

# Authenticated Data Structures as Functors

Andreas Lochbihler     Ognjen Maric

Digital Asset

February 6, 2026

## 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.

## Contents

<b>1</b>	<b>Authenticated Data Structures</b>	<b>2</b>
1.1	Interface . . . . .	2
1.1.1	Types . . . . .	2
1.1.2	Properties . . . . .	2
1.2	Auxiliary definitions . . . . .	3
1.2.1	Blinding . . . . .	3
1.2.2	Merging . . . . .	4
1.3	Interface equality . . . . .	5
1.4	Parametricity rules . . . . .	5
<b>2</b>	<b>Building blocks for authenticated data structures on datatypes</b>	<b>6</b>
2.1	Building Block: Identity Functor . . . . .	6
2.1.1	Example: instantiation for <i>unit</i> . . . . .	7

2.2	Building Block: Blindable Position . . . . .	7
2.2.1	Hashes . . . . .	8
2.2.2	Blinding . . . . .	8
2.2.3	Merging . . . . .	9
2.2.4	Merkle interface . . . . .	10
2.2.5	Non-recursive blindable positions . . . . .	10
2.3	Building block: Sums . . . . .	11
2.3.1	Hashes . . . . .	11
2.3.2	Blinding . . . . .	11
2.3.3	Merging . . . . .	12
2.3.4	Merkle interface . . . . .	12
2.4	Building Block: Products . . . . .	13
2.4.1	Hashes . . . . .	13
2.4.2	Blinding . . . . .	13
2.4.3	Merging . . . . .	13
2.4.4	Merkle Interface . . . . .	14
2.5	Building Block: Lists . . . . .	14
2.5.1	The Isomorphism . . . . .	15
2.5.2	Hashes . . . . .	16
2.5.3	Blinding . . . . .	17
2.5.4	Merging . . . . .	17
2.5.5	Transferring the Constructions to Lists . . . . .	18
2.6	Building block: function space . . . . .	19
2.6.1	Hashes . . . . .	19
2.6.2	Blinding . . . . .	19
2.6.3	Merging . . . . .	20
2.6.4	Merkle Interface . . . . .	21
2.7	Rose trees . . . . .	21
2.7.1	Hashes . . . . .	21
2.7.2	Blinding . . . . .	22
2.7.3	Merging . . . . .	24
2.7.4	Merkle interface . . . . .	25
<b>3</b>	<b>Generic construction of authenticated data structures</b>	<b>25</b>
3.1	Functors . . . . .	25
3.1.1	Source functor . . . . .	25
3.1.2	Base Merkle functor . . . . .	26
3.1.3	Least fixpoint . . . . .	26
3.1.4	Composition . . . . .	26
3.2	Root hash . . . . .	27
3.2.1	Base functor . . . . .	27
3.2.2	Least fixpoint . . . . .	27
3.2.3	Composition . . . . .	27
3.3	Blinding relation . . . . .	28

3.3.1	Blinding on the base functor ( $F_m$ ) . . . . .	28
3.3.2	Blinding on least fixpoints . . . . .	28
3.3.3	Blinding on composition . . . . .	29
3.4	Merging . . . . .	30
3.4.1	Merging on the base functor . . . . .	30
3.4.2	Merging on the least fixpoint . . . . .	30
3.4.3	Merging and composition . . . . .	31
3.5	Inclusion proof construction for rose trees . . . . .	32
3.5.1	Hashing, embedding and blinding source trees . . . . .	32
3.5.2	Auxiliary definitions: selectors and list splits . . . . .	33
3.5.3	Zippers . . . . .	33
3.6	All zippers of a rose tree . . . . .	36
<b>4</b>	<b>Canton's hierarchical transaction trees</b>	<b>38</b>
4.1	Views as authenticated data structures . . . . .	38
4.2	Transaction trees as authenticated data structures . . . . .	41
4.3	Constructing authenticated data structures for views . . . . .	43
4.3.1	Inclusion proof for the mediator . . . . .	46
4.3.2	Inclusion proofs for participants . . . . .	46

**theory** *Merkle-Interface*

**imports**

*Main*

*HOL-Library.Conditional-Parametricity*

*HOL-Library.Monad-Syntax*

**begin**

**alias** *vimage2p* = *BNF-Def.vimage2p*

**alias** *Grp* = *BNF-Def.Grp*

**alias** *setl* = *Basic-BNFs.setl*

**alias** *setr* = *Basic-BNFs.setr*

**alias** *fsts* = *Basic-BNFs.fsts*

**alias** *snds* = *Basic-BNFs.snds*

$\langle ML \rangle$

**lemma** *vimage2p-mono'*:  $R \leq S \implies vimage2p\ f\ g\ R \leq vimage2p\ f\ g\ S$

$\langle proof \rangle$

**lemma** *vimage2p-map-rel-prod*:

$vimage2p\ (map\ prod\ f\ g)\ (map\ prod\ f'\ g')\ (rel\ prod\ A\ B) = rel\ prod\ (vimage2p\ f\ f'\ A)\ (vimage2p\ g\ g'\ B)$

$\langle proof \rangle$

**lemma** *vimage2p-map-list-all2*:

$vimage2p\ (map\ f)\ (map\ g)\ (list\ all2\ A) = list\ all2\ (vimage2p\ f\ g\ A)$

*<proof>*

**lemma** *equivclp-least*:

**assumes** *le*:  $r \leq s$  **and** *s*: *equivp s*

**shows** *equivclp*  $r \leq s$

*<proof>*

**lemma** *reflp-eq-onp*: *reflp*  $R \longleftrightarrow eq\text{-onp } (\lambda x. True) \leq R$

*<proof>*

**lemma** *eq-onpE*:

**assumes** *eq-onp*  $P x y$

**obtains**  $x = y P y$

*<proof>*

**lemma** *case-unit-parametric* [*transfer-rule*]: *rel-fun*  $A (rel\text{-fun } (=) A)$  *case-unit*

*case-unit*

*<proof>*

# 1 Authenticated Data Structures

## 1.1 Interface

### 1.1.1 Types

**type-synonym**  $('a_m, 'a_h)$  *hash* =  $'a_m \Rightarrow 'a_h$  — Type of hash operation

**type-synonym**  $'a_m$  *blinding-of* =  $'a_m \Rightarrow 'a_m \Rightarrow bool$

**type-synonym**  $'a_m$  *merge* =  $'a_m \Rightarrow 'a_m \Rightarrow 'a_m$  *option* — merging that can fail for values with different hashes

### 1.1.2 Properties

**locale** *merkle-interface* =

**fixes**  $h :: ('a_m, 'a_h)$  *hash*

**and**  $bo :: 'a_m$  *blinding-of*

**and**  $m :: 'a_m$  *merge*

**assumes** *merge-respects-hashes*:  $h a = h b \longleftrightarrow (\exists ab. m a b = Some ab)$

**and** *idem*:  $m a a = Some a$

**and** *commute*:  $m a b = m b a$

**and** *assoc*:  $m a b \gg m c = m b c \gg m a$

**and** *bo-def*:  $bo a b \longleftrightarrow m a b = Some b$

**begin**

**lemma** *reflp*: *reflp* *bo*

*<proof>*

**lemma** *antisymp*: *antisymp* *bo*

*<proof>*

**lemma** *transp*: *transp* *bo*

*<proof>*

**lemma** *hash*:  $bo \leq vimage2p\ h\ h\ (=)$   
*<proof>*

**lemma** *join*:  $m\ a\ b = Some\ ab \iff bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall u. bo\ a\ u \implies bo\ b\ u \implies bo\ ab\ u)$   
*<proof>*

The equivalence closure of the blinding relation are the equivalence classes of the hash function (the kernel).

**lemma** *equivclp-blinding-of*:  $equivclp\ bo = vimage2p\ h\ h\ (=)$  (**is** *?lhs = ?rhs*)  
*<proof>*

**end**

## 1.2 Auxiliary definitions

Directly proving that an interface satisfies the specification of a Merkle interface as given above is difficult. Instead, we provide several layers of auxiliary definitions that can easily be proved layer-by-layer.

In particular, proving that an interface on recursive datatypes is a Merkle interface requires induction. As the induction hypothesis only applies to a subset of values of a type, we add auxiliary definitions equipped with an explicit set  $A$  of values to which the definition applies. Once the induction proof is complete, we can typically instantiate  $A$  with  $UNIV$ . In particular, in the induction proof for a layer, we can assume that properties for the earlier layers hold for *all* values, not just those in the induction hypothesis.

### 1.2.1 Blinding

**locale** *blinding-respects-hashes* =  
  **fixes**  $h :: ('a_m, 'a_h)\ hash$   
  **and**  $bo :: 'a_m\ blinding-of$   
  **assumes**  $hash: bo \leq vimage2p\ h\ h\ (=)$   
**begin**

**lemma** *blinding-hash-eq*:  $bo\ x\ y \implies h\ x = h\ y$   
*<proof>*

**end**

**locale** *blinding-of-on* =  
  *blinding-respects-hashes*  $h\ bo$   
  **for**  $A :: 'a_m\ set$   
  **and**  $h :: ('a_m, 'a_h)\ hash$   
  **and**  $bo :: 'a_m\ blinding-of$

+ **assumes** *refl*:  $x \in A \implies bo\ x\ x$   
**and** *trans*:  $\llbracket bo\ x\ y; bo\ y\ z; x \in A \rrbracket \implies bo\ x\ z$   
**and** *antisym*:  $\llbracket bo\ x\ y; bo\ y\ x; x \in A \rrbracket \implies x = y$   
**begin**

**lemma** *refl-pointfree*:  $eq\text{-}onp\ (\lambda x. x \in A) \leq bo$   
 $\langle proof \rangle$

**lemma** *blinding-respects-hashes*: *blinding-respects-hashes* *h bo*  $\langle proof \rangle$   
**lemmas** *hash = hash*

**lemma** *trans-pointfree*:  $eq\text{-}onp\ (\lambda x. x \in A)\ OO\ bo\ OO\ bo \leq bo$   
 $\langle proof \rangle$

**lemma** *antisym-pointfree*:  $inf\ (eq\text{-}onp\ (\lambda x. x \in A)\ OO\ bo)\ bo^{-1-1} \leq (=)$   
 $\langle proof \rangle$

**end**

### 1.2.2 Merging

In general, we prove the properties of blinding before the properties of merging. Thus, in the following definitions we assume that the blinding properties already hold on *UNIV*. The *Ball* restricts the argument of the merge operation on which induction will be done.

**locale** *merge-on* =  
*blinding-of-on UNIV h bo*  
**for** *A* ::  $'a_m$  *set*  
**and** *h* ::  $('a_m, 'a_n)$  *hash*  
**and** *bo* ::  $'a_m$  *blinding-of*  
**and** *m* ::  $'a_m$  *merge* +  
**assumes** *join*:  $\llbracket h\ a = h\ b; a \in A \rrbracket$   
 $\implies \exists ab. m\ a\ b = Some\ ab \wedge bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall u. bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow$   
*bo ab u)*  
**and** *undefined*:  $\llbracket h\ a \neq h\ b; a \in A \rrbracket \implies m\ a\ b = None$   
**begin**

**lemma** *same*:  $a \in A \implies m\ a\ a = Some\ a$   
 $\langle proof \rangle$

**lemma** *blinding-of-antisym-on*: *blinding-of-on UNIV h bo*  $\langle proof \rangle$

**lemma** *transp*: *transp bo*  
 $\langle proof \rangle$

**lemmas** *hash = hash*  
**and** *refl = refl*  
**and** *antisym = antisym[OF - - UNIV-I]*

**lemma** *respects-hashes*:

$a \in A \implies h a = h b \iff (\exists ab. m a b = \text{Some } ab)$   
*<proof>*

**lemma** *join'*:

$a \in A \implies \forall ab. m a b = \text{Some } ab \iff bo a ab \wedge bo b ab \wedge (\forall u. bo a u \longrightarrow bo b u \longrightarrow bo ab u)$   
*<proof>*

**lemma** *merge-on-subset*:

$B \subseteq A \implies \text{merge-on } B h bo m$   
*<proof>*

**end**

### 1.3 Interface equality

Here, we prove that the auxiliary definitions specify the same interface as the original ones.

**lemma** *merkle-interface-aux*: *merkle-interface*  $h bo m = \text{merge-on UNIV } h bo m$

(**is** *?lhs = ?rhs*)  
*<proof>*

**lemma** *merkle-interfaceI* [*locale-witness*]:

**assumes** *merge-on UNIV h bo m*  
**shows** *merkle-interface h bo m*  
*<proof>*

**lemma** (**in** *merkle-interface*) *merkle-interfaceD*: *merge-on UNIV h bo m*

*<proof>*

### 1.4 Parametricity rules

**context includes** *lifting-syntax* **begin**

**parametric-constant** *le-fun-parametric*[*transfer-rule*]: *le-fun-def*

**parametric-constant** *vimage2p-parametric*[*transfer-rule*]: *vimage2p-def*

**parametric-constant** *blinding-respects-hashes-parametric-aux*: *blinding-respects-hashes-def*

**lemma** *blinding-respects-hashes-parametric* [*transfer-rule*]:

$((A1 \implies A2) \implies (A1 \implies A1 \implies (\longleftrightarrow))) \implies (\longleftrightarrow)$   
*blinding-respects-hashes blinding-respects-hashes*  
**if** [*transfer-rule*]: *bi-unique A2 bi-total A1*  
*<proof>*

**parametric-constant** *blinding-of-on-axioms-parametric* [*transfer-rule*]:

*blinding-of-on-axioms-def*[*folded Ball-def, unfolded le-fun-def le-bool-def eq-onp-def relcompp.simps, simplified*]

**parametric-constant** *blinding-of-on-parametric* [*transfer-rule*]: *blinding-of-on-def*

```

parametric-constant antisymp-parametric[transfer-rule]: antisymp-def
parametric-constant transp-parametric[transfer-rule]: transp-def

parametric-constant merge-on-axioms-parametric [transfer-rule]: merge-on-axioms-def
parametric-constant merge-on-parametric[transfer-rule]: merge-on-def

parametric-constant merkle-interface-parametric[transfer-rule]: merkle-interface-def
end

end

theory ADS-Construction imports
  Merkle-Interface
  HOL-Library.Simps-Case-Conv
begin

```

## 2 Building blocks for authenticated data structures on datatypes

### 2.1 Building Block: Identity Functor

If nothing is blidable in a type, then the type itself is the hash and the ADS of itself.

**abbreviation** (*input*) *hash-discrete* :: ('a, 'a) *hash* **where** *hash-discrete*  $\equiv$  *id*

**abbreviation** (*input*) *blinding-of-discrete* :: 'a *blinding-of* **where**  
*blinding-of-discrete*  $\equiv$  (=)

**definition** *merge-discrete* :: 'a *merge* **where**  
*merge-discrete*  $x\ y = (if\ x = y\ then\ Some\ y\ else\ None)$

**lemma** *blinding-of-discrete-hash*:  
*blinding-of-discrete*  $\leq$  *vimage2p hash-discrete hash-discrete* (=)  
 <*proof*>

**lemma** *blinding-of-on-discrete* [*locale-witness*]:  
*blinding-of-on UNIV hash-discrete blinding-of-discrete*  
 <*proof*>

**lemma** *merge-on-discrete* [*locale-witness*]:  
*merge-on UNIV hash-discrete blinding-of-discrete merge-discrete*  
 <*proof*>

**lemma** *merkle-discrete* [*locale-witness*]:  
*merkle-interface hash-discrete blinding-of-discrete merge-discrete*  
 <*proof*>

**parametric-constant** *merge-discrete-parametric* [transfer-rule]: *merge-discrete-def*

### 2.1.1 Example: instantiation for *unit*

**abbreviation** (*input*) *hash-unit* :: (*unit*, *unit*) *hash* **where** *hash-unit*  $\equiv$  *hash-discrete*

**abbreviation** *blinding-of-unit* :: *unit* *blinding-of* **where**  
*blinding-of-unit*  $\equiv$  *blinding-of-discrete*

**abbreviation** *merge-unit* :: *unit* *merge* **where** *merge-unit*  $\equiv$  *merge-discrete*

**lemma** *blinding-of-unit-hash*:  
*blinding-of-unit*  $\leq$  *vimage2p* *hash-unit* *hash-unit* (=)  
(*proof*)

**lemma** *blinding-of-on-unit*:  
*blinding-of-on UNIV* *hash-unit* *blinding-of-unit*  
(*proof*)

**lemma** *merge-on-unit*:  
*merge-on UNIV* *hash-unit* *blinding-of-unit* *merge-unit*  
(*proof*)

**lemma** *merkle-interface-unit*:  
*merkle-interface* *hash-unit* *blinding-of-unit* *merge-unit*  
(*proof*)

## 2.2 Building Block: Blindable Position

**type-synonym** *'a* *blindable* = *'a*

The following type represents the hashes of a datatype. We model hashes as being injective, but not surjective; some hashes do not correspond to any values of the original datatypes. We model such values as "garbage" coming from a countable set (here, naturals).

**type-synonym** *garbage* = *nat*

**datatype** *'a<sub>h</sub>* *blindable<sub>h</sub>* = *Content* *'a<sub>h</sub>* | *Garbage* *garbage*

**datatype** (*'a<sub>m</sub>*, *'a<sub>h</sub>*) *blindable<sub>m</sub>* = *Unblinded* *'a<sub>m</sub>* | *Blinded* *'a<sub>h</sub>* *blindable<sub>h</sub>*

### 2.2.1 Hashes

**primrec** *hash-blindable'* :: ((*'a<sub>h</sub>*, *'a<sub>h</sub>*) *blindable<sub>m</sub>*, *'a<sub>h</sub>* *blindable<sub>h</sub>*) *hash* **where**  
*hash-blindable'* (*Unblinded* *x*) = *Content* *x*  
| *hash-blindable'* (*Blinded* *x*) = *x*

**definition** *hash-blindable* :: (*'a<sub>m</sub>*, *'a<sub>h</sub>*) *hash*  $\Rightarrow$  ((*'a<sub>m</sub>*, *'a<sub>h</sub>*) *blindable<sub>m</sub>*, *'a<sub>h</sub>* *blindable<sub>h</sub>*) *hash* **where**

*hash-blindable*  $h = \text{hash-blindable}' \circ \text{map-blindable}_m h \text{ id}$

**lemma** *hash-blindable-simps* [simp]:

*hash-blindable*  $h$  (*Unblinded*  $x$ ) = *Content* ( $h$   $x$ )

*hash-blindable*  $h$  (*Blinded*  $y$ ) =  $y$

*<proof>*

**lemma** *hash-map-blindable-simp*:

*hash-blindable*  $f$  (*map-blindable* <sub>$m$</sub>   $f'$  *id*  $x$ ) = *hash-blindable* ( $f \circ f'$ )  $x$

*<proof>*

**parametric-constant** *hash-blindable'-parametric* [transfer-rule]: *hash-blindable'-def*

**parametric-constant** *hash-blindable-parametric* [transfer-rule]: *hash-blindable-def*

## 2.2.2 Blinding

**context**

**fixes**  $h :: ('a_m, 'a_h)$  *hash*

**and**  $bo :: 'a_m$  *blinding-of*

**begin**

**inductive** *blinding-of-blindable* :: ( $'a_m, 'a_h$ ) *blindable* <sub>$m$</sub>  *blinding-of* **where**

*blinding-of-blindable* (*Unblinded*  $x$ ) (*Unblinded*  $y$ ) **if**  $bo$   $x$   $y$

| *blinding-of-blindable* (*Blinded*  $x$ )  $t$  **if** *hash-blindable*  $h$   $t = x$

**inductive-simps** *blinding-of-blindable-simps* [simp]:

*blinding-of-blindable* (*Unblinded*  $x$ )  $y$

*blinding-of-blindable* (*Blinded*  $x$ )  $y$

*blinding-of-blindable*  $z$  (*Unblinded*  $x$ )

*blinding-of-blindable*  $z$  (*Blinded*  $x$ )

**inductive-simps** *blinding-of-blindable-simps2*:

*blinding-of-blindable* (*Unblinded*  $x$ ) (*Unblinded*  $y$ )

*blinding-of-blindable* (*Unblinded*  $x$ ) (*Blinded*  $y'$ )

*blinding-of-blindable* (*Blinded*  $x'$ ) (*Unblinded*  $y$ )

*blinding-of-blindable* (*Blinded*  $x'$ ) (*Blinded*  $y'$ )

**end**

**lemma** *blinding-of-blindable-mono*:

**assumes**  $bo \leq bo'$

**shows** *blinding-of-blindable*  $h$   $bo \leq \text{blinding-of-blindable } h \text{ } bo'$

*<proof>*

**lemma** *blinding-of-blindable-hash*:

**assumes**  $bo \leq \text{vimage2p } h \text{ } h$  (=)

**shows** *blinding-of-blindable*  $h$   $bo \leq \text{vimage2p } (\text{hash-blindable } h) (\text{hash-blindable } h)$  (=)

*<proof>*

**lemma** *blinding-of-on-blindable* [locale-witness]:

**assumes** *blinding-of-on A h bo*

**shows** *blinding-of-on {x. set1-blindable<sub>m</sub> x ⊆ A} (hash-blindable h) (blinding-of-blindable h bo)*

**(is** *blinding-of-on ?A ?h ?bo*)

*<proof>*

**lemmas** *blinding-of-blindable* [locale-witness] = *blinding-of-on-blindable*[of UNIV, simplified]

**case-of-simps** *blinding-of-blindable-alt-def: blinding-of-blindable-simps2*

**parametric-constant** *blinding-of-blindable-parametric* [transfer-rule]: *blinding-of-blindable-alt-def*

### 2.2.3 Merging

**context**

**fixes** *h :: ('a<sub>m</sub>, 'a<sub>h</sub>) hash*

**fixes** *m :: 'a<sub>m</sub> merge*

**begin**

**fun** *merge-blindable* :: ('a<sub>m</sub>, 'a<sub>h</sub>) *blindable<sub>m</sub> merge* **where**

*merge-blindable (Unblinded x) (Unblinded y) = map-option Unblinded (m x y)*

| *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)*

| *merge-blindable (Blinded t) (Blinded u) = (if t = u then Some (Blinded u) else None)*

**lemma** *merge-on-blindable* [locale-witness]:

**assumes** *merge-on A h bo m*

**shows** *merge-on {x. set1-blindable<sub>m</sub> x ⊆ A} (hash-blindable h) (blinding-of-blindable h bo) merge-blindable*

**(is** *merge-on ?A ?h ?bo ?m*)

*<proof>*

**lemmas** *merge-blindable* [locale-witness] =

*merge-on-blindable*[of UNIV, simplified]

**end**

**lemma** *merge-blindable-alt-def*:

*merge-blindable h m x y = (case (x, y) of*

*(Unblinded x, Unblinded y) ⇒ map-option Unblinded (m x y)*

| *(Blinded x, Unblinded y) ⇒ (if Content (h y) = x then Some (Unblinded y) else None)*

| *(Unblinded y, Blinded x) ⇒ (if Content (h y) = x then Some (Unblinded y) else*

*None*)  
 | (*Blinded t, Blinded u*)  $\Rightarrow$  (*if t = u then Some (Blinded u) else None*)  
 ⟨*proof*⟩

**parametric-constant** *merge-blindable-parametric* [*transfer-rule*]: *merge-blindable-alt-def*

**lemma** *merge-blindable-cong* [*fundef-cong*]:  
 assumes  $\bigwedge a b. \llbracket a \in \text{set1-blindable}_m x; b \in \text{set1-blindable}_m y \rrbracket \Longrightarrow m a b = m'$   
*a b*  
 shows *merge-blindable h m x y = merge-blindable h m' x y*  
 ⟨*proof*⟩

## 2.2.4 Merkle interface

**lemma** *merkle-blindable* [*locale-witness*]:  
 assumes *merkle-interface h bo m*  
 shows *merkle-interface (hash-blindable h) (blinding-of-blindable h bo) (merge-blindable h m)*  
 ⟨*proof*⟩

## 2.2.5 Non-recursive blindable positions

For a non-recursive data type *'a*, the type of hashes in *blindable<sub>m</sub>* is fixed to be simply *'a blindable<sub>h</sub>*. We obtain this by instantiating the type variable with the identity building block.

**type-synonym** *'a nr-blindable* = (*'a, 'a*) *blindable<sub>m</sub>*

**abbreviation** *hash-nr-blindable* :: (*'a nr-blindable, 'a blindable<sub>h</sub>*) *hash* **where**  
*hash-nr-blindable*  $\equiv$  *hash-blindable hash-discrete*

**abbreviation** *blinding-of-nr-blindable* :: *'a nr-blindable blinding-of* **where**  
*blinding-of-nr-blindable*  $\equiv$  *blinding-of-blindable hash-discrete blinding-of-discrete*

**abbreviation** *merge-nr-blindable* :: *'a nr-blindable merge* **where**  
*merge-nr-blindable*  $\equiv$  *merge-blindable hash-discrete merge-discrete*

**lemma** *merge-on-nr-blindable*:  
*merge-on UNIV hash-nr-blindable blinding-of-nr-blindable merge-nr-blindable*  
 ⟨*proof*⟩

**lemma** *merkle-nr-blindable*:  
*merkle-interface hash-nr-blindable blinding-of-nr-blindable merge-nr-blindable*  
 ⟨*proof*⟩

## 2.3 Building block: Sums

We prove that we can lift the ADS construction through sums.

**type-synonym** (*'a<sub>h</sub>, 'b<sub>h</sub>*) *sum<sub>h</sub>* = *'a<sub>h</sub> + 'b<sub>h</sub>*

**type-notation**  $sum_h$  (**infixr**  $\langle +_h \rangle$  10)

**type-synonym** ( $'a_m, 'b_m$ )  $sum_m = 'a_m + 'b_m$

— If a functor does not introduce blindable positions, then we don't need the type variable copies.

**type-notation**  $sum_m$  (**infixr**  $\langle +_m \rangle$  10)

### 2.3.1 Hashes

**abbreviation** (*input*)  $hash-sum' :: ('a_h +_h 'b_h, 'a_h +_h 'b_h) hash$  **where**  
 $hash-sum' \equiv id$

**abbreviation** (*input*)  $hash-sum :: ('a_m, 'a_h) hash \Rightarrow ('b_m, 'b_h) hash \Rightarrow ('a_m +_m 'b_m, 'a_h +_h 'b_h) hash$   
**where**  $hash-sum \equiv map-sum$

### 2.3.2 Blinding

**abbreviation** (*input*)  $blinding-of-sum :: 'a_m blinding-of \Rightarrow 'b_m blinding-of \Rightarrow ('a_m +_m 'b_m) blinding-of$  **where**  
 $blinding-of-sum \equiv rel-sum$

**lemmas**  $blinding-of-sum-mono = sum.rel-mono$

**lemma**  $blinding-of-sum-hash$ :

**assumes**  $boa \leq vimage2p\ rha\ rha (=) bob \leq vimage2p\ rhb\ rhb (=)$

**shows**  $blinding-of-sum\ boa\ bob \leq vimage2p\ (hash-sum\ rha\ rhb)\ (hash-sum\ rha\ rhb) (=)$

$\langle proof \rangle$

**lemma**  $blinding-of-on-sum$  [*locale-witness*]:

**assumes**  $blinding-of-on\ A\ rha\ boa\ blinding-of-on\ B\ rhb\ bob$

**shows**  $blinding-of-on\ \{x.\ setl\ x \subseteq A \wedge\ setr\ x \subseteq B\}\ (hash-sum\ rha\ rhb)\ (blinding-of-sum\ boa\ bob)$

(**is**  $blinding-of-on\ ?A\ ?h\ ?bo$ )

$\langle proof \rangle$

**lemmas**  $blinding-of-sum$  [*locale-witness*] =  $blinding-of-on-sum$ [*of UNIV - - UNIV, simplified*]

### 2.3.3 Merging

**context**

**fixes**  $ma :: 'a_m\ merge$

**fixes**  $mb :: 'b_m\ merge$

**begin**

**fun**  $merge-sum :: ('a_m +_m 'b_m)\ merge$  **where**

$merge-sum\ (Inl\ x)\ (Inl\ y) = map-option\ Inl\ (ma\ x\ y)$

|  $merge-sum\ (Inr\ x)\ (Inr\ y) = map-option\ Inr\ (mb\ x\ y)$

| *merge-sum* - - = *None*

**lemma** *merge-on-sum* [*locale-witness*]:

**assumes** *merge-on* *A rha boa ma merge-on B rhb bob mb*

**shows** *merge-on* {*x. setl x*  $\subseteq$  *A*  $\wedge$  *setr x*  $\subseteq$  *B*} (*hash-sum rha rhb*) (*blinding-of-sum* *boa bob*) *merge-sum*

(**is** *merge-on* ?*A* ?*h* ?*bo* ?*m*)

*<proof>*

**lemmas** *merge-sum* [*locale-witness*] = *merge-on-sum*[**where** *A=UNIV* **and** *B=UNIV*, *simplified*]

**lemma** *merge-sum-alt-def*:

*merge-sum x y* = (*case* (*x, y*) *of*

(*Inl x, Inl y*)  $\Rightarrow$  *map-option Inl (ma x y)*

| (*Inr x, Inr y*)  $\Rightarrow$  *map-option Inr (mb x y)*

| -  $\Rightarrow$  *None*)

*<proof>*

**end**

**lemma** *merge-sum-cong*[*fundef-cong*]:

$\llbracket x = x'; y = y';$

$\wedge xl\ yl. \llbracket x = Inl\ xl; y = Inl\ yl \rrbracket \Longrightarrow ma\ xl\ yl = ma'\ xl\ yl;$

$\wedge xr\ yr. \llbracket x = Inr\ xr; y = Inr\ yr \rrbracket \Longrightarrow mb\ xr\ yr = mb'\ xr\ yr \rrbracket \Longrightarrow$

*merge-sum ma mb x y* = *merge-sum ma' mb' x' y'*

*<proof>*

**parametric-constant** *merge-sum-parametric* [*transfer-rule*]: *merge-sum-alt-def*

### 2.3.4 Merkle interface

**lemma** *merkle-sum* [*locale-witness*]:

**assumes** *merkle-interface rha boa ma merkle-interface rhb bob mb*

**shows** *merkle-interface* (*hash-sum rha rhb*) (*blinding-of-sum boa bob*) (*merge-sum* *ma mb*)

*<proof>*

## 2.4 Building Block: Products

We prove that we can lift the ADS construction through products.

**type-synonym** (*'a<sub>h</sub>*, *'b<sub>h</sub>*) *prod<sub>h</sub>* = *'a<sub>h</sub>  $\times$  'b<sub>h</sub>*

**type-notation** *prod<sub>h</sub>* ( $\langle\langle - \times_h / - \rangle\rangle$  [*21*, *20*] *20*)

**type-synonym** (*'a<sub>m</sub>*, *'b<sub>m</sub>*) *prod<sub>m</sub>* = *'a<sub>m</sub>  $\times$  'b<sub>m</sub>*

— If a functor does not introduce blindable positions, then we don't need the type variable copies.

**type-notation** *prod<sub>m</sub>* ( $\langle\langle - \times_m / - \rangle\rangle$  [*21*, *20*] *20*)

### 2.4.1 Hashes

**abbreviation** (*input*) *hash-prod'* :: ('a<sub>h</sub> ×<sub>h</sub> 'b<sub>h</sub>, 'a<sub>h</sub> ×<sub>h</sub> 'b<sub>h</sub>) *hash* **where**  
*hash-prod'* ≡ *id*

**abbreviation** (*input*) *hash-prod* :: ('a<sub>m</sub>, 'a<sub>h</sub>) *hash* ⇒ ('b<sub>m</sub>, 'b<sub>h</sub>) *hash* ⇒ ('a<sub>m</sub> ×<sub>m</sub>  
'b<sub>m</sub>, 'a<sub>h</sub> ×<sub>h</sub> 'b<sub>h</sub>) *hash*  
**where** *hash-prod* ≡ *map-prod*

### 2.4.2 Blinding

**abbreviation** (*input*) *blinding-of-prod* :: 'a<sub>m</sub> *blinding-of* ⇒ 'b<sub>m</sub> *blinding-of* ⇒  
( 'a<sub>m</sub> ×<sub>m</sub> 'b<sub>m</sub>) *blinding-of* **where**  
*blinding-of-prod* ≡ *rel-prod*

**lemmas** *blinding-of-prod-mono* = *prod.rel-mono*

**lemma** *blinding-of-prod-hash*:

**assumes** *boa* ≤ *vimage2p rha rha* (=) *bob* ≤ *vimage2p rhb rhb* (=)  
**shows** *blinding-of-prod boa bob* ≤ *vimage2p (hash-prod rha rhb) (hash-prod rha rhb)* (=)  
{*proof*}

**lemma** *blinding-of-on-prod [locale-witness]*:

**assumes** *blinding-of-on A rha boa blinding-of-on B rhb bob*  
**shows** *blinding-of-on {x. fsts x ⊆ A ∧ snds x ⊆ B} (hash-prod rha rhb) (blinding-of-prod  
boa bob)*  
(**is** *blinding-of-on ?A ?h ?bo*)  
{*proof*}

**lemmas** *blinding-of-prod [locale-witness]* = *blinding-of-on-prod[where A=UNIV  
and B=UNIV, simplified]*

### 2.4.3 Merging

**context**

**fixes** *ma* :: 'a<sub>m</sub> *merge*

**fixes** *mb* :: 'b<sub>m</sub> *merge*

**begin**

**fun** *merge-prod* :: ('a<sub>m</sub> ×<sub>m</sub> 'b<sub>m</sub>) *merge* **where**

*merge-prod* (*x*, *y*) (*x'*, *y'*) = *Option.bind* (*ma x x'*) (λ*x''*. *map-option* (*Pair x''*)  
(*mb y y'*))

**lemma** *merge-on-prod [locale-witness]*:

**assumes** *merge-on A rha boa ma merge-on B rhb bob mb*  
**shows** *merge-on {x. fsts x ⊆ A ∧ snds x ⊆ B} (hash-prod rha rhb) (blinding-of-prod  
boa bob) merge-prod*  
(**is** *merge-on ?A ?h ?bo ?m*)  
{*proof*}

**lemmas** *merge-prod* [*locale-witness*] = *merge-on-prod*[**where**  $A=UNIV$  **and**  $B=UNIV$ , *simplified*]

**lemma** *merge-prod-alt-def*:

*merge-prod* =  $(\lambda(x, y) (x', y'). \text{Option.bind } (ma \ x \ x') (\lambda x''. \text{map-option } (\text{Pair } x'') (mb \ y \ y')))$   
 ⟨*proof*⟩

**end**

**lemma** *merge-prod-cong*[*fundef-cong*]:

**assumes**  $\bigwedge a \ b. \llbracket a \in \text{fst} \ p1; b \in \text{fst} \ p2 \rrbracket \implies ma \ a \ b = ma' \ a \ b$   
**and**  $\bigwedge a \ b. \llbracket a \in \text{snd} \ p1; b \in \text{snd} \ p2 \rrbracket \implies mb \ a \ b = mb' \ a \ b$   
**shows** *merge-prod*  $ma \ mb \ p1 \ p2 = \text{merge-prod } ma' \ mb' \ p1 \ p2$   
 ⟨*proof*⟩

**parametric-constant** *merge-prod-parametric* [*transfer-rule*]: *merge-prod-alt-def*

#### 2.4.4 Merkle Interface

**lemma** *merkle-product* [*locale-witness*]:

**assumes** *merkle-interface*  $rha \ boa \ ma \ merkle\text{-interface} \ rhb \ bob \ mb$   
**shows** *merkle-interface*  $(\text{hash-prod } rha \ rhb) (\text{blinding-of-prod } boa \ bob) (\text{merge-prod } ma \ mb)$   
 ⟨*proof*⟩

### 2.5 Building Block: Lists

The ADS construction on lists is done the easiest through a separate isomorphic datatype that has only a single constructor. We hide this construction in a locale.

**locale** *list-R1* **begin**

**type-synonym**  $('a, 'b) \text{list-F} = \text{unit} + 'a \times 'b$

**abbreviation**  $(\text{input}) \text{set-base-F}_m \equiv \lambda x. \text{setr } x \gg\gg \text{fst}$

**abbreviation**  $(\text{input}) \text{set-rec-F}_m \equiv \lambda A. \text{setr } A \gg\gg \text{snd}$

**abbreviation**  $(\text{input}) \text{map-F} \equiv \lambda fb \ fr. \text{map-sum } id (\text{map-prod } fb \ fr)$

**datatype**  $'a \text{list-R1} = \text{list-R1} (\text{unR}: ('a, 'a \text{list-R1}) \text{list-F})$

**lemma** *list-R1-const-into-dest*:  $\text{list-R1 } F = l \longleftrightarrow F = \text{unR } l$   
 ⟨*proof*⟩

**declare** *list-R1.split*[*split*]

**lemma** *list-R1-induct*[*case-names list-R1*]:

**assumes**  $\bigwedge F. \llbracket \bigwedge l'. l' \in \text{set-rec-F}_m \ F \implies P \ l' \rrbracket \implies P (\text{list-R1 } F)$

**shows**  $P\ l$   
 $\langle proof \rangle$

**lemma** *set-list-R1-eq*:  
 $\{x. \text{set-base-}F_m\ x \subseteq A \wedge \text{set-rec-}F_m\ x \subseteq B\} =$   
 $\{x. \text{setl}\ x \subseteq UNIV \wedge \text{setr}\ x \subseteq \{x. \text{fst}\ x \subseteq A \wedge \text{snd}\ x \subseteq B\}\}$   
 $\langle proof \rangle$

### 2.5.1 The Isomorphism

**primrec** (*transfer*) *list-R1-to-list* :: 'a list-R1  $\Rightarrow$  'a list **where**  
*list-R1-to-list* (*list-R1* l) = (case map-sum id (map-prod id *list-R1-to-list*) l of Inl  
( $\Rightarrow$  [] | Inr (x, xs)  $\Rightarrow$  x # xs)

**lemma** *list-R1-to-list-simps* [*simp*]:  
*list-R1-to-list* (*list-R1* (Inl ())) = []  
*list-R1-to-list* (*list-R1* (Inr (x, xs))) = x # *list-R1-to-list* xs  
 $\langle proof \rangle$

**declare** *list-R1-to-list.simps* [*simp del*]

**primrec** (*transfer*) *list-to-list-R1* :: 'a list  $\Rightarrow$  'a list-R1 **where**  
*list-to-list-R1* [] = *list-R1* (Inl ())  
| *list-to-list-R1* (x#xs) = *list-R1* (Inr (x, *list-to-list-R1* xs))

**lemma** *R1-of-list*: *list-R1-to-list* (*list-to-list-R1* x) = x  
 $\langle proof \rangle$

**lemma** *list-of-R1*: *list-to-list-R1* (*list-R1-to-list* x) = x  
 $\langle proof \rangle$

**lemma** *list-R1-def*: *type-definition* *list-to-list-R1* *list-R1-to-list* UNIV  
 $\langle proof \rangle$

**setup-lifting** *list-R1-def*

**lemma** *map-list-R1-list-to-list-R1*: *map-list-R1* f (*list-to-list-R1* xs) = *list-to-list-R1*  
(*map* f xs)  
 $\langle proof \rangle$

**lemma** *list-R1-map-trans* [*transfer-rule*]: **includes** *lifting-syntax* **shows**  
(((=)  $\implies$  (=))  $\implies$  pcr-list (=)  $\implies$  pcr-list (=)) *map-list-R1* map  
 $\langle proof \rangle$

**lemma** *set-list-R1-list-to-list-R1*: *set-list-R1* (*list-to-list-R1* xs) = set xs  
 $\langle proof \rangle$

**lemma** *list-R1-set-trans* [*transfer-rule*]: **includes** *lifting-syntax* **shows**  
(pcr-list (=)  $\implies$  (=)) *set-list-R1* set

*<proof>*

**lemma** *rel-list-R1-list-to-list-R1*:

*rel-list-R1 R (list-to-list-R1 xs) (list-to-list-R1 ys)  $\longleftrightarrow$  list-all2 R xs ys*  
*(is ?lhs  $\longleftrightarrow$  ?rhs)*

*<proof>*

**lemma** *list-R1-rel-trans[transfer-rule]*: **includes** *lifting-syntax* **shows**

*((=)  $\implies$  (=)  $\implies$  (=))  $\implies$  pcr-list (=)  $\implies$  pcr-list (=)  $\implies$   
(=) *rel-list-R1 list-all2**

*<proof>*

## 2.5.2 Hashes

**type-synonym** *('a<sub>h</sub>, 'b<sub>h</sub>) list-F<sub>h</sub>* = *unit +<sub>h</sub> 'a<sub>h</sub> ×<sub>h</sub> 'b<sub>h</sub>*

**type-synonym** *('a<sub>m</sub>, 'b<sub>m</sub>) list-F<sub>m</sub>* = *unit +<sub>m</sub> 'a<sub>m</sub> ×<sub>m</sub> 'b<sub>m</sub>*

**type-synonym** *'a<sub>h</sub> list-R1<sub>h</sub>* = *'a<sub>h</sub> list-R1*

— In theory, we should define a separate datatype here of the functor *('a<sub>h</sub>, -)* *list-F<sub>h</sub>*. We take a shortcut because they're isomorphic.

**type-synonym** *'a<sub>m</sub> list-R1<sub>m</sub>* = *'a<sub>m</sub> list-R1*

— In theory, we should define a separate datatype here of the functor *('a<sub>m</sub>, -)* *list-F<sub>m</sub>*. We take a shortcut because they're isomorphic.

**definition** *hash-F* :: *('a<sub>m</sub>, 'a<sub>h</sub>) hash  $\Rightarrow$  ('b<sub>m</sub>, 'b<sub>h</sub>) hash  $\Rightarrow$  (('a<sub>m</sub>, 'b<sub>m</sub>) list-F<sub>m</sub>, ('a<sub>h</sub>, 'b<sub>h</sub>) list-F<sub>h</sub>) hash* **where**

*hash-F h rhL = hash-sum hash-unit (hash-prod h rhL)*

**abbreviation** *(input) hash-R1* :: *('a<sub>m</sub>, 'a<sub>h</sub>) hash  $\Rightarrow$  ('a<sub>m</sub> list-R1<sub>m</sub>, 'a<sub>h</sub> list-R1<sub>h</sub>) hash* **where**

*hash-R1  $\equiv$  map-list-R1*

**parametric-constant** *hash-F-parametric[transfer-rule]*: *hash-F-def*

## 2.5.3 Blinding

**definition** *blinding-of-F* :: *'a<sub>m</sub> blinding-of  $\Rightarrow$  'b<sub>m</sub> blinding-of  $\Rightarrow$  ('a<sub>m</sub>, 'b<sub>m</sub>) list-F<sub>m</sub> blinding-of* **where**

*blinding-of-F bo bL = blinding-of-sum blinding-of-unit (blinding-of-prod bo bL)*

**abbreviation** *(input) blinding-of-R1* :: *'a blinding-of  $\Rightarrow$  'a list-R1 blinding-of* **where**

*blinding-of-R1  $\equiv$  rel-list-R1*

**lemma** *blinding-of-hash-R1*:

**assumes** *bo  $\leq$  vimage2p h h (=)*

**shows** *blinding-of-R1 bo  $\leq$  vimage2p (hash-R1 h) (hash-R1 h) (=)*

*<proof>*

**lemma** *blinding-of-on-R1* [*locale-witness*]:  
**assumes** *blinding-of-on A h bo*  
**shows** *blinding-of-on {x. set-list-R1 x  $\subseteq$  A} (hash-R1 h) (blinding-of-R1 bo)*  
*(is blinding-of-on ?A ?h ?bo)*  
*<proof>*

**lemmas** *blinding-of-R1* [*locale-witness*] = *blinding-of-on-R1* [**where** *A=UNIV, simplified*]

**parametric-constant** *blinding-of-F-parametric*[*transfer-rule*]: *blinding-of-F-def*

## 2.5.4 Merging

**definition** *merge-F* :: *'a<sub>m</sub> merge  $\Rightarrow$  'b<sub>m</sub> merge  $\Rightarrow$  ('a<sub>m</sub>, 'b<sub>m</sub>) list-F<sub>m</sub> merge* **where**

*merge-F m mL = merge-sum merge-unit (merge-prod m mL)*

**lemma** *merge-F-cong*[*fundef-cong*]:  
**assumes**  $\bigwedge a b. \llbracket a \in \text{set-base-F}_m x; b \in \text{set-base-F}_m y \rrbracket \Longrightarrow m a b = m' a b$   
**and**  $\bigwedge a b. \llbracket a \in \text{set-rec-F}_m x; b \in \text{set-rec-F}_m y \rrbracket \Longrightarrow mL a b = mL' a b$   
**shows** *merge-F m mL x y = merge-F m' mL' x y*  
*<proof>*

**context**

**fixes** *m* :: *'a<sub>m</sub> merge*

**notes** *setr.simps*[*simp*]

**begin**

**fun** *merge-R1* :: *'a<sub>m</sub> list-R1<sub>m</sub> merge* **where**

*merge-R1 (list-R1 l1) (list-R1 l2) = map-option list-R1 (merge-F m merge-R1 l1 l2)*

**end**

**case-of-simps** *merge-cases* [*simp*]: *merge-R1.simps*

**lemma** *merge-on-R1*:

**assumes** *merge-on A h bo m*

**shows** *merge-on {x. set-list-R1 x  $\subseteq$  A} (hash-R1 h) (blinding-of-R1 bo) (merge-R1 m)*

*(is merge-on ?A ?h ?bo ?m)*

*<proof>*

**lemmas** *merge-R1* [*locale-witness*] = *merge-on-R1* [**where** *A=UNIV, simplified*]

**lemma** *merkle-list-R1* [*locale-witness*]:

**assumes** *merkle-interface h bo m*

**shows** *merkle-interface (hash-R1 h) (blinding-of-R1 bo) (merge-R1 m)*

*<proof>*

**lemma** *merge-R1-cong* [*fundef-cong*]:  
**assumes**  $\bigwedge a b. \llbracket a \in \text{set-list-R1 } x; b \in \text{set-list-R1 } y \rrbracket \implies m a b = m' a b$   
**shows**  $\text{merge-R1 } m x y = \text{merge-R1 } m' x y$   
 $\langle \text{proof} \rangle$

**parametric-constant** *merge-F-parametric*[*transfer-rule*]: *merge-F-def*

**lemma** *merge-R1-parametric* [*transfer-rule*]:  
**includes** *lifting-syntax*  
**notes** [*simp del*] = *merge-cases*  
**assumes** [*transfer-rule*]: *bi-unique A*  
**shows**  $((A \implies A \implies \text{rel-option } A) \implies \text{rel-list-R1 } A \implies \text{rel-list-R1 } A \implies \text{rel-option } (\text{rel-list-R1 } A))$   
 $\text{merge-R1 } \text{merge-R1}$   
 $\langle \text{proof} \rangle$

**end**

## 2.5.5 Transferring the Constructions to Lists

**type-synonym**  $'a_h \text{ list}_h = 'a_h \text{ list}$   
**type-synonym**  $'a_m \text{ list}_m = 'a_m \text{ list}$

**context begin**

**interpretation** *list-R1*  $\langle \text{proof} \rangle$

**abbreviation** (*input*) *hash-list* ::  $('a_m, 'a_h) \text{ hash} \Rightarrow ('a_m \text{ list}_m, 'a_h \text{ list}_h) \text{ hash}$   
**where**  $\text{hash-list} \equiv \text{map}$

**abbreviation** (*input*) *blinding-of-list* ::  $'a_m \text{ blinding-of} \Rightarrow 'a_m \text{ list}_m \text{ blinding-of}$   
**where**  $\text{blinding-of-list} \equiv \text{list-all2}$

**lift-definition** *merge-list* ::  $'a_m \text{ merge} \Rightarrow 'a_m \text{ list}_m \text{ merge}$  **is** *merge-R1*  $\langle \text{proof} \rangle$

**lemma** *blinding-of-list-mono*:

$\llbracket \bigwedge x y. \text{bo } x y \longrightarrow \text{bo}' x y \rrbracket \implies$   
 $\text{blinding-of-list } \text{bo } x y \longrightarrow \text{blinding-of-list } \text{bo}' x y$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{blinding-of-list-hash} = \text{blinding-of-hash-R1}[\text{Transfer.transferred}]$   
**and**  $\text{blinding-of-on-list} [\text{locale-witness}] = \text{blinding-of-on-R1}[\text{Transfer.transferred}]$   
**and**  $\text{blinding-of-list} [\text{locale-witness}] = \text{blinding-of-R1}[\text{Transfer.transferred}]$   
**and**  $\text{merge-on-list} [\text{locale-witness}] = \text{merge-on-R1}[\text{Transfer.transferred}]$   
**and**  $\text{merge-list} [\text{locale-witness}] = \text{merge-R1}[\text{Transfer.transferred}]$   
**and**  $\text{merge-list-cong} = \text{merge-R1-cong}[\text{Transfer.transferred}]$

**lemma** *blinding-of-list-mono-pred*:

$R \leq R' \implies \text{blinding-of-list } R \leq \text{blinding-of-list } R'$   
 $\langle \text{proof} \rangle$

**lemma** *blinding-of-list-simp*:  $\text{blinding-of-list} = \text{list-all2}$

*<proof>*

**lemma** *merkle-list* [*locale-witness*]:

**assumes** [*locale-witness*]: *merkle-interface h bo m*

**shows** *merkle-interface (hash-list h) (blinding-of-list bo) (merge-list m)*

*<proof>*

**parametric-constant** *merge-list-parametric* [*transfer-rule*]: *merge-list-def*

**lifting-update** *list.lifting*

**lifting-forget** *list.lifting*

**end**

## 2.6 Building block: function space

We prove that we can lift the ADS construction through functions.

**type-synonym** (*'a*, *'b<sub>h</sub>*) *fun<sub>h</sub>* = *'a*  $\Rightarrow$  *'b<sub>h</sub>*

**type-notation** *fun<sub>h</sub>* (**infixr**  $\langle \Rightarrow_h \rangle$  0)

**type-synonym** (*'a*, *'b<sub>m</sub>*) *fun<sub>m</sub>* = *'a*  $\Rightarrow$  *'b<sub>m</sub>*

**type-notation** *fun<sub>m</sub>* (**infixr**  $\langle \Rightarrow_m \rangle$  0)

### 2.6.1 Hashes

Only the range is live, the domain is dead like for BNFs.

**abbreviation** (*input*) *hash-fun'* :: (*'a*  $\Rightarrow_m$  *'b<sub>h</sub>*, *'a*  $\Rightarrow_h$  *'b<sub>h</sub>*) *hash* **where**

*hash-fun'*  $\equiv$  *id*

**abbreviation** (*input*) *hash-fun* :: (*'b<sub>m</sub>*, *'b<sub>h</sub>*) *hash*  $\Rightarrow$  (*'a*  $\Rightarrow_m$  *'b<sub>m</sub>*, *'a*  $\Rightarrow_h$  *'b<sub>h</sub>*) *hash*

**where** *hash-fun*  $\equiv$  *comp*

### 2.6.2 Blinding

**abbreviation** (*input*) *blinding-of-fun* :: *'b<sub>m</sub>* *blinding-of*  $\Rightarrow$  (*'a*  $\Rightarrow_m$  *'b<sub>m</sub>*) *blinding-of*

**where**

*blinding-of-fun*  $\equiv$  *rel-fun* (=)

**lemmas** *blinding-of-fun-mono* = *fun.rel-mono*

**lemma** *blinding-of-fun-hash*:

**assumes** *bo*  $\leq$  *vimage2p rh rh* (=)

**shows** *blinding-of-fun bo*  $\leq$  *vimage2p (hash-fun rh) (hash-fun rh)* (=)

*<proof>*

**lemma** *blinding-of-on-fun* [*locale-witness*]:

**assumes** *blinding-of-on A rh bo*

**shows** *blinding-of-on*  $\{x. \text{range } x \subseteq A\}$  (*hash-fun rh*) (*blinding-of-fun bo*)

(**is blinding-of-on** ?A ?h ?bo)  
 ⟨proof⟩

**lemmas** *blinding-of-fun* [locale-witness] = *blinding-of-on-fun*[**where** A=UNIV, simplified]

### 2.6.3 Merging

**context**

**fixes**  $m :: 'b_m \text{ merge}$

**begin**

**definition** *merge-fun* :: ( $'a \Rightarrow_m 'b_m$ ) *merge* **where**

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

**lemma** *merge-on-fun* [locale-witness]:

**assumes** *merge-on* A rh bo m

**shows** *merge-on* { $x. \text{range}\ x \subseteq A$ } (*hash-fun* rh) (*blinding-of-fun* bo) *merge-fun*

(**is** *merge-on* ?A ?h ?bo ?m)

⟨proof⟩

**lemmas** *merge-fun* [locale-witness] = *merge-on-fun*[**where** A=UNIV, simplified]

**end**

**lemma** *merge-fun-cong*[fundef-cong]:

**assumes**  $\bigwedge a\ b. \llbracket a \in \text{range}\ f; b \in \text{range}\ g \rrbracket \implies m\ a\ b = m'\ a\ b$

**shows** *merge-fun*  $m\ f\ g = \text{merge-fun}\ m'\ f\ g$

⟨proof⟩

**lemma** *is-none-alt-def*: *Option.is-none*  $x \longleftrightarrow (\text{case}\ x\ \text{of}\ \text{None} \Rightarrow \text{True} \mid \text{Some}\ - \Rightarrow \text{False})$

⟨proof⟩

**parametric-constant** *is-none-parametric* [transfer-rule]: *is-none-alt-def*

**lemma** *merge-fun-parametric* [transfer-rule]: **includes** *lifting-syntax* **shows**

$((A \text{====>} B \text{====>} \text{rel-option}\ C) \text{====>} ((=) \text{====>} A) \text{====>} ((=) \text{====>} B) \text{====>} \text{rel-option}\ ((=) \text{====>} C))$

*merge-fun* *merge-fun*

⟨proof⟩

### 2.6.4 Merkle Interface

**lemma** *merkle-fun* [locale-witness]:

**assumes** *merkle-interface* rh bo m

**shows** *merkle-interface* (*hash-fun* rh) (*blinding-of-fun* bo) (*merge-fun* m)

⟨proof⟩

## 2.7 Rose trees

We now define an ADS over rose trees, which is like a arbitrarily branching Merkle tree where each node in the tree can be blinded, including the root. The number of children and the position of a child among its siblings cannot be hidden. The construction allows to plug in further blindable positions in the labels of the nodes.

**type-synonym**  $( 'a, 'b )$  *rose-tree-F* =  $'a \times 'b$  list

**abbreviation** *(input)* *map-rose-tree-F* **where**  
 $map-rose-tree-F\ f1\ f2 \equiv map-prod\ f1\ (map\ f2)$

**definition** *map-rose-tree-F-const* **where**  
 $map-rose-tree-F-const\ f1\ f2 \equiv map-rose-tree-F\ f1\ f2$

**datatype**  $'a$  *rose-tree* =  $Tree\ ('a, 'a\ rose-tree)\ rose-tree-F$

**type-synonym**  $( 'a_h, 'b_h )$  *rose-tree-F<sub>h</sub>* =  $( 'a_h \times_h 'b_h\ list_h )\ blindable_h$

**datatype**  $'a_h$  *rose-tree<sub>h</sub>* =  $Tree_h\ ('a_h, 'a_h\ rose-tree_h)\ rose-tree-F_h$

**type-synonym**  $( 'a_m, 'a_h, 'b_m, 'b_h )$  *rose-tree-F<sub>m</sub>* =  $( 'a_m \times_m 'b_m\ list_m, 'a_h \times_h 'b_h\ list_h )\ blindable_m$

**datatype**  $( 'a_m, 'a_h )$  *rose-tree<sub>m</sub>* =  $Tree_m\ ('a_m, 'a_h, ('a_m, 'a_h)\ rose-tree_m, 'a_h\ rose-tree_h)\ rose-tree-F_m$

**abbreviation** *(input)* *map-rose-tree-F<sub>m</sub>*  
 $:: ('ma \Rightarrow 'a) \Rightarrow ('mr \Rightarrow 'r) \Rightarrow ('ma, 'ha, 'mr, 'hr)\ rose-tree-F_m \Rightarrow ('a, 'ha, 'r, 'hr)\ rose-tree-F_m$

**where**  
 $map-rose-tree-F_m\ f\ g \equiv map-blindable_m\ (map-prod\ f\ (map\ g))\ id$

### 2.7.1 Hashes

**abbreviation** *(input)* *hash-rt-F'*  
 $:: (( 'a_h, 'a_h, 'b_h, 'b_h )\ rose-tree-F_m, ('a_h, 'b_h)\ rose-tree-F_h)\ hash$

**where**  
 $hash-rt-F' \equiv hash-blindable\ id$

**definition** *hash-rt-F<sub>m</sub>*  
 $:: ('a_m, 'a_h)\ hash \Rightarrow ('b_m, 'b_h)\ hash \Rightarrow$   
 $(( 'a_m, 'a_h, 'b_m, 'b_h )\ rose-tree-F_m, ('a_h, 'b_h)\ rose-tree-F_h)\ hash$  **where**  
 $hash-rt-F_m\ h\ rhm \equiv hash-rt-F'\ o\ map-rose-tree-F_m\ h\ rhm$

**lemma** *hash-rt-F<sub>m</sub>-alt-def*:  $hash-rt-F_m\ h\ rhm = hash-blindable\ (map-prod\ h\ (map\ rhm))$   
 $\langle proof \rangle$

**primrec** *(transfer)* *hash-rt-tree'*

$:: (('a_h, 'a_h) \text{ rose-tree}_m, 'a_h \text{ rose-tree}_h) \text{ hash where}$   
 $\text{hash-rt-tree}' (\text{Tree}_m x) = \text{Tree}_h (\text{hash-rt-F}' (\text{map-rose-tree-F}_m \text{ id hash-rt-tree}'$   
 $x))$

**definition** *hash-tree*

$:: ('a_m, 'a_h) \text{ hash} \Rightarrow (('a_m, 'a_h) \text{ rose-tree}_m, 'a_h \text{ rose-tree}_h) \text{ hash where}$   
 $\text{hash-tree } h = \text{hash-rt-tree}' \circ \text{map-rose-tree}_m \text{ h id}$

**lemma** *blindable<sub>m</sub>-map-compositionality*:

$\text{map-blindable}_m f g \circ \text{map-blindable}_m f' g' = \text{map-blindable}_m (f \circ f') (g \circ g')$   
 $\langle \text{proof} \rangle$

**lemma** *hash-tree-simps [simp]*:

$\text{hash-tree } h (\text{Tree}_m x) = \text{Tree}_h (\text{hash-rt-F}_m h (\text{hash-tree } h) x)$   
 $\langle \text{proof} \rangle$

**parametric-constant** *hash-rt-F<sub>m</sub>-parametric [transfer-rule]: hash-rt-F<sub>m</sub>-alt-def*

**parametric-constant** *hash-tree-parametric [transfer-rule]: hash-tree-def*

## 2.7.2 Blinding

**abbreviation** (*input*) *blinding-of-rt-F<sub>m</sub>*

$:: ('a_m, 'a_h) \text{ hash} \Rightarrow 'a_m \text{ blinding-of} \Rightarrow ('b_m, 'b_h) \text{ hash} \Rightarrow 'b_m \text{ blinding-of}$   
 $\Rightarrow ('a_m, 'a_h, 'b_m, 'b_h) \text{ rose-tree-F}_m \text{ blinding-of where}$   
 $\text{blinding-of-rt-F}_m \text{ ha boa hb bob} \equiv \text{blinding-of-blindable} (\text{hash-prod ha} (\text{map hb}))$   
 $(\text{blinding-of-prod boa} (\text{blinding-of-list bob}))$

**lemma** *blinding-of-rt-F<sub>m</sub>-mono*:

$\llbracket \text{boa} \leq \text{boa}'; \text{bob} \leq \text{bob}' \rrbracket \Longrightarrow \text{blinding-of-rt-F}_m \text{ ha boa hb bob} \leq \text{blinding-of-rt-F}_m$   
 $\text{ha boa}' \text{ hb bob}'$   
 $\langle \text{proof} \rangle$

**lemma** *blinding-of-rt-F<sub>m</sub>-mono-inductive*:

**assumes**  $\bigwedge x y. \text{boa } x y \longrightarrow \text{boa}' x y \bigwedge x y. \text{bob } x y \longrightarrow \text{bob}' x y$   
**shows**  $\text{blinding-of-rt-F}_m \text{ ha boa hb bob } x y \longrightarrow \text{blinding-of-rt-F}_m \text{ ha boa}' \text{ hb bob}'$   
 $x y$   
 $\langle \text{proof} \rangle$

**context**

**fixes**  $h :: ('a_m, 'a_h) \text{ hash}$

**and**  $bo :: 'a_m \text{ blinding-of}$

**begin**

**inductive** *blinding-of-tree*  $:: ('a_m, 'a_h) \text{ rose-tree}_m \text{ blinding-of where}$

$\text{blinding-of-tree} (\text{Tree}_m t1) (\text{Tree}_m t2)$

**if**  $\text{blinding-of-rt-F}_m h bo (\text{hash-tree } h) \text{ blinding-of-tree } t1 t2$

**monos** *blinding-of-rt-F<sub>m</sub>-mono-inductive*

**end**

**inductive-simps** *blinding-of-tree-simps* [*simp*]:

*blinding-of-tree* *h bo* (*Tree<sub>m</sub> t1*) (*Tree<sub>m</sub> t2*)

**lemma** *blinding-of-rt-F<sub>m</sub>-hash*:

**assumes** *boa*  $\leq$  *vimage2p ha ha* (=) *bob*  $\leq$  *vimage2p hb hb* (=)

**shows** *blinding-of-rt-F<sub>m</sub> ha* *boa hb bob*  $\leq$  *vimage2p (hash-rt-F<sub>m</sub> ha hb)* (*hash-rt-F<sub>m</sub> ha hb*) (=)

*<proof>*

**lemma** *blinding-of-tree-hash*:

**assumes** *bo*  $\leq$  *vimage2p h h* (=)

**shows** *blinding-of-tree h bo*  $\leq$  *vimage2p (hash-tree h)* (*hash-tree h*) (=)

*<proof>*

**abbreviation** (*input*) *set1-rt-F<sub>m</sub>* :: (*'a<sub>m</sub>, 'a<sub>h</sub>, 'b<sub>h</sub>, 'b<sub>m</sub>*) *rose-tree-F<sub>m</sub>*  $\Rightarrow$  *'a<sub>m</sub> set*  
**where**

*set1-rt-F<sub>m</sub> x*  $\equiv$  *set1-blindable<sub>m</sub> x*  $\ggg$  *fsts*

**abbreviation** (*input*) *set3-rt-F<sub>m</sub>* :: (*'a<sub>m</sub>, 'a<sub>h</sub>, 'b<sub>m</sub>, 'b<sub>h</sub>*) *rose-tree-F<sub>m</sub>*  $\Rightarrow$  *'b<sub>m</sub> set*  
**where**

*set3-rt-F<sub>m</sub> x*  $\equiv$  (*set1-blindable<sub>m</sub> x*  $\ggg$  *snds*)  $\ggg$  *set*

**lemma** *set-rt-F<sub>m</sub>-eq*:

$\{x. \text{set1-rt-F}_m x \subseteq A \wedge \text{set3-rt-F}_m x \subseteq B\} =$

$\{x. \text{set1-blindable}_m x \subseteq \{x. \text{fsts } x \subseteq A \wedge \text{snds } x \subseteq \{x. \text{set } x \subseteq B\}\}$

*<proof>*

**lemma** *hash-blindable-map*: *hash-blindable f*  $\circ$  *map-blindable<sub>m</sub> g id* = *hash-blindable (f*  $\circ$  *g)*

*<proof>*

**lemma** *blinding-of-on-tree* [*locale-witness*]:

**assumes** *blinding-of-on A h bo*

**shows** *blinding-of-on*  $\{x. \text{set1-rose-tree}_m x \subseteq A\}$  (*hash-tree h*) (*blinding-of-tree h bo*)

(**is** *blinding-of-on ?A ?h ?bo*)

*<proof>*

**lemmas** *blinding-of-tree* [*locale-witness*] = *blinding-of-on-tree*[**where** *A=UNIV*, *simplified*]

**lemma** *blinding-of-tree-mono*:

*bo*  $\leq$  *bo'*  $\implies$  *blinding-of-tree h bo*  $\leq$  *blinding-of-tree h bo'*

*<proof>*

### 2.7.3 Merging

**definition** *merge-rt- $F_m$*

$:: ('a_m, 'a_h) \text{hash} \Rightarrow 'a_m \text{merge} \Rightarrow ('b_m, 'b_h) \text{hash} \Rightarrow 'b_m \text{merge} \Rightarrow$   
 $('a_m, 'a_h, 'b_m, 'b_h) \text{rose-tree-}F_m \text{merge}$

**where**

$\text{merge-rt-}F_m \text{ha ma hr mr} \equiv \text{merge-blindable} (\text{hash-prod ha} (\text{hash-list hr})) (\text{merge-prod}$   
 $\text{ma} (\text{merge-list mr}))$

**lemma** *merge-rt- $F_m$ -cong* [*fundef-cong*]:

**assumes**  $\bigwedge a b. \llbracket a \in \text{set1-rt-}F_m x; b \in \text{set1-rt-}F_m y \rrbracket \Longrightarrow \text{ma } a \ b = \text{ma}' a \ b$

**and**  $\bigwedge a b. \llbracket a \in \text{set3-rt-}F_m x; b \in \text{set3-rt-}F_m y \rrbracket \Longrightarrow \text{mm } a \ b = \text{mm}' a \ b$

**shows**  $\text{merge-rt-}F_m \text{ha ma hm mm } x \ y = \text{merge-rt-}F_m \text{ha ma}' \text{hm mm}' x \ y$

*<proof>*

**lemma** *in-set1-blindable $_m$ -iff*:  $x \in \text{set1-blindable}_m y \longleftrightarrow y = \text{Unblinded } x$

*<proof>*

**context**

**fixes**  $h :: ('a_m, 'a_h) \text{hash}$

**and**  $\text{ma} :: 'a_m \text{merge}$

**notes** *in-set1-blindable $_m$ -iff*[*simp*]

**begin**

**fun** *merge-tree*  $:: ('a_m, 'a_h) \text{rose-tree}_m \text{merge}$  **where**

$\text{merge-tree} (\text{Tree}_m x) (\text{Tree}_m y) = \text{map-option } \text{Tree}_m (\text{merge-rt-}F_m \ h \ \text{ma} \ (\text{hash-tree } h) \ \text{merge-tree } x \ y)$

**end**

**lemma** *merge-on-tree* [*locale-witness*]:

**assumes** *merge-on*  $A \ h \ \text{bo} \ m$

**shows** *merge-on*  $\{x. \text{set1-rose-tree}_m x \subseteq A\} (\text{hash-tree } h) (\text{blinding-of-tree } h \ \text{bo})$   
 $(\text{merge-tree } h \ m)$

**(is** *merge-on*  $?A \ ?h \ ?bo \ ?m)$

*<proof>*

**lemmas** *merge-tree* [*locale-witness*] = *merge-on-tree*[**where**  $A = \text{UNIV}$ , *simplified*]

**lemma** *option-bind-comm*:

$((x :: 'a \ \text{option}) \gg (\lambda y. c \gg (\lambda z. f \ y \ z))) = (c \gg (\lambda y. x \gg (\lambda z. f \ z \ y)))$

*<proof>*

**parametric-constant** *merge-rt- $F_m$ -parametric* [*transfer-rule*]: *merge-rt- $F_m$ -def*

### 2.7.4 Merkle interface

**lemma** *merkle-tree* [*locale-witness*]:

**assumes** *merkle-interface*  $h \ \text{bo} \ m$

**shows** *merkle-interface*  $(\text{hash-tree } h) (\text{blinding-of-tree } h \ \text{bo}) (\text{merge-tree } h \ m)$

*<proof>*

```

lemma merge-tree-cong [fundef-cong]:
  assumes  $\bigwedge a\ b. \llbracket a \in \text{set1-rose-tree}_m\ x; b \in \text{set1-rose-tree}_m\ y \rrbracket \implies m\ a\ b = m'$ 
  a b
  shows merge-tree h m x y = merge-tree h m' x y
  <proof>

end

```

```

theory Generic-ADS-Construction imports
  Merkle-Interface
  HOL-Library.BNF-Axiomatization
begin

```

### 3 Generic construction of authenticated data structures

#### 3.1 Functors

##### 3.1.1 Source functor

We want to allow ADSs of arbitrary ADTs, which we call "source trees". The ADTs we are interested in can in general be represented as the least fixpoints of some bounded natural (bi-)functor (BNF)  $(\prime a, \prime b) F$ , where  $\prime a$  is the type of "source" data, and  $\prime b$  is a recursion "handle". However, Isabelle's type system does not support higher kinds, necessary to parameterize our definitions over functors. Instead, we first develop a general theory of ADSs over an arbitrary, but fixed functor, and its least fixpoint. We show that the theory is compositional, in that the functor's least fixed point can then be reused as the "source" data of another functor.

We start by defining the arbitrary fixed functor, its fixpoints, and showing how composition can be done. A higher-level explanation is found in the paper.

```

bnf-axiomatization  $(\prime a, \prime b) F$  [wits: 'a  $\Rightarrow$   $(\prime a, \prime b) F$ ]

```

```

context notes [[[typedef-overloaded]]] begin
datatype  $\prime a\ T = T\ (\prime a, \prime a\ T) F$ 
end

```

##### 3.1.2 Base Merkle functor

This type captures the ADS hashes.

```

bnf-axiomatization  $(\prime a, \prime b) F_h$  [wits: 'a  $\Rightarrow$   $(\prime a, \prime b) F_h$ ]

```

It intuitively contains mixed garbage and source values. The functor's recursive handle  $\prime b$  might contain partial garbage.

This type captures the ADS inclusion proofs. The functor  $(\prime a, \prime a', \prime b, \prime b')$   $F_m$  has all type variables doubled. This type represents all values including the information which parts are blinded. The original type variable  $\prime a$  now represents the source data, which for compositionality can contain blindable positions. The type  $\prime b$  is a recursive handle to inclusion sub-proofs (which can be partially blinded). The type  $\prime a'$  represent "hashes" of the source data in  $\prime a$ , i.e., a mix of source values and garbage. The type  $\prime b'$  is a recursive handle to ADS hashes of subtrees.

The corresponding type of recursive authenticated trees is then a fixpoint of this functor.

**bnf-axiomatization**  $(\prime a_m, \prime a_h, \prime b_m, \prime b_h) F_m$  [*wits*:  $\prime a_m \Rightarrow \prime a_h \Rightarrow \prime b_h \Rightarrow (\prime a_m, \prime a_h, \prime b_m, \prime b_h) F_m$ ]

### 3.1.3 Least fixpoint

**context notes**  $[[\textit{typedef-overloaded}]]$  **begin**  
**datatype**  $\prime a_h T_h = T_h (\prime a_h, \prime a_h T_h) F_h$   
**end**

**context notes**  $[[\textit{typedef-overloaded}]]$  **begin**  
**datatype**  $(\prime a_m, \prime a_h) T_m = T_m (\textit{the-}T_m: (\prime a_m, \prime a_h, (\prime a_m, \prime a_h) T_m, \prime a_h T_h) F_m)$   
**end**

### 3.1.4 Composition

Finally, we show how to compose two Merkle functors. For simplicity, we reuse  $(\prime a, \prime b) F$  and  $\prime a T$ .

**context notes**  $[[\textit{typedef-overloaded}]]$  **begin**

**datatype**  $(\prime a, \prime b) G = G (\prime a T, \prime b) F$

**datatype**  $(\prime a_h, \prime b_h) G_h = G_h (\textit{the-}G_h: (\prime a_h T_h, \prime b_h) F_h)$

**datatype**  $(\prime a_m, \prime a_h, \prime b_m, \prime b_h) G_m = G_m (\textit{the-}G_m: ((\prime a_m, \prime a_h) T_m, \prime a_h T_h, \prime b_m, \prime b_h) F_m)$

**end**

## 3.2 Root hash

### 3.2.1 Base functor

The root hash of an authenticated value is modelled as a blindable value of type  $(\prime a', \prime b') F_h$ . (Actually, we want to use an abstract datatype for root hashes, but we omit this distinction here for simplicity.)

**consts**  $\textit{root-hash-}F' :: ((\prime a_h, \prime a_h, \prime b_h, \prime b_h) F_m, (\prime a_h, \prime b_h) F_h) \textit{hash}$

— Root hash operation where we assume that all atoms have already been replaced by root hashes. This assumption is reflected in the equality of the type parameters of  $F_m$

**type-synonym**  $( 'a_m, 'a_h, 'b_m, 'b_h ) \text{ hash-F} =$   
 $( 'a_m, 'a_h ) \text{ hash} \Rightarrow ( 'b_m, 'b_h ) \text{ hash} \Rightarrow (( 'a_m, 'a_h, 'b_m, 'b_h ) F_m, ( 'a_h, 'b_h ) F_h)$   
 $\text{hash}$

**definition**  $\text{root-hash-F} :: ( 'a_m, 'a_h, 'b_m, 'b_h ) \text{ hash-F}$  **where**  
 $\text{root-hash-F rha rhb} = \text{root-hash-F}' \circ \text{map-F}_m \text{ rha id rhb id}$

### 3.2.2 Least fixpoint

**primrec**  $\text{root-hash-T}' :: (( 'a_h, 'a_h ) T_m, 'a_h T_h) \text{ hash}$  **where**  
 $\text{root-hash-T}' (T_m x) = T_h (\text{root-hash-F}' (\text{map-F}_m \text{id id root-hash-T}' \text{id } x))$

**definition**  $\text{root-hash-T} :: ( 'a_m, 'a_h ) \text{ hash} \Rightarrow (( 'a_m, 'a_h ) T_m, 'a_h T_h) \text{ hash}$  **where**  
 $\text{root-hash-T rha} = \text{root-hash-T}' \circ \text{map-T}_m \text{ rha id}$

**lemma**  $\text{root-hash-T-simps}$  [simp]:  
 $\text{root-hash-T rha} (T_m x) = T_h (\text{root-hash-F rha} (\text{root-hash-T rha } x))$   
 $\langle \text{proof} \rangle$

### 3.2.3 Composition

**primrec**  $\text{root-hash-G}' :: (( 'a_h, 'a_h, 'b_h, 'b_h ) G_m, ( 'a_h, 'b_h ) G_h) \text{ hash}$  **where**  
 $\text{root-hash-G}' (G_m x) = G_h (\text{root-hash-F}' (\text{map-F}_m \text{root-hash-T}' \text{id id id } x))$

**definition**  $\text{root-hash-G} :: ( 'a_m, 'a_h ) \text{ hash} \Rightarrow ( 'b_m, 'b_h ) \text{ hash} \Rightarrow (( 'a_m, 'a_h, 'b_m,$   
 $'b_h ) G_m, ( 'a_h, 'b_h ) G_h) \text{ hash}$  **where**  
 $\text{root-hash-G rha rhb} = \text{root-hash-G}' \circ \text{map-G}_m \text{ rha id rhb id}$

**lemma**  $\text{root-hash-G-unfold}$ :  
 $\text{root-hash-G rha rhb} = G_h \circ \text{root-hash-F} (\text{root-hash-T rha } rhb) \circ \text{the-G}_m$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{root-hash-G-simps}$  [simp]:  
 $\text{root-hash-G rha rhb} (G_m x) = G_h (\text{root-hash-F} (\text{root-hash-T rha } rhb x))$   
 $\langle \text{proof} \rangle$

## 3.3 Blinding relation

The blinding relation determines whether one ADS value is a blinding of another.

### 3.3.1 Blinding on the base functor ( $F_m$ )

**type-synonym**  $( 'a_m, 'a_h, 'b_m, 'b_h ) \text{ blinding-of-F} =$   
 $( 'a_m, 'a_h ) \text{ hash} \Rightarrow 'a_m \text{ blinding-of} \Rightarrow ( 'b_m, 'b_h ) \text{ hash} \Rightarrow 'b_m \text{ blinding-of} \Rightarrow ( 'a_m,$   
 $'a_h, 'b_m, 'b_h ) F_m \text{ blinding-of}$

— Computes whether a partially blinded ADS is a blinding of another one

**axiomatization** *blinding-of-F* :: ('a<sub>m</sub>, 'a<sub>h</sub>, 'b<sub>m</sub>, 'b<sub>h</sub>) *blinding-of-F* **where**  
*blinding-of-F-mono*:  $\llbracket \text{boa} \leq \text{boa}'; \text{bob} \leq \text{bob}' \rrbracket$   
 $\implies \text{blinding-of-F rha } \text{boa rhb } \text{bob} \leq \text{blinding-of-F rha } \text{boa}' \text{ rhb } \text{bob}'$   
 — Monotonicity must be unconditional (without the assumption *blinding-of-on*)  
 such that we can justify the recursive definition for the least fixpoint.  
**and** *blinding-respects-hashes-F* [*locale-witness*]:  
 $\llbracket \text{blinding-respects-hashes rha } \text{boa}; \text{blinding-respects-hashes rhb } \text{bob} \rrbracket$   
 $\implies \text{blinding-respects-hashes (root-hash-F rha rhb) (blinding-of-F rha } \text{boa rhb } \text{bob)}$   
**and** *blinding-of-on-F* [*locale-witness*]:  
 $\llbracket \text{blinding-of-on } A \text{ rha } \text{boa}; \text{blinding-of-on } B \text{ rhb } \text{bob} \rrbracket$   
 $\implies \text{blinding-of-on } \{x. \text{set1-F}_m x \subseteq A \wedge \text{set3-F}_m x \subseteq B\} \text{ (root-hash-F rha rhb)}$   
 (*blinding-of-F rha } \text{boa rhb } \text{bob}*)

**lemma** *blinding-of-F-mono-inductive*:  
**assumes** *a*:  $\bigwedge x y. \text{boa } x y \longrightarrow \text{boa}' x y$   
**and** *b*:  $\bigwedge x y. \text{bob } x y \longrightarrow \text{bob}' x y$   
**shows** *blinding-of-F rha } \text{boa rhb } \text{bob } x y \longrightarrow \text{blinding-of-F rha } \text{boa}' \text{ rhb } \text{bob}' x y*  
 ⟨*proof*⟩

### 3.3.2 Blinding on least fixpoints

**context**  
**fixes** *rh* :: ('a<sub>m</sub>, 'a<sub>h</sub>) *hash*  
**and** *bo* :: 'a<sub>m</sub> *blinding-of*  
**begin**

**inductive** *blinding-of-T* :: ('a<sub>m</sub>, 'a<sub>h</sub>) *T<sub>m</sub>* *blinding-of* **where**  
*blinding-of-T (T<sub>m</sub> x) (T<sub>m</sub> y)* **if**  
*blinding-of-F rh bo (root-hash-T rh) blinding-of-T x y*  
**monos** *blinding-of-F-mono-inductive*

**end**

**lemma** *blinding-of-T-mono*:  
**assumes** *bo* ≤ *bo'*  
**shows** *blinding-of-T rh bo* ≤ *blinding-of-T rh bo'*  
 ⟨*proof*⟩

**lemma** *blinding-of-T-root-hash*:  
**assumes** *bo* ≤ *vimage2p rh rh (=)*  
**shows** *blinding-of-T rh bo* ≤ *vimage2p (root-hash-T rh) (root-hash-T rh) (=)*  
 ⟨*proof*⟩

**lemma** *blinding-respects-hashes-T* [*locale-witness*]:  
*blinding-respects-hashes rh bo*  $\implies$  *blinding-respects-hashes (root-hash-T rh) (blinding-of-T rh bo)*  
 ⟨*proof*⟩

**lemma** *blinding-of-on-T* [*locale-witness*]:  
**assumes** *blinding-of-on A rh bo*  
**shows** *blinding-of-on*  $\{x. \text{set1-}T_m x \subseteq A\}$  (*root-hash-T rh*) (*blinding-of-T rh bo*)  
(is *blinding-of-on ?A ?h ?bo*)  
⟨*proof*⟩

**lemmas** *blinding-of-T* [*locale-witness*] = *blinding-of-on-T*[**where**  $A=UNIV$ , *simplified*]

### 3.3.3 Blinding on composition

**context**

**fixes** *rha* ::  $('a_m, 'a_h)$  *hash*  
**and** *boa* ::  $'a_m$  *blinding-of*  
**and** *rhb* ::  $('b_m, 'b_h)$  *hash*  
**and** *bob* ::  $'b_m$  *blinding-of*

**begin**

**inductive** *blinding-of-G* ::  $('a_m, 'a_h, 'b_m, 'b_h)$   $G_m$  *blinding-of* **where**  
*blinding-of-G* ( $G_m x$ ) ( $G_m y$ ) **if**  
*blinding-of-F* (*root-hash-T rha*) (*blinding-of-T rha boa*) *rhb bob x y*

**lemma** *blinding-of-G-unfold*:

*blinding-of-G* = *vimage2p the-G<sub>m</sub> the-G<sub>m</sub>* (*blinding-of-F* (*root-hash-T rha*) (*blinding-of-T rha boa*) *rhb bob*)  
⟨*proof*⟩

**end**

**lemma** *blinding-of-G-mono*:

**assumes**  $boa \leq boa'$   $bob \leq bob'$   
**shows** *blinding-of-G rha boa rhb bob*  $\leq$  *blinding-of-G rha boa' rhb bob'*  
⟨*proof*⟩

**lemma** *blinding-of-G-root-hash*:

**assumes**  $boa \leq \text{vimage2p rha rha}$  (=) **and**  $bob \leq \text{vimage2p rhb rhb}$  (=)  
**shows** *blinding-of-G rha boa rhb bob*  $\leq$  *vimage2p (root-hash-G rha rhb) (root-hash-G rha rhb)* (=)  
⟨*proof*⟩

**lemma** *blinding-of-on-G* [*locale-witness*]:

**assumes** *blinding-of-on A rha boa blinding-of-on B rhb bob*  
**shows** *blinding-of-on*  $\{x. \text{set1-}G_m x \subseteq A \wedge \text{set3-}G_m x \subseteq B\}$  (*root-hash-G rha rhb*) (*blinding-of-G rha boa rhb bob*)  
(is *blinding-of-on ?A ?h ?bo*)  
⟨*proof*⟩

**lemmas** *blinding-of-G* [*locale-witness*] = *blinding-of-on-G*[**where**  $A=UNIV$  **and**

$B=UNIV$ , *simplified*]

### 3.4 Merging

Two Merkle values with the same root hash can be merged into a less blinded Merkle value. The operation is unspecified for trees with different root hashes.

#### 3.4.1 Merging on the base functor

**axiomatization**  $merge-F :: ('a_m, 'a_h) hash \Rightarrow 'a_m merge \Rightarrow ('b_m, 'b_h) hash \Rightarrow 'b_m merge$

$\Rightarrow ('a_m, 'a_h, 'b_m, 'b_h) F_m merge$  **where**

$merge-F-cong$  [*fundef-cong*]:

$\llbracket \bigwedge a b. a \in set1-F_m x \implies ma a b = ma' a b; \bigwedge a b. a \in set3-F_m x \implies mb a b = mb' a b \rrbracket$

$\implies merge-F rha ma rhb mb x y = merge-F rha ma' rhb mb' x y$

**and**

$merge-on-F$  [*locale-witness*]:

$\llbracket merge-on A rha boa ma; merge-on B rhb bob mb \rrbracket$

$\implies merge-on \{x. set1-F_m x \subseteq A \wedge set3-F_m x \subseteq B\} (root-hash-F rha rhb) (blinding-of-F rha boa rhb bob) (merge-F rha ma rhb mb)$

**lemmas**  $merge-F$  [*locale-witness*] =  $merge-on-F$ [**where**  $A=UNIV$  **and**  $B=UNIV$ , *simplified*]

#### 3.4.2 Merging on the least fixpoint

**lemma**  $wfP-subterm-T: wfP (\lambda x y. x \in set3-F_m (the-T_m y))$

*<proof>*

**lemma**  $irrefl-subterm-T: x \in set3-F_m y \implies y \neq the-T_m x$

*<proof>*

**context**

**fixes**  $rh :: ('a_m, 'a_h) hash$

**fixes**  $m :: 'a_m merge$

**begin**

**function**  $merge-T :: ('a_m, 'a_h) T_m merge$  **where**

$merge-T (T_m x) (T_m y) = map-option T_m (merge-F rh m (root-hash-T rh) merge-T x y)$

*<proof>*

**termination**

*<proof>*

**lemma**  $merge-on-T$  [*locale-witness*]:

**assumes**  $merge-on A rh bo m$

**shows** *merge-on*  $\{x. \text{set1-}T_m x \subseteq A\}$  (*root-hash-T rh*) (*blinding-of-T rh bo*)  
*merge-T*  
 (**is** *merge-on* ?A ?h ?bo ?m)  
 ⟨*proof*⟩

**lemmas** *merge-T* [*locale-witness*] = *merge-on-T*[**where**  $A=UNIV$ , *simplified*]

**end**

**lemma** *merge-T-cong* [*fundef-cong*]:  
**assumes**  $\bigwedge a b. a \in \text{set1-}T_m x \implies m a b = m' a b$   
**shows** *merge-T rh*  $m x y = \text{merge-T rh } m' x y$   
 ⟨*proof*⟩

### 3.4.3 Merging and composition

**context**

**fixes** *rha* :: ('a<sub>m</sub>, 'a<sub>h</sub>) *hash*  
**fixes** *ma* :: 'a<sub>m</sub> *merge*  
**fixes** *rhb* :: ('b<sub>m</sub>, 'b<sub>h</sub>) *hash*  
**fixes** *mb* :: 'b<sub>m</sub> *merge*

**begin**

**primrec** *merge-G* :: ('a<sub>m</sub>, 'a<sub>h</sub>, 'b<sub>m</sub>, 'b<sub>h</sub>) *G<sub>m</sub> merge where*  
*merge-G* (*G<sub>m</sub> x*) *y'* = (*case y'* of *G<sub>m</sub> y*  $\Rightarrow$   
*map-option G<sub>m</sub> (merge-F (root-hash-T rha) (merge-T rha ma) rhb mb x y)*)

**lemma** *merge-G-simps* [*simp*]:  
*merge-G* (*G<sub>m</sub> x*) (*G<sub>m</sub> y*) = *map-option G<sub>m</sub> (merge-F (root-hash-T rha) (merge-T rha ma) rhb mb x y)*  
 ⟨*proof*⟩

**declare** *merge-G.simps* [*simp del*]

**lemma** *merge-on-G*:

**assumes** *a: merge-on A rha boa ma* **and** *b: merge-on B rhb bob mb*  
**shows** *merge-on*  $\{x. \text{set1-}G_m x \subseteq A \wedge \text{set3-}G_m x \subseteq B\}$  (*root-hash-G rha rhb*)  
 (*blinding-of-G rha boa rhb bob*) *merge-G*  
 (**is** *merge-on* ?A ?h ?bo ?m)  
 ⟨*proof*⟩

**lemmas** *merge-G* [*locale-witness*] = *merge-on-G*[**where**  $A=UNIV$  **and**  $B=UNIV$ , *simplified*]

**end**

**lemma** *merge-G-cong* [*fundef-cong*]:  
 [  $\bigwedge a b. a \in \text{set1-}G_m x \implies ma a b = ma' a b; \bigwedge a b. a \in \text{set3-}G_m x \implies mb a b = mb' a b$  ]

$\Rightarrow \text{merge-G rha ma rhb mb } x y = \text{merge-G rha ma' rhb mb' } x y$   
 ⟨proof⟩

**end**

**theory** *Inclusion-Proof-Construction* **imports**

*ADS-Construction*

**begin**

**primrec** *blind-blindable* :: ('a<sub>m</sub> ⇒ 'a<sub>h</sub>) ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>) *blindable*<sub>m</sub> ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>)  
*blindable*<sub>m</sub> **where**

*blind-blindable* *h* (*Blinded* *x*) = *Blinded* *x*

| *blind-blindable* *h* (*Unblinded* *x*) = *Blinded* (*Content* (*h* *x*))

**lemma** *hash-blind-blindable* [*simp*]: *hash-blindable* *h* (*blind-blindable* *h* *x*) = *hash-blindable*  
*h* *x*

⟨proof⟩

### 3.5 Inclusion proof construction for rose trees

#### 3.5.1 Hashing, embedding and blinding source trees

**context** *fixes* *h* :: 'a ⇒ 'a<sub>h</sub> **begin**

**fun** *hash-source-tree* :: 'a *rose-tree* ⇒ 'a<sub>h</sub> *rose-tree*<sub>h</sub> **where**

*hash-source-tree* (*Tree* (*data*, *subtrees*)) = *Tree*<sub>h</sub> (*Content* (*h* *data*, *map* *hash-source-tree*  
*subtrees*))

**end**

**context** *fixes* *e* :: 'a ⇒ 'a<sub>m</sub> **begin**

**fun** *embed-source-tree* :: 'a *rose-tree* ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>) *rose-tree*<sub>m</sub> **where**

*embed-source-tree* (*Tree* (*data*, *subtrees*)) =

*Tree*<sub>m</sub> (*Unblinded* (*e* *data*, *map* *embed-source-tree* *subtrees*))

**end**

**context** *fixes* *h* :: 'a ⇒ 'a<sub>h</sub> **begin**

**fun** *blind-source-tree* :: 'a *rose-tree* ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>) *rose-tree*<sub>m</sub> **where**

*blind-source-tree* (*Tree* (*data*, *subtrees*)) = *Tree*<sub>m</sub> (*Blinded* (*Content* (*h* *data*, *map*  
*(hash-source-tree* *h*) *subtrees*)))

**end**

**case-of-simps** *blind-source-tree-cases*: *blind-source-tree.simps*

**fun** *is-blinded* :: ('a<sub>m</sub>, 'a<sub>h</sub>) *rose-tree*<sub>m</sub> ⇒ *bool* **where**

*is-blinded* (*Tree*<sub>m</sub> (*Blinded* -)) = *True*

| *is-blinded* - = *False*

**lemma** *hash-blinded-simp*: *hash-tree* *h'* (*blind-source-tree* *h* *st*) = *hash-source-tree*  
*h* *st*

⟨proof⟩

**lemma** *hash-embedded-simp*:

*hash-tree*  $h$  (*embed-source-tree*  $e$   $st$ ) = *hash-source-tree* ( $h \circ e$ )  $st$   
<proof>

**lemma** *blinded-embedded-same-hash*:

*hash-tree*  $h''$  (*blind-source-tree* ( $h \circ e$ )  $st$ ) = *hash-tree*  $h$  (*embed-source-tree*  $e$   $st$ )  
<proof>

**lemma** *blinding-blinds* [*simp*]:

*is-blinded* (*blind-source-tree*  $h$   $t$ )  
<proof>

**lemma** *blinded-blinds-embedded*:

*blinding-of-tree*  $h$   $bo$  (*blind-source-tree* ( $h \circ e$ )  $st$ ) (*embed-source-tree*  $e$   $st$ )  
<proof>

**fun** *embed-hash-tree* ::  $'a$  *rose-tree* <sub>$h$</sub>   $\Rightarrow$  ( $'a$ ,  $'a$ ) *rose-tree* <sub>$m$</sub>  **where**

*embed-hash-tree* (*Tree* <sub>$h$</sub>   $h$ ) = *Tree* <sub>$m$</sub>  (*Blinded*  $h$ )

### 3.5.2 Auxiliary definitions: selectors and list splits

**fun** *children* ::  $'a$  *rose-tree*  $\Rightarrow$   $'a$  *rose-tree list* **where**

*children* (*Tree* ( $data$ ,  $subtrees$ )) =  $subtrees$

**fun** *children* <sub>$m$</sub>  :: ( $'a$ ,  $'a_h$ ) *rose-tree* <sub>$m$</sub>   $\Rightarrow$  ( $'a$ ,  $'a_h$ ) *rose-tree* <sub>$m$</sub>  *list* **where**

*children* <sub>$m$</sub>  (*Tree* <sub>$m$</sub>  (*Unblinded* ( $data$ ,  $subtrees$ ))) =  $subtrees$

| *children* <sub>$m$</sub>  - = *undefined*

**fun** *splits* ::  $'a$  *list*  $\Rightarrow$  ( $'a$  *list*  $\times$   $'a$   $\times$   $'a$  *list*) *list* **where**

*splits* [] = []

| *splits* ( $x\#xs$ ) = ([],  $x$ ,  $xs$ ) # *map* ( $\lambda(l, y, r). (x \# l, y, r)$ ) (*splits*  $xs$ )

**lemma** *splits-iff*: ( $l, a, r$ )  $\in$  *set* (*splits*  $ll$ ) = ( $ll = l @ a \# r$ )

<proof>

### 3.5.3 Zippers

Zippers provide a neat representation of tree-like ADSs when they have only a single unblinded subtree. The zipper path provides the "inclusion proof" that the unblinded subtree is included in a larger structure.

**type-synonym**  $'a$  *path-elem* =  $'a \times 'a$  *rose-tree list*  $\times$   $'a$  *rose-tree list*

**type-synonym**  $'a$  *path* =  $'a$  *path-elem list*

**type-synonym**  $'a$  *zipper* =  $'a$  *path*  $\times$   $'a$  *rose-tree*

**definition** *zipper-of-tree* ::  $'a$  *rose-tree*  $\Rightarrow$   $'a$  *zipper* **where**

*zipper-of-tree*  $t \equiv$  ([],  $t$ )

**fun** *tree-of-zipper* ::  $'a$  *zipper*  $\Rightarrow$   $'a$  *rose-tree* **where**

*tree-of-zipper* ( $[], t$ ) =  $t$   
| *tree-of-zipper* ( $(a, l, r) \# z, t$ ) = *tree-of-zipper* ( $z, (Tree\ a, (l\ @\ t\ \#\ r))$ )

**case-of-simps** *tree-of-zipper-cases*: *tree-of-zipper.simps*

**lemma** *tree-of-zipper-id*[*iff*]: *tree-of-zipper* (*zipper-of-tree*  $t$ ) =  $t$   
⟨*proof*⟩

**fun** *zipper-children* ::  $'a\ zipper \Rightarrow 'a\ zipper\ list$  **where**  
*zipper-children* ( $p, Tree\ (a, ts)$ ) = *map* ( $\lambda(l, t, r). ((a, l, r) \# p, t)$ ) (*splits*  $ts$ )

**lemma** *zipper-children-same-tree*:  
**assumes**  $z' \in set\ (zipper-children\ z)$   
**shows** *tree-of-zipper*  $z' = tree-of-zipper\ z$   
⟨*proof*⟩

**type-synonym**  $(a_m, a_h)\ path\_elem_m = a_m \times (a_m, a_h)\ rose\_tree_m\ list \times (a_m, a_h)\ rose\_tree_m\ list$

**type-synonym**  $(a_m, a_h)\ path_m = (a_m, a_h)\ path\_elem_m\ list$

**type-synonym**  $(a_m, a_h)\ zipper_m = (a_m, a_h)\ path_m \times (a_m, a_h)\ rose\_tree_m$

**definition** *zipper-of-tree<sub>m</sub>* ::  $(a_m, a_h)\ rose\_tree_m \Rightarrow (a_m, a_h)\ zipper_m$  **where**  
*zipper-of-tree<sub>m</sub>*  $t \equiv ([] , t)$

**fun** *tree-of-zipper<sub>m</sub>* ::  $(a_m, a_h)\ zipper_m \Rightarrow (a_m, a_h)\ rose\_tree_m$  **where**  
*tree-of-zipper<sub>m</sub>* ( $[], t$ ) =  $t$   
| *tree-of-zipper<sub>m</sub>* ( $(m, l, r) \# z, t$ ) = *tree-of-zipper<sub>m</sub>* ( $z, Tree_m\ (Unblinded\ (m, l\ @\ t\ \#\ r))$ )

**lemma** *tree-of-zipper<sub>m</sub>-append*:  
*tree-of-zipper<sub>m</sub>* ( $p\ @\ p', t$ ) = *tree-of-zipper<sub>m</sub>* ( $p', tree-of-zipper_m\ (p, t)$ )  
⟨*proof*⟩

**fun** *zipper-children<sub>m</sub>* ::  $(a_m, a_h)\ zipper_m \Rightarrow (a_m, a_h)\ zipper_m\ list$  **where**  
*zipper-children<sub>m</sub>* ( $p, Tree_m\ (Unblinded\ (a, ts))$ ) = *map* ( $\lambda(l, t, r). ((a, l, r) \# p, t)$ ) (*splits*  $ts$ )  
| *zipper-children<sub>m</sub>* - =  $[]$

**lemma** *zipper-children-same-tree<sub>m</sub>*:  
**assumes**  $z' \in set\ (zipper-children_m\ z)$   
**shows** *tree-of-zipper<sub>m</sub>*  $z' = tree-of-zipper_m\ z$   
⟨*proof*⟩

**fun** *blind-path-elem* ::  $(a \Rightarrow a_m) \Rightarrow (a_m \Rightarrow a_h) \Rightarrow 'a\ path\_elem \Rightarrow (a_m, a_h)\ path\_elem_m$  **where**  
*blind-path-elem*  $e\ h\ (x, l, r)$  =  $(e\ x, map\ (blind\_source\_tree\ (h\ \circ\ e))\ l, map\ (blind\_source\_tree\ (h\ \circ\ e))\ r)$

**case-of-simps** *blind-path-elem-cases*: *blind-path-elem.simps*

**definition** *blind-path* :: ('a ⇒ 'a<sub>m</sub>) ⇒ ('a<sub>m</sub> ⇒ 'a<sub>h</sub>) ⇒ 'a path ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>) path<sub>m</sub>  
**where**

*blind-path e h* ≡ map (*blind-path-elem e h*)

**fun** *embed-path-elem* :: ('a ⇒ 'a<sub>m</sub>) ⇒ 'a path-elem ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>) path-elem<sub>m</sub> **where**  
*embed-path-elem e (d, l, r)* = (e d, map (*embed-source-tree e*) l, map (*embed-source-tree e*) r)

**definition** *embed-path* :: ('a ⇒ 'a<sub>m</sub>) ⇒ 'a path ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>) path<sub>m</sub> **where**  
*embed-path embed-elem* ≡ map (*embed-path-elem embed-elem*)

**lemma** *hash-tree-of-zipper-same-path*:

*hash-tree h (tree-of-zipper<sub>m</sub> (p, v))* = *hash-tree h (tree-of-zipper<sub>m</sub> (p, v'))*  
 $\longleftrightarrow$  *hash-tree h v* = *hash-tree h v'*  
 ⟨proof⟩

**fun** *hash-path-elem* :: ('a<sub>m</sub> ⇒ 'a<sub>h</sub>) ⇒ ('a<sub>m</sub>, 'a<sub>h</sub>) path-elem<sub>m</sub> ⇒ ('a<sub>h</sub> × 'a<sub>h</sub> rose-tree<sub>h</sub> list × 'a<sub>h</sub> rose-tree<sub>h</sub> list) **where**  
*hash-path-elem h (e, l, r)* = (h e, map (*hash-tree h*) l, map (*hash-tree h*) r)

**lemma** *hash-view-zipper-eqI*:

[[ *hash-list (hash-path-elem h) p* = *hash-list (hash-path-elem h') p'*;  
*hash-tree h v* = *hash-tree h' v'* ]] ⇒  
*hash-tree h (tree-of-zipper<sub>m</sub> (p, v))* = *hash-tree h' (tree-of-zipper<sub>m</sub> (p', v'))*  
 ⟨proof⟩

**lemma** *blind-embed-path-same-hash*:

*hash-tree h (tree-of-zipper<sub>m</sub> (blind-path e h p, t))* = *hash-tree h (tree-of-zipper<sub>m</sub> (embed-path e p, t))*  
 ⟨proof⟩

**lemma** *tree-of-embed-commute*:

*tree-of-zipper<sub>m</sub> (embed-path e p, embed-source-tree e t)* = *embed-source-tree e (tree-of-zipper (p, t))*  
 ⟨proof⟩

**lemma** *childz-same-tree*:

(l, t, r) ∈ set (*splits ts*) ⇒  
*tree-of-zipper<sub>m</sub> (embed-path e p, embed-source-tree e (Tree (d, ts)))*  
 = *tree-of-zipper<sub>m</sub> (embed-path e ((d, l, r) # p), embed-source-tree e t)*  
 ⟨proof⟩

**lemma** *blinding-of-same-path*:

**assumes** bo: *blinding-of-on UNIV h bo*

**shows**

*blinding-of-tree h bo (tree-of-zipper<sub>m</sub> (p, t)) (tree-of-zipper<sub>m</sub> (p, t'))*  
 $\longleftrightarrow$  *blinding-of-tree h bo t t'*

⟨proof⟩

**lemma** *zipper-children-size-change* [*termination-simp*]:  $(a, b) \in \text{set } (\text{zipper-children } (p, v)) \implies \text{size } b < \text{size } v$   
 ⟨*proof*⟩

### 3.6 All zippers of a rose tree

**context** *fixes*  $e :: 'a \Rightarrow 'a_m$  and  $h :: 'a_m \Rightarrow 'a_h$  **begin**

**fun** *zippers-rose-tree* ::  $'a \text{ zipper} \Rightarrow ('a_m, 'a_h) \text{ zipper}_m \text{ list}$  **where**  
*zippers-rose-tree*  $(p, t) = (\text{blind-path } e \ h \ p, \text{embed-source-tree } e \ t) \#$   
*concat* (*map* *zippers-rose-tree* (*zipper-children*  $(p, t)$ ))

**end**

**lemmas** [*simp del*] = *zippers-rose-tree.simps zipper-children.simps*

**lemma** *zippers-rose-tree-same-hash'*:  
**assumes**  $z \in \text{set } (\text{zippers-rose-tree } e \ h \ (p, t))$   
**shows**  $\text{hash-tree } h \ (\text{tree-of-zipper}_m \ z) =$   
 $\text{hash-tree } h \ (\text{tree-of-zipper}_m \ (\text{embed-path } e \ p, \text{embed-source-tree } e \ t))$   
 ⟨*proof*⟩

**lemma** *zippers-rose-tree-blinding-of*:  
**assumes** *blinding-of-on UNIV*  $h \ bo$   
**and**  $z \in \text{set } (\text{zippers-rose-tree } e \ h \ (p, t))$   
**shows** *blinding-of-tree*  $h \ bo \ (\text{tree-of-zipper}_m \ z) \ (\text{tree-of-zipper}_m \ (\text{blind-path } e \ h \ p,$   
*embed-source-tree*  $e \ t))$   
 ⟨*proof*⟩

**lemma** *zippers-rose-tree-neq-Nil*:  $\text{zippers-rose-tree } e \ h \ (p, t) \neq []$   
 ⟨*proof*⟩

**lemma** (*in comp-fun-idem*) *fold-set-union*:  
**assumes** *finite*  $A$  *finite*  $B$   
**shows**  $\text{Finite-Set.fold } f \ z \ (A \cup B) = \text{Finite-Set.fold } f \ (\text{Finite-Set.fold } f \ z \ A) \ B$   
 ⟨*proof*⟩

**context** *merkle-interface* **begin**

**lemma** *comp-fun-idem-merge*:  $\text{comp-fun-idem } (\lambda x \ yo. \ yo \gg\! = \ m \ x)$   
 ⟨*proof*⟩

**interpretation** *merge*:  $\text{comp-fun-idem } \lambda x \ yo. \ yo \gg\! = \ m \ x$  ⟨*proof*⟩

**definition** *Merge* ::  $'a_m \text{ set} \Rightarrow 'a_m \text{ option}$  **where**  
*Merge*  $A = (\text{if } A = \{\} \vee \text{infinite } A \text{ then } \text{None} \text{ else } \text{Finite-Set.fold } (\lambda x \ yo. \ yo \gg\! = \ m \ x) \ (\text{Some } (\text{SOME } x. \ x \in A)) \ A)$

**lemma** *Merge-empty* [simp]:  $\text{Merge } \{\} = \text{None}$   
<proof>

**lemma** *Merge-infinite* [simp]:  $\text{infinite } A \implies \text{Merge } A = \text{None}$   
<proof>

**lemma** *Merge-cong-start*:  
 $\text{Finite-Set.fold } (\lambda x \ yo. \ yo \gg m \ x) \ (\text{Some } x) \ A = \text{Finite-Set.fold } (\lambda x \ yo. \ yo \gg m \ x) \ (\text{Some } y) \ A$  **(is ?lhs = ?rhs)**  
**if**  $x \in A \ y \in A \ \text{finite } A$   
<proof>

**lemma** *Merge-insert* [simp]:  $\text{Merge } (\text{insert } x \ A) = (\text{if } A = \{\} \ \text{then } \text{Some } x \ \text{else } \text{Merge } A \gg m \ x)$  **(is ?lhs = ?rhs)**  
<proof>

**lemma** *Merge-insert-alt*:  
 $\text{Merge } (\text{insert } x \ A) = \text{Finite-Set.fold } (\lambda x \ yo. \ yo \gg m \ x) \ (\text{Some } x) \ A$  **(is ?lhs = ?rhs)** **if**  $\text{finite } A$   
<proof>

**lemma** *Merge-None* [simp]:  $\text{Finite-Set.fold } (\lambda x \ yo. \ yo \gg m \ x) \ \text{None} \ A = \text{None}$   
<proof>

**lemma** *Merge-union*:  
 $\text{Merge } (A \cup B) = (\text{if } A = \{\} \ \text{then } \text{Merge } B \ \text{else if } B = \{\} \ \text{then } \text{Merge } A \ \text{else } (\text{Merge } A \gg (\lambda a. \ \text{Merge } B \gg m \ a)))$   
**(is ?lhs = ?rhs)**  
<proof>

**lemma** *Merge-upper*:  
**assumes**  $m: \text{Merge } A = \text{Some } x$  **and**  $y: y \in A$   
**shows**  $bo \ y \ x$   
<proof>

**lemma** *Merge-least*:  
**assumes**  $m: \text{Merge } A = \text{Some } x$  **and**  $u[\text{rule-format}]: \forall a \in A. \ bo \ a \ u$   
**shows**  $bo \ x \ u$   
<proof>

**lemma** *Merge-defined*:  
**assumes**  $\text{finite } A \ A \neq \{\} \ \forall a \in A. \ \forall b \in A. \ h \ a = h \ b$   
**shows**  $\text{Merge } A \neq \text{None}$   
<proof>

**lemma** *Merge-hash*:  
**assumes**  $\text{Merge } A = \text{Some } x$   $a \in A$   
**shows**  $h \ a = h \ x$   
<proof>

**end**

**end**

**theory** *Canton-Transaction-Tree* **imports**

*Inclusion-Proof-Construction*

**begin**

## 4 Canton's hierarchical transaction trees

**typedecl** *view-data*

**typedecl** *view-metadata*

**typedecl** *common-metadata*

**typedecl** *participant-metadata*

**datatype** *view* = *View view-metadata view-data (subviews: view list)*

**datatype** *transaction* = *Transaction common-metadata participant-metadata (views: view list)*

### 4.1 Views as authenticated data structures

**type-synonym** *view-metadata<sub>h</sub>* = *view-metadata blindable<sub>h</sub>*

**type-synonym** *view-data<sub>h</sub>* = *view-data blindable<sub>h</sub>*

**datatype** *view<sub>h</sub>* = *View<sub>h</sub> ((view-metadata<sub>h</sub> ×<sub>h</sub> view-data<sub>h</sub>) ×<sub>h</sub> view<sub>h</sub> list<sub>h</sub>) blindable<sub>h</sub>*

**type-synonym** *view-metadata<sub>m</sub>* = *(view-metadata, view-metadata) blindable<sub>m</sub>*

**type-synonym** *view-data<sub>m</sub>* = *(view-data, view-data) blindable<sub>m</sub>*

**datatype** *view<sub>m</sub>* = *View<sub>m</sub>*

*((view-metadata<sub>m</sub> ×<sub>m</sub> view-data<sub>m</sub>) ×<sub>m</sub> view<sub>m</sub> list<sub>m</sub>,  
(view-metadata<sub>h</sub> ×<sub>h</sub> view-data<sub>h</sub>) ×<sub>h</sub> view<sub>h</sub> list<sub>h</sub>) blindable<sub>m</sub>*

**abbreviation** (*input*) *hash-view-data* :: *(view-data<sub>m</sub>, view-data<sub>h</sub>) hash where  
hash-view-data ≡ hash-blindable id*

**abbreviation** (*input*) *blinding-of-view-data* :: *view-data<sub>m</sub> blinding-of where  
blinding-of-view-data ≡ blinding-of-blindable id (=)*

**abbreviation** (*input*) *merge-view-data* :: *view-data<sub>m</sub> 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  
<proof>*

**abbreviation** (*input*) *hash-view-metadata* :: *(view-metadata<sub>m</sub>, view-metadata<sub>h</sub>)  
hash where*

*hash-view-metadata ≡ hash-blindable id*

**abbreviation** (*input*) *blinding-of-view-metadata* :: *view-metadata<sub>m</sub>* *blinding-of* **where**  
*blinding-of-view-metadata*  $\equiv$  *blinding-of-blindable id* (=)

**abbreviation** (*input*) *merge-view-metadata* :: *view-metadata<sub>m</sub>* *merge* **where**  
*merge-view-metadata*  $\equiv$  *merge-blindable id merge-discrete*

**lemma** *merkle-view-metadata*:

*merkle-interface hash-view-metadata blinding-of-view-metadata merge-view-metadata*  
*<proof>*

**type-synonym** *view-content* = *view-metadata*  $\times$  *view-data*

**type-synonym** *view-content<sub>h</sub>* = *view-metadata<sub>h</sub>*  $\times_h$  *view-data<sub>h</sub>*

**type-synonym** *view-content<sub>m</sub>* = *view-metadata<sub>m</sub>*  $\times_m$  *view-data<sub>m</sub>*

**locale** *view-merkle* **begin**

**type-synonym** *view<sub>h</sub>'* = *view-content<sub>h</sub>* *rose-tree<sub>h</sub>*

**primrec** *from-view<sub>h</sub>* :: *view<sub>h</sub>*  $\Rightarrow$  *view<sub>h</sub>'* **where**

*from-view<sub>h</sub>* (*View<sub>h</sub>* *x*) = *Tree<sub>h</sub>* (*map-blindable<sub>h</sub>* (*map-prod id* (*map from-view<sub>h</sub>*))  
*x*)

**primrec** *to-view<sub>h</sub>* :: *view<sub>h</sub>'*  $\Rightarrow$  *view<sub>h</sub>* **where**

*to-view<sub>h</sub>* (*Tree<sub>h</sub>* *x*) = *View<sub>h</sub>* (*map-blindable<sub>h</sub>* (*map-prod id* (*map to-view<sub>h</sub>*)) *x*)

**lemma** *from-to-view<sub>h</sub>* [*simp*]: *from-view<sub>h</sub>* (*to-view<sub>h</sub>* *x*) = *x*  
*<proof>*

**lemma** *to-from-view<sub>h</sub>* [*simp*]: *to-view<sub>h</sub>* (*from-view<sub>h</sub>* *x*) = *x*  
*<proof>*

**lemma** *iso-view<sub>h</sub>*: *type-definition from-view<sub>h</sub> to-view<sub>h</sub> UNIV*  
*<proof>*

**setup-lifting** *iso-view<sub>h</sub>*

**lemma** *cr-view<sub>h</sub>-Grp*: *cr-view<sub>h</sub>* = *Grp UNIV to-view<sub>h</sub>*  
*<proof>*

**lemma** *View<sub>h</sub>-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**

(*rel-blindable<sub>h</sub>* (*rel-prod* (=) (*list-all2 pcr-view<sub>h</sub>*))  $\implies$  *pcr-view<sub>h</sub>*) *Tree<sub>h</sub> View<sub>h</sub>*  
*<proof>*

**type-synonym** *view<sub>m</sub>'* = (*view-content<sub>m</sub>*, *view-content<sub>h</sub>*) *rose-tree<sub>m</sub>*

**primrec** *from-view<sub>m</sub>* :: *view<sub>m</sub>*  $\Rightarrow$  *view<sub>m</sub>'* **where**

*from-view<sub>m</sub>* (*View<sub>m</sub>* *x*) = *Tree<sub>m</sub>* (*map-blindable<sub>m</sub>* (*map-prod id* (*map from-view<sub>m</sub>*))  
(*map-prod id* (*map from-view<sub>h</sub>*)) *x*)

**primrec** *to-view<sub>m</sub>* :: *view<sub>m</sub>'*  $\Rightarrow$  *view<sub>m</sub>* **where**

$to-view_m (Tree_m x) = View_m (map-blindable_m (map-prod id (map to-view_m)) (map-prod id (map to-view_h)) x)$

**lemma** *from-to-view<sub>m</sub>* [simp]: *from-view<sub>m</sub>* (to-view<sub>m</sub> x) = x  
⟨proof⟩

**lemma** *to-from-view<sub>m</sub>* [simp]: to-view<sub>m</sub> (from-view<sub>m</sub> x) = x  
⟨proof⟩

**lemma** *iso-view<sub>m</sub>*: type-definition from-view<sub>m</sub> to-view<sub>m</sub> UNIV  
⟨proof⟩

**setup-lifting** *iso-view<sub>m</sub>*

**lemma** *cr-view<sub>m</sub>-Grp*: cr-view<sub>m</sub> = Grp UNIV to-view<sub>m</sub>  
⟨proof⟩

**lemma** *View<sub>m</sub>-transfer* [transfer-rule]: **includes** *lifting-syntax* **shows**  
(rel-blindable<sub>m</sub> (rel-prod (=) (list-all2 pcr-view<sub>m</sub>)) (rel-prod (=) (list-all2 pcr-view<sub>h</sub>)))  
====> pcr-view<sub>m</sub>) Tree<sub>m</sub> View<sub>m</sub>  
⟨proof⟩

**end**

**code-datatype** View<sub>h</sub>  
**code-datatype** View<sub>m</sub>

**context begin**  
**interpretation** *view-merkle* ⟨proof⟩

**abbreviation** (input) *hash-view-content* :: (view-content<sub>m</sub>, view-content<sub>h</sub>) hash  
**where**  
*hash-view-content* ≡ hash-prod hash-view-metadata hash-view-data

**abbreviation** (input) *blinding-of-view-content* :: view-content<sub>m</sub> blinding-of **where**  
*blinding-of-view-content* ≡ blinding-of-prod blinding-of-view-metadata blinding-of-view-data

**abbreviation** (input) *merge-view-content* :: view-content<sub>m</sub> merge **where**  
*merge-view-content* ≡ merge-prod merge-view-metadata merge-view-data

**lift-definition** *hash-view* :: (view<sub>m</sub>, view<sub>h</sub>) hash **is**  
*hash-tree hash-view-content* ⟨proof⟩

**lift-definition** *blinding-of-view* :: view<sub>m</sub> blinding-of **is**  
*blinding-of-tree hash-view-content blinding-of-view-content* ⟨proof⟩

**lift-definition** *merge-view* :: view<sub>m</sub> merge **is**  
*merge-tree hash-view-content merge-view-content* ⟨proof⟩

**lemma** *merkle-view* [*locale-witness*]: *merkle-interface hash-view blinding-of-view merge-view*  
 ⟨*proof*⟩

**lemma** *hash-view-simps* [*simp*]:  
 $hash\_view (View_m x) =$   
 $View_h (hash\_blindable (hash\_prod hash\_view\_content (hash\_list hash\_view))) x$   
 ⟨*proof*⟩

**lemma** *blinding-of-view-iff* [*simp*]:  
 $blinding\_of\_view (View_m x) (View_m y) \longleftrightarrow$   
 $blinding\_of\_blindable (hash\_prod hash\_view\_content (hash\_list hash\_view))$   
 $(blinding\_of\_prod blinding\_of\_view\_content (blinding\_of\_list blinding\_of\_view)) x$   
 $y$   
 ⟨*proof*⟩

**lemma** *blinding-of-view-induct* [*consumes 1, induct pred: blinding-of-view*]:  
**assumes** *blinding-of-view*  $x y$   
**and**  $\bigwedge x y. blinding\_of\_blindable (hash\_prod hash\_view\_content (hash\_list hash\_view))$   
 $(blinding\_of\_prod blinding\_of\_view\_content (blinding\_of\_list (\lambda x y. blind-$   
*ing-of-view*  $x y \wedge P x y))) x y$   
 $\implies P (View_m x) (View_m y)$   
**shows**  $P x y$   
 ⟨*proof*⟩

**lemma** *merge-view-simps* [*simp*]:  
 $merge\_view (View_m x) (View_m y) =$   
 $map\_option View_m (merge\_rt-F_m hash\_view\_content merge\_view\_content hash\_view$   
 $merge\_view x y)$   
 ⟨*proof*⟩

**end**

## 4.2 Transaction trees as authenticated data structures

**type-synonym**  $common\_metadata_h = common\_metadata blindable_h$

**type-synonym**  $common\_metadata_m = (common\_metadata, common\_metadata) blind-$   
 $able_m$

**type-synonym**  $participant\_metadata_h = participant\_metadata blindable_h$

**type-synonym**  $participant\_metadata_m = (participant\_metadata, participant\_metadata)$   
 $blindable_m$

**datatype**  $transaction_h = Transaction_h$

(*the-Transaction<sub>h</sub>*:  $((common\_metadata_h \times_h participant\_metadata_h) \times_h view_h$   
 $list_h) blindable_h$ )

**datatype**  $transaction_m = Transaction_m$

(*the-Transaction<sub>m</sub>*:  $((common\_metadata_m \times_m participant\_metadata_m) \times_m view_m$

*list<sub>m</sub>,*

*(common-metadata<sub>h</sub> ×<sub>h</sub> participant-metadata<sub>h</sub>) ×<sub>h</sub> view<sub>h</sub> list<sub>h</sub>) blindable<sub>m</sub>)*

**abbreviation** (*input*) *hash-common-metadata* :: *(common-metadata<sub>m</sub>, common-metadata<sub>h</sub>)*  
*hash where*

*hash-common-metadata* ≡ *hash-blindable id*

**abbreviation** (*input*) *blinding-of-common-metadata* :: *common-metadata<sub>m</sub> blind-*  
*ing-of where*

*blinding-of-common-metadata* ≡ *blinding-of-blindable id (=)*

**abbreviation** (*input*) *merge-common-metadata* :: *common-metadata<sub>m</sub> merge where*  
*merge-common-metadata* ≡ *merge-blindable id merge-discrete*

**abbreviation** (*input*) *hash-participant-metadata* :: *(participant-metadata<sub>m</sub>, partic-*  
*ipant-metadata<sub>h</sub>) hash where*

*hash-participant-metadata* ≡ *hash-blindable id*

**abbreviation** (*input*) *blinding-of-participant-metadata* :: *participant-metadata<sub>m</sub> blind-*  
*ing-of where*

*blinding-of-participant-metadata* ≡ *blinding-of-blindable id (=)*

**abbreviation** (*input*) *merge-participant-metadata* :: *participant-metadata<sub>m</sub> merge*  
*where*

*merge-participant-metadata* ≡ *merge-blindable id merge-discrete*

**locale** *transaction-merkle begin*

**lemma** *iso-transaction<sub>h</sub>: type-definition the-Transaction<sub>h</sub> Transaction<sub>h</sub> UNIV*  
*<proof>*

**setup-lifting** *iso-transaction<sub>h</sub>*

**lemma** *Transaction<sub>h</sub>-transfer [transfer-rule]: includes lifting-syntax shows*  
*((=) == => pcr-transaction<sub>h</sub>) id Transaction<sub>h</sub>*  
*<proof>*

**lemma** *iso-transaction<sub>m</sub>: type-definition the-Transaction<sub>m</sub> Transaction<sub>m</sub> UNIV*  
*<proof>*

**setup-lifting** *iso-transaction<sub>m</sub>*

**lemma** *Transaction<sub>m</sub>-transfer [transfer-rule]: includes lifting-syntax shows*  
*((=) == => pcr-transaction<sub>m</sub>) id Transaction<sub>m</sub>*  
*<proof>*

**end**

**code-datatype** *Transaction<sub>h</sub>*

**code-datatype** *Transaction<sub>m</sub>*

**context begin**

**interpretation** *transaction-merkle <proof>*

**lift-definition** *hash-transaction* :: (*transaction<sub>m</sub>*, *transaction<sub>h</sub>*) *hash* **is**  
*hash-blindable* (*hash-prod* (*hash-prod* *hash-common-metadata* *hash-participant-metadata*)  
(*hash-list* *hash-view*)) *<proof>*

**lift-definition** *blinding-of-transaction* :: *transaction<sub>m</sub>* *blinding-of* **is**  
*blinding-of-blindable*  
(*hash-prod* (*hash-prod* *hash-common-metadata* *hash-participant-metadata*) (*hash-list*  
*hash-view*))  
(*blinding-of-prod* (*blinding-of-prod* *blinding-of-common-metadata* *blinding-of-participant-metadata*)  
(*blinding-of-list* *blinding-of-view*)) *<proof>*

**lift-definition** *merge-transaction* :: *transaction<sub>m</sub>* *merge* **is**  
*merge-blindable*  
(*hash-prod* (*hash-prod* *hash-common-metadata* *hash-participant-metadata*) (*hash-list*  
*hash-view*))  
(*merge-prod* (*merge-prod* *merge-common-metadata* *merge-participant-metadata*)  
(*merge-list* *merge-view*)) *<proof>*

**lemma** *merkle-transaction* [*locale-witness*]:  
*merkle-interface* *hash-transaction* *blinding-of-transaction* *merge-transaction*  
*<proof>*

**lemmas** *hash-transaction-simps* [*simp*] = *hash-transaction.abs-eq*

**lemmas** *blinding-of-transaction-iff* [*simp*] = *blinding-of-transaction.abs-eq*

**lemmas** *merge-transaction-simps* [*simp*] = *merge-transaction.abs-eq*

**end**

**interpretation** *transaction*:  
*merkle-interface* *hash-transaction* *blinding-of-transaction* *merge-transaction*  
*<proof>*

### 4.3 Constructing authenticated data structures for views

**context** *view-merkle* **begin**

**type-synonym** *view'* = (*view-metadata* × *view-data*) *rose-tree*

**primrec** *from-view* :: *view* ⇒ *view'* **where**  
*from-view* (*View* *vm* *vd* *vs*) = *Tree* ((*vm*, *vd*), *map* *from-view* *vs*)

**primrec** *to-view* :: *view'* ⇒ *view* **where**  
*to-view* (*Tree* *x*) = *View* (*fst* (*fst* *x*)) (*snd* (*fst* *x*)) (*snd* (*map-prod* *id* (*map* *to-view*)  
*x*))

**lemma** *from-to-view* [*simp*]: *from-view* (*to-view* *x*) = *x*  
*<proof>*

**lemma** *to-from-view* [*simp*]: *to-view* (*from-view*  $x$ ) =  $x$   
⟨*proof*⟩

**lemma** *iso-view*: *type-definition* *from-view* *to-view* *UNIV*  
⟨*proof*⟩

**setup-lifting** *iso-view*

**definition** *View'* :: (*view-metadata* × *view-data*) × *view list* ⇒ *view* **where**  
*View'* = (λ((*vm*, *vd*), *vs*). *View* *vm* *vd* *vs*)

**lemma** *View-View'*: *View* = (λ*vm* *vd* *vs*. *View'* ((*vm*, *vd*), *vs*))  
⟨*proof*⟩

**lemma** *cr-view-Grp*: *cr-view* = *Grp* *UNIV* *to-view*  
⟨*proof*⟩

**lemma** *View'-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**  
(*rel-prod* (=) (*list-all2* *pcr-view*) ==> *pcr-view*) *Tree* *View'*  
⟨*proof*⟩

**end**

**code-datatype** *View*

**context** **begin**

**interpretation** *view-merkle* ⟨*proof*⟩

**abbreviation** *embed-view-content* :: *view-metadata* × *view-data* ⇒ *view-metadata<sub>m</sub>*  
× *view-data<sub>m</sub>* **where**  
*embed-view-content* ≡ *map-prod* *Unblinded* *Unblinded*

**lift-definition** *embed-view* :: *view* ⇒ *view<sub>m</sub>* **is** *embed-source-tree* *embed-view-content*  
⟨*proof*⟩

**lemma** *embed-view-simps* [*simp*]:  
*embed-view* (*View* *vm* *vd* *vs*) = *View<sub>m</sub>* (*Unblinded* ((*Unblinded* *vm*, *Unblinded*  
*vd*), *map* *embed-view* *vs*))  
⟨*proof*⟩

**end**

**context** *transaction-merkle* **begin**

**primrec** *the-Transaction* :: *transaction* ⇒ (*common-metadata* × *participant-metadata*)  
× *view list* **where**  
*the-Transaction* (*Transaction* *cm* *pm* *views*) = ((*cm*, *pm*), *views*) **for** *views*

**definition** *Transaction'* :: (*common-metadata* × *participant-metadata*) × *view list*

$\Rightarrow$  *transaction* **where**

$Transaction' = (\lambda((cm, pm), views). Transaction\ cm\ pm\ views)$

**lemma** *Transaction-Transaction'*:  $Transaction = (\lambda cm\ pm\ views. Transaction'\ ((cm, pm), views))$   
 $\langle proof \rangle$

**lemma** *the-Transaction-inverse* [simp]:  $Transaction'\ (the-Transaction\ x) = x$   
 $\langle proof \rangle$

**lemma** *Transaction'-inverse* [simp]:  $the-Transaction\ (Transaction'\ x) = x$   
 $\langle proof \rangle$

**lemma** *iso-transaction*: *type-definition the-Transaction Transaction' UNIV*  
 $\langle proof \rangle$

**setup-lifting** *iso-transaction*

**lemma** *Transaction'-transfer* [transfer-rule]: **includes** *lifting-syntax* **shows**  
 $((=) == => pcr-transaction)\ id\ Transaction'$   
 $\langle proof \rangle$

**end**

**code-datatype** *Transaction*

**context begin**

**interpretation** *transaction-merkle*  $\langle proof \rangle$

**lift-definition** *embed-transaction* ::  $transaction \Rightarrow transaction_m$  **is**  
 $Unblinded \circ map-prod\ (map-prod\ Unblinded\ Unblinded)\ (map\ embed-view)\ \langle proof \rangle$

**lemma** *embed-transaction-simps* [simp]:  
 $embed-transaction\ (Transaction\ cm\ pm\ views) =$   
 $Transaction_m\ (Unblinded\ ((Unblinded\ cm,\ Unblinded\ pm),\ map\ embed-view\ views))$   
**for** *views*  $\langle proof \rangle$

**end**

### 4.3.1 Inclusion proof for the mediator

**primrec** *mediator-view* ::  $view \Rightarrow view_m$  **where**

$mediator-view\ (View\ vm\ vd\ vs) =$   
 $View_m\ (Unblinded\ ((Unblinded\ vm,\ Blinded\ (Content\ vd)),\ map\ mediator-view\ vs))$

**primrec** *mediator-transaction-tree* ::  $transaction \Rightarrow transaction_m$  **where**  
 $mediator-transaction-tree\ (Transaction\ cm\ pm\ views) =$

$Transaction_m$  (Unblinded ((Unblinded  $cm$ , Blinded (Content  $pm$ )), map mediator-view views))

for views

**lemma** *blinding-of-mediator-view* [simp]: *blinding-of-view* (mediator-view view) (embed-view view)  
 ⟨proof⟩

**lemma** *blinding-of-mediator-transaction-tree*:  
*blinding-of-transaction* (mediator-transaction-tree tt) (embed-transaction tt)  
 ⟨proof⟩

### 4.3.2 Inclusion proofs for participants

Next, we define a function for producing all transaction views from a given view, and prove its properties.

**type-synonym** *view-path-elem* = (view-metadata × view-data) blindable × view list × view list

**type-synonym** *view-path* = view-path-elem list

**type-synonym** *view-zipper* = view-path × view

**type-synonym** *view-path-elem<sub>m</sub>* = (view-metadata<sub>m</sub> ×<sub>m</sub> view-data<sub>m</sub>) × view<sub>m</sub> list<sub>m</sub> × view<sub>m</sub> list<sub>m</sub>

**type-synonym** *view-path<sub>m</sub>* = view-path-elem<sub>m</sub> list

**type-synonym** *view-zipper<sub>m</sub>* = view-path<sub>m</sub> × view<sub>m</sub>

**context begin**

**interpretation** *view-merkle* ⟨proof⟩

**lift-definition** *zipper-of-view* :: view ⇒ view-zipper **is** zipper-of-tree ⟨proof⟩

**lift-definition** *view-of-zipper* :: view-zipper ⇒ view **is** tree-of-zipper ⟨proof⟩

**lift-definition** *zipper-of-view<sub>m</sub>* :: view<sub>m</sub> ⇒ view-zipper<sub>m</sub> **is** zipper-of-tree<sub>m</sub> ⟨proof⟩

**lift-definition** *view-of-zipper<sub>m</sub>* :: view-zipper<sub>m</sub> ⇒ view<sub>m</sub> **is** tree-of-zipper<sub>m</sub> ⟨proof⟩

**lemma** *view-of-zipper<sub>m</sub>-Nil* [simp]: *view-of-zipper<sub>m</sub>* ([], t) = t  
 ⟨proof⟩

**lift-definition** *blind-view-path-elem* :: view-path-elem ⇒ view-path-elem<sub>m</sub> **is**  
*blind-path-elem* embed-view-content hash-view-content ⟨proof⟩

**lift-definition** *blind-view-path* :: view-path ⇒ view-path<sub>m</sub> **is**  
*blind-path* embed-view-content hash-view-content ⟨proof⟩

**lift-definition** *embed-view-path-elem* :: view-path-elem ⇒ view-path-elem<sub>m</sub> **is**  
*embed-path-elem* embed-view-content ⟨proof⟩

**lift-definition** *embed-view-path* :: view-path ⇒ view-path<sub>m</sub> **is**  
*embed-path* embed-view-content ⟨proof⟩

**lift-definition** *hash-view-path-elim* :: *view-path-elim<sub>m</sub>* ⇒ (*view-content<sub>h</sub>* × *view<sub>h</sub>* *list* × *view<sub>h</sub>* *list*) **is**  
*hash-path-elim hash-view-content* ⟨*proof*⟩

**lift-definition** *zippers-view* :: *view-zipper* ⇒ *view-zipper<sub>m</sub>* *list* **is**  
*zippers-rose-tree embed-view-content hash-view-content* ⟨*proof*⟩

**lemma** *embed-view-path-Nil* [*simp*]: *embed-view-path* [] = []  
 ⟨*proof*⟩

**lemma** *zippers-view-same-hash*:  
**assumes** *z* ∈ *set* (*zippers-view* (*p*, *t*))  
**shows** *hash-view* (*view-of-zipper<sub>m</sub>* *z*) = *hash-view* (*view-of-zipper<sub>m</sub>* (*embed-view-path* *p*, *embed-view* *t*))  
 ⟨*proof*⟩

**lemma** *zippers-view-blinding-of*:  
**assumes** *z* ∈ *set* (*zippers-view* (*p*, *t*))  
**shows** *blinding-of-view* (*view-of-zipper<sub>m</sub>* *z*) (*view-of-zipper<sub>m</sub>* (*blind-view-path* *p*, *embed-view* *t*))  
 ⟨*proof*⟩

**end**

**primrec** *blind-view* :: *view* ⇒ *view<sub>m</sub>* **where**  
*blind-view* (*View* *vm* *vd* *subviews*) =  
*View<sub>m</sub>* (*Blinded* (*Content* ((*Content* *vm*, *Content* *vd*), *map* (*hash-view* ∘ *embed-view*) *subviews*)))  
**for** *subviews*

**lemma** *hash-blind-view*: *hash-view* (*blind-view* *view*) = *hash-view* (*embed-view* *view*)  
 ⟨*proof*⟩

**primrec** *blind-transaction* :: *transaction* ⇒ *transaction<sub>m</sub>* **where**  
*blind-transaction* (*Transaction* *cm* *pm* *views*) =  
*Transaction<sub>m</sub>* (*Blinded* (*Content* ((*Content* *cm*, *Content* *pm*), *map* (*hash-view* ∘ *blind-view*) *views*)))  
**for** *views*

**lemma** *hash-blind-transaction*:  
*hash-transaction* (*blind-transaction* *transaction*) = *hash-transaction* (*embed-transaction* *transaction*)  
 ⟨*proof*⟩

**typedecl** *participant*  
**consts** *recipients* :: *view-metadata* ⇒ *participant list*

**fun** *view-recipients* :: *view<sub>m</sub>* ⇒ *participant set* **where**  
*view-recipients* (*View<sub>m</sub>* (*Unblinded* ((*Unblinded* *vm*, *vd*), *subviews*))) = *set* (*recipients*  
*vm*) **for** *subviews*  
| *view-recipients* - = {} — Sane default case

**context** *fixes participant* :: *participant* **begin**

**definition** *view-trees-for* :: *view* ⇒ *view<sub>m</sub> list* **where**

*view-trees-for* *view* =  
*map* *view-of-zipper<sub>m</sub>*  
(*filter* ( $\lambda(-, t). \text{participant} \in \text{view-recipients } t$ )  
(*zippers-view* ([], *view*)))

**primrec** *transaction-views-for* :: *transaction* ⇒ *transaction<sub>m</sub> list* **where**

*transaction-views-for* (*Transaction* *cm* *pm* *views*) =  
*map* ( $\lambda \text{view}_m. \text{Transaction}_m (\text{Unblinded} ((\text{Unblinded } \text{cm}, \text{Unblinded } \text{pm}), \text{view}_m))$ )  
(*concat* (*map* ( $\lambda(l, v, r). \text{map } (\lambda v_m. \text{map } \text{blind-view } l \text{ @ } [v_m] \text{ @ } \text{map } \text{blind-view}$   
*r*) (*view-trees-for* *v*)) (*splits* *views*)))  
**for** *views*

**lemma** *view-trees-for-same-hash*:

*vt* ∈ *set* (*view-trees-for* *view*) ⇒ *hash-view* *vt* = *hash-view* (*embed-view* *view*)  
⟨*proof*⟩

**lemma** *transaction-views-for-same-hash*:

*t<sub>m</sub>* ∈ *set* (*transaction-views-for* *t*) ⇒ *hash-transaction* *t<sub>m</sub>* = *hash-transaction*  
(*embed-transaction* *t*)  
⟨*proof*⟩

**definition** *transaction-projection-for* :: *transaction* ⇒ *transaction<sub>m</sub>* **where**

*transaction-projection-for* *t* =  
(*let* *tvs* = *transaction-views-for* *t*  
in if *tvs* = [] then *blind-transaction* *t* else *the* (*transaction.Merge* (*set* *tvs*)))

**lemma** *transaction-projection-for-same-hash*:

*hash-transaction* (*transaction-projection-for* *t*) = *hash-transaction* (*embed-transaction*  
*t*)  
⟨*proof*⟩

**lemma** *transaction-projection-for-upper*:

**assumes** *t<sub>m</sub>* ∈ *set* (*transaction-views-for* *t*)  
**shows** *blinding-of-transaction* *t<sub>m</sub>* (*transaction-projection-for* *t*)  
⟨*proof*⟩

**end**

**end**