns each such zipper into an inclusion proof. Finally, the initiator merges each view proof with the proof from the zipper for the transaction metadata and ships it to the recipients.

At the end of the two-phase commit protocol, the domain's commit message contains an inclusion proof of the view metadata for all the views that the participant should have received. The participant can decide whether it has received all views it was supposed to receive, it compares this inclusion proof against the merged inclusion proofs that it had received from the initiator, using the inclusion proof order *blinding-of-transaction* on transactions.

5 Related Work

Miller et al. developed a lambda calculus with authentication primitives for generic tree structures [21]. The calculus was formalized in Isabelle/HOL by Brun and Traytel [5]. 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 [5] 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 generic correctness proofs for functions on the data structures.

Several other works formalize (binary) Merkle trees. White [25] formalized sparse Merkle trees [9] as part of a Coq model of a cryptographic ledger. An asset belongs to an address if the address encodes a path in the sparse Merkle tree from the root node to a leaf with the asset. A merge operation allows a single Merkle tree to provide several inclusion proofs. Our generic development can be instantiated to cover this structure. Yu et al. [26] 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 [22] formalize binary Merkle trees as used in a timestamping protocol. They automatically verify parts of the protocol using the Mona theorem prover.

As part of the Everest project, HACL* contains a formal verification of balanced binary Merkle trees [13]. The balanced trees represent a sequence of hashes, which is padded with dummy values to a power of 2. A reduction proof shows that hash collisions between root hashes can be traced back to hash collisions of the underlying hash function. The main focus is on a refinement to an efficient executable implementation. It would be interesting to investigate whether and how their reduction-proof approach to dealing with hash collisions can be generalized compositionally to our general ADS setting.

Seefried [23] observed that inclusion proofs in a Merkle tree correspond to Huet-style zippers [15], where the subtrees in zipper context have been replaced by the Merkle root hashes. McBride showed that zippers represent one-hole contexts [19]. In this analogy, our inclusion multi-proofs correspond to contexts with arbitrarily many holes. These many-hole zippers must not be confused with Kiselyov's zippers [17] and Hinze and Jeuring's webs [14], 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 tree-shaped HOL datatypes (i.e., functors), 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 real-world structures used in Canton, a Byzantine fault tolerant commit protocol.

The ongoing formalization of the Canton protocol will continue to test our abstractions and trigger further improvements. As noted earlier, ADSs not 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* in HOL to invert hash functions. In fact, our current model does not even distinguish between the authenticated data structure and its digest because of this. A sound confidentiality analysis must therefore restrict the adversary using an appropriate calculus, e.g., a Dolev-Yao style deduction relation [11]. The analysis must take into account situations such as a Merkle tree node with two children with identical hashes; unblinding one child automatically unblinds the other. However, our representation distinguishes between the two, which might represent a problem. Another situation where this might be a problem is when merging inclusion proofs for commutative structures. One option is to 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 [12].

Moreover, our representation of hashes as terms makes hashing injective. While this is "morally equivalent" to standard cryptographic assumptions, an alternative (followed by [5]) 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 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. 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.

We have based our construction on bounded natural functors (BNFs) as they are the semantic domain for datatypes in Isabelle/HOL and closed under least fixpoints. Fortunately, our Merkle constructions and proof need very little of the BNF structure and therefore generalize straightforwardly to other systems. For example, Lean's quotients of polynomial functors (QFPs) [1] are more general than BNFs and also closed under fixpoints. The concept of a Merkle functor can be directly expressed on QPFs as the BNF setter in Def. 4 can be replaced by the predicate lifting for QPFs. The closure proofs for composition and least fixpoint also work with predicate lifting. Moreover, the meta-theory can be formalized in Lean's more expressive type system, even for functors of arbitrary arity, and then instantiated for the concrete functor at hand. So in Lean, we would not have to redo the proof for every ADS. This also applies to other systems like Agda and Coq. Furthermore, the construction of concrete functors can be mimicked in any system that supports mutually recursive algebraic datatypes and higher-order functions, as all our ADS are built from sums, products, function spaces, and nested recursion through other datatypes, e.g., $\mathit{blindable}_h$ and $\mathit{blindable}_m$. (Nested datatype recursion can be reduced to mutual recursion [2], so mutually recursive algebraic datatypes suffice.)

References

1. Jeremy Avigad, Mario Carneiro, and Simon Hudon. Data types as quotients of polynomial functors. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019*, volume 141 of *LIPIcs*, pages 6:1–6:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.6.
2. Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL – lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry,

editors, *TPHOLs 1999*, volume 1690 of *LNCS*, pages 19–36. Springer, 1999. doi:10.1007/3-540-48256-3_3.

3 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In *Interactive Theorem Proving (ITP 2014)*, pages 93–110, 2014.

4 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.

5 Matthias Brun and Dmitriy Traytel. Generic authenticated data structures, formally. In *Interactive Theorem Proving (ITP 2019)*, pages 10:1—10:18, 2019.

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

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

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

9 Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs. Cryptology ePrint Archive, Report 2016/683, 2016. https://eprint.iacr.org/2016/683.

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

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

12 Basil Fürer, Andreas Lochbihler, Joshua Schneider, and Dmitriy Traytel. Quotients of bounded natural functors. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning (IJCAR 2020)*, volume 12167 of *LNAI*, pages 58–78. Springer, 2020. doi:10.1007/978-3-030-51054-1_4.

13 Hacl* verified merkle tree library. https://github.com/project-everest/hacl-star/tree/master/secure_api/merkle_tree, 2020.

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

15 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549—554, 1997.

16 Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs (CPP 2013)*, pages 131—-146, 2013.

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

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

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

20 Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology (CRYPTO 1987)*, pages 369–378, 1987.

21 Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Principles of Programming Languages (POPL 2014)*, pages 411—423, 2014.

22 Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. Proving properties of incremental Merkle trees. In *Automated Deduction (CADE 2005)*, pages 424–440, 2005.

23 Sean Seefried. Merkle proofs for free! Functional Programming Sydney, http://code.ouroborus.net/fp-syd/past/2017/2017-04-Seefried-Merkle.pdf, 2017.

24 D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996. doi:10.1006/jpdc.1996.0160.

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

26 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.