Authenticated Data Structures as Functors

Andreas Lochbihler Ognjen Maric

Digital Asset

March 17, 2025

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, knowledgable 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

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*

⟨*ML*⟩

lemma *vimage2p-mono′*: $R \leq S \Longrightarrow$ *vimage2p f g R* $\leq$ *vimage2p f g S*
⟨*proof*⟩

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*)
⟨*proof*⟩

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-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-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*: $m\ a\ b = Some\ ab \longleftrightarrow bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u)$
⟨*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 $bo :: 'a_m\ blinding\text{-}of$
  assumes *hash*: $bo \leq vimage2p\ h\ h\ (=)$
begin

lemma *blinding-hash-eq*: $bo\ x\ y \Longrightarrow h\ x = h\ y$
⟨*proof*⟩

end

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

+ **assumes** *refl*: $x \in A \Longrightarrow bo\ x\ x$
  and *trans*: $[\![\ bo\ x\ y;\ bo\ y\ z;\ x \in A\ ]\!] \Longrightarrow bo\ x\ z$
  and *antisym*: $[\![\ bo\ x\ y;\ bo\ y\ x;\ x \in A\ ]\!] \Longrightarrow x = y$
begin

lemma *refl-pointfree*: *eq-onp* $(\lambda x.\ x \in A) \leq bo$
⟨*proof*⟩

lemma *blinding-respects-hashes*: *blinding-respects-hashes h bo* ⟨*proof*⟩
lemmas *hash* = *hash*

lemma *trans-pointfree*: *eq-onp* $(\lambda x.\ x \in A)$ *OO bo OO bo* $\leq bo$
⟨*proof*⟩

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

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 $h :: ('a_m, 'a_h)$ *hash*
    and *bo* :: $'a_m$ *blinding-of*
    and $m :: 'a_m$ *merge* +
  assumes *join*: $[\![\ h\ a = h\ b;\ a \in A\ ]\!]$
    $\Longrightarrow \exists ab.\ m\ a\ b = Some\ ab \wedge bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u)$
    and *undefined*: $[\![\ h\ a \neq h\ b;\ a \in A\ ]\!] \Longrightarrow m\ a\ b = None$
begin

lemma *same*: $a \in A \Longrightarrow m\ a\ a = Some\ a$
⟨*proof*⟩

lemma *blinding-of-antisym-on*: *blinding-of-on UNIV h bo* ⟨*proof*⟩

lemma *transp*: *transp bo*
⟨*proof*⟩

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

lemma *respects-hashes*:
$a \in A \Longrightarrow h\ a = h\ b \longleftrightarrow (\exists\, ab.\ m\ a\ b = Some\ ab)$
⟨*proof*⟩

lemma *join′*:
$a \in A \Longrightarrow \forall\, ab.\ m\ a\ b = Some\ ab \longleftrightarrow bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall\, u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u)$
⟨*proof*⟩

lemma *merge-on-subset*:
$B \subseteq A \Longrightarrow$ *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* = *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* ===> *A2*) ===> (*A1* ===> *A1* ===> (⟷)) ===> (⟷))
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 blindable 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* ≡ *id*

abbreviation (*input*) *blinding-of-discrete* :: '*a blinding-of* **where**
blinding-of-discrete ≡ (=)

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 ≤ *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* ≡ *hash-discrete*

abbreviation *blinding-of-unit* :: *unit blinding-of* **where**
blinding-of-unit ≡ *blinding-of-discrete*

abbreviation *merge-unit* :: *unit merge* **where** *merge-unit* ≡ *merge-discrete*

lemma *blinding-of-unit-hash*:
blinding-of-unit ≤ *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 $'a_h$ $blindable_h$ = *Content* $'a_h$ | *Garbage garbage*

datatype ($'a_m$, $'a_h$) $blindable_m$ = *Unblinded* $'a_m$ | *Blinded* $'a_h$ $blindable_h$

2.2.1 Hashes

primrec *hash-blindable'* :: (($'a_h$, $'a_h$) $blindable_m$, $'a_h$ $blindable_h$) *hash* **where**
hash-blindable' (*Unblinded x*) = *Content x*
| *hash-blindable'* (*Blinded x*) = *x*

definition *hash-blindable* :: ($'a_m$, $'a_h$) *hash* ⇒ (($'a_m$, $'a_h$) $blindable_m$, $'a_h$ $blindable_h$) *hash* **where**

hash-blindable h = hash-blindable′ ∘ map-blindable$_m$ h 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$_m$ f′ id x*) = *hash-blindable* (*f o 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$_m$ blinding-of* **where**
blinding-of-blindable (*Unblinded x*) (*Unblinded y*) **if** *bo x y*
| *blinding-of-blindable* (*Blinded x*) *t* **if** *hash-blindable h t* = *x*

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* ≤ *bo′*
shows *blinding-of-blindable h bo* ≤ *blinding-of-blindable h bo′*
⟨*proof*⟩

lemma *blinding-of-blindable-hash*:
assumes *bo* ≤ *vimage2p h h* (=)
shows *blinding-of-blindable h bo* ≤ *vimage2p* (*hash-blindable h*) (*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*$_m$ *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_m, ′a_h) *hash*
fixes *m* :: ′a_m *merge*
begin

fun *merge-blindable* :: (′a_m, ′a_h) *blindable*$_m$ *merge* **where**
merge-blindable (*Unblinded x*) (*Unblinded y*) = *map-option Unblinded* (*m x y*)
| *merge-blindable* (*Blinded 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*$_m$ *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*) ⇒ (*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 ⋀*a b*. ⟦ *a* ∈ *set1-blindable*$_m$ *x*; *b* ∈ *set1-blindable*$_m$ *y* ⟧ ⟹ *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*$_m$ is fixed to be simply *'a blindable*$_h$. We obtain this by instantiating the type variable with the identity building block.

type-synonym *'a nr-blindable* = (*'a*, *'a*) *blindable*$_m$

abbreviation *hash-nr-blindable* :: (*'a nr-blindable*, *'a blindable*$_h$) *hash* **where**
hash-nr-blindable ≡ *hash-blindable hash-discrete*

abbreviation *blinding-of-nr-blindable* :: *'a nr-blindable blinding-of* **where**
blinding-of-nr-blindable ≡ *blinding-of-blindable hash-discrete blinding-of-discrete*

abbreviation *merge-nr-blindable* :: *'a nr-blindable merge* **where**
merge-nr-blindable ≡ *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*$_h$, *'b*$_h$) *sum*$_h$ = *'a*$_h$ + *'b*$_h$

type-notation sum_h (**infixr** ‹$+_h$› *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** ‹$+_m$› *10*)

2.3.1 Hashes

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

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

2.3.2 Blinding

abbreviation (*input*) *blinding-of-sum* :: $'a_m$ *blinding-of* ⇒ $'b_m$ *blinding-of* ⇒ ($'a_m$ +$_m$ $'b_m$) *blinding-of* **where**
blinding-of-sum ≡ *rel-sum*

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

lemma *blinding-of-sum-hash*:
assumes *boa* ≤ *vimage2p rha rha* (=) *bob* ≤ *vimage2p rhb rhb* (=)
shows *blinding-of-sum boa bob* ≤ *vimage2p* (*hash-sum rha rhb*) (*hash-sum rha rhb*) (=)
⟨*proof*⟩

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* ⊆ *A* ∧ *setr x* ⊆ *B*} (*hash-sum rha rhb*) (*blinding-of-sum boa bob*)
(**is** *blinding-of-on ?A ?h ?bo*)
⟨*proof*⟩

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* ⊆ *A* ∧ *setr x* ⊆ *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*) ⇒ *map-option Inl* (*ma x y*)
| (*Inr x*, *Inr y*) ⇒ *map-option Inr* (*mb x y*)
| - ⇒ *None*)
⟨*proof*⟩

end

lemma *merge-sum-cong*[*fundef-cong*]:
⟦ *x* = *x*′; *y* = *y*′;
⋀*xl yl*. ⟦ *x* = *Inl xl*; *y* = *Inl yl* ⟧ ⟹ *ma xl yl* = *ma*′ *xl yl*;
⋀*xr yr*. ⟦ *x* = *Inr xr*; *y* = *Inr yr* ⟧ ⟹ *mb xr yr* = *mb*′ *xr yr* ⟧ ⟹
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_h, 'b_h) *prod*$_h$ = 'a_h × 'b_h
type-notation *prod*$_h$ (‹(- ×$_h$/ -)› [*21*, *20*] *20*)

type-synonym ('a_m, 'b_m) *prod*$_m$ = 'a_m × 'b_m
— If a functor does not introduce blindable positions, then we don't need the type variable copies.
type-notation *prod*$_m$ (‹(- ×$_m$/ -)› [*21*, *20*] *20*)

2.4.1 Hashes

abbreviation (*input*) *hash-prod′* :: (′a_h ×$_h$ ′b_h, ′a_h ×$_h$ ′b_h) *hash* **where**
hash-prod′ ≡ *id*

abbreviation (*input*) *hash-prod* :: (′a_m, ′a_h) *hash* ⇒ (′b_m, ′b_h) *hash* ⇒ (′a_m ×$_m$ ′b_m, ′a_h ×$_h$ ′b_h) *hash*
where *hash-prod* ≡ *map-prod*

2.4.2 Blinding

abbreviation (*input*) *blinding-of-prod* :: ′a_m *blinding-of* ⇒ ′b_m *blinding-of* ⇒ (′a_m ×$_m$ ′b_m) *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_m *merge*
fixes *mb* :: ′b_m *merge*
begin

fun *merge-prod* :: (′a_m ×$_m$ ′b_m) *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 = (λ(*x*, *y*) (*x'*, *y'*). *Option.bind* (*ma x x'*) (λ*x''*. *map-option* (*Pair x''*) (*mb y y'*)))
⟨*proof*⟩

end

lemma *merge-prod-cong*[*fundef-cong*]:
assumes ⋀*a b*. ⟦ *a* ∈ *fsts p1*; *b* ∈ *fsts p2* ⟧ ⟹ *ma a b* = *ma' a b*
and ⋀*a b*. ⟦ *a* ∈ *snds p1*; *b* ∈ *snds p2* ⟧ ⟹ *mb a b* = *mb' a b*
shows *merge-prod ma mb p1 p2* = *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-interface rhb bob mb*
shows *merkle-interface* (*hash-prod rha rhb*) (*blinding-of-prod boa bob*) (*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*) *list-F* = *unit* + '*a* × '*b*

abbreviation (*input*) *set-base-F*$_m$ ≡ λ*x*. *setr x* ⤜ *fsts*
abbreviation (*input*) *set-rec-F*$_m$ ≡ λ*A*. *setr A* ⤜ *snds*
abbreviation (*input*) *map-F* ≡ λ*fb fr*. *map-sum id* (*map-prod fb fr*)

datatype '*a list-R1* = *list-R1* (*unR*: ('*a*, '*a list-R1*) *list-F*)

lemma *list-R1-const-into-dest*: *list-R1 F* = *l* ⟷ *F* = *unR l*
⟨*proof*⟩

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

lemma *list-R1-induct*[*case-names list-R1*]:
assumes ⋀*F*. ⟦ ⋀*l'*. *l'* ∈ *set-rec-F*$_m$ *F* ⟹ *P l'* ⟧ ⟹ *P* (*list-R1 F*)

shows $P\ l$
⟨*proof*⟩

lemma *set-list-R1-eq*:
$\{x.\ \textit{set-base-}F_m\ x \subseteq A \wedge \textit{set-rec-}F_m\ x \subseteq B\} =$
$\{x.\ \textit{setl}\ x \subseteq \textit{UNIV} \wedge \textit{setr}\ x \subseteq \{x.\ \textit{fsts}\ x \subseteq A \wedge \textit{snds}\ x \subseteq B\}\}$
⟨*proof*⟩

2.5.1 The Isomorphism

primrec (*transfer*) *list-R1-to-list* :: *'a list-R1* ⇒ *'a list* **where**
list-R1-to-list (*list-R1 l*) = (*case map-sum id* (*map-prod id list-R1-to-list*) *l of Inl* () ⇒ [] | *Inr* (*x*, *xs*) ⇒ *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*
⟨*proof*⟩

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

primrec (*transfer*) *list-to-list-R1* :: *'a list* ⇒ *'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*
⟨*proof*⟩

lemma *list-of-R1*: *list-to-list-R1* (*list-R1-to-list x*) = *x*
⟨*proof*⟩

lemma *list-R1-def*: *type-definition list-to-list-R1 list-R1-to-list UNIV*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *list-R1-map-trans* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((=) ===> (=)) ===> *pcr-list* (=) ===> *pcr-list* (=)) *map-list-R1 map*
⟨*proof*⟩

lemma *set-list-R1-list-to-list-R1*: *set-list-R1* (*list-to-list-R1 xs*) = *set xs*
⟨*proof*⟩

lemma *list-R1-set-trans* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*pcr-list* (=) ===> (=)) *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*) ⟷ *list-all2 R xs ys*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *list-R1-rel-trans*[*transfer-rule*]: **includes** *lifting-syntax* **shows**
(((=) ===> (=) ===> (=)) ===> *pcr-list* (=) ===> *pcr-list* (=) ===> (=)) *rel-list-R1 list-all2*
⟨*proof*⟩

2.5.2 Hashes

type-synonym ('a_h, 'b_h) *list-F*$_h$ = *unit* +$_h$ 'a_h ×$_h$ 'b_h

type-synonym ('a_m, 'b_m) *list-F*$_m$ = *unit* +$_m$ 'a_m ×$_m$ 'b_m

type-synonym 'a_h *list-R1*$_h$ = 'a_h *list-R1*
— In theory, we should define a separate datatype here of the functor ('a_h, -) *list-F*$_h$. We take a shortcut because they're isomorphic.

type-synonym 'a_m *list-R1*$_m$ = 'a_m *list-R1*
— In theory, we should define a separate datatype here of the functor ('a_m, -) *list-F*$_m$. We take a shortcut because they're isomorphic.

definition *hash-F* :: ('a_m, 'a_h) *hash* ⇒ ('b_m, 'b_h) *hash* ⇒ (('a_m, 'b_m) *list-F*$_m$, ('a_h, 'b_h) *list-F*$_h$) *hash* **where**
hash-F h rhL = *hash-sum hash-unit* (*hash-prod h rhL*)

abbreviation (*input*) *hash-R1* :: ('a_m, 'a_h) *hash* ⇒ ('a_m *list-R1*$_m$, 'a_h *list-R1*$_h$) *hash* **where**
hash-R1 ≡ *map-list-R1*

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

2.5.3 Blinding

definition *blinding-of-F* :: 'a_m *blinding-of* ⇒ 'b_m *blinding-of* ⇒ ('a_m, 'b_m) *list-F*$_m$ *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* ⇒ '*a list-R1 blinding-of* **where**
blinding-of-R1 ≡ *rel-list-R1*

lemma *blinding-of-hash-R1*:
assumes *bo* ≤ *vimage2p h h* (=)
shows *blinding-of-R1 bo* ≤ *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* ⊆ *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_m *merge* ⇒ 'b_m *merge* ⇒ ('a_m, 'b_m) *list-F*$_m$ *merge* **where**

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

lemma *merge-F-cong*[*fundef-cong*]:
assumes ⋀*a b*. ⟦ *a* ∈ *set-base-F*$_m$ *x*; *b* ∈ *set-base-F*$_m$ *y* ⟧ ⟹ *m a b* = *m*′ *a b*
and ⋀*a b*. ⟦ *a* ∈ *set-rec-F*$_m$ *x*; *b* ∈ *set-rec-F*$_m$ *y* ⟧ ⟹ *mL a b* = *mL*′ *a b*
shows *merge-F m mL x y* = *merge-F m*′ *mL*′ *x y*
⟨*proof*⟩

context
fixes *m* :: 'a_m *merge*
notes *setr.simps*[*simp*]
begin
fun *merge-R1* :: 'a_m *list-R1*$_m$ *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* ⊆ *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 $\bigwedge a\ b.\ [\![\ a \in \textit{set-list-R1}\ x;\ b \in \textit{set-list-R1}\ y\ ]\!] \Longrightarrow m\ a\ b = m'\ a\ b$
shows *merge-R1 m x y* = *merge-R1 m′ x y*
⟨*proof*⟩

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* ===> *A* ===> *rel-option A*) ===> *rel-list-R1 A* ===> *rel-list-R1 A* ===> *rel-option* (*rel-list-R1 A*))
merge-R1 merge-R1
⟨*proof*⟩

end

2.5.5 Transferring the Constructions to Lists

type-synonym $'a_h\ list_h = 'a_h\ list$
type-synonym $'a_m\ list_m = 'a_m\ list$

context begin
interpretation *list-R1* ⟨*proof*⟩

abbreviation (*input*) *hash-list* :: $('a_m, 'a_h)\ hash \Rightarrow ('a_m\ list_m, 'a_h\ list_h)\ hash$
where *hash-list* ≡ *map*
abbreviation (*input*) *blinding-of-list* :: $'a_m\ blinding\text{-}of \Rightarrow 'a_m\ list_m\ blinding\text{-}of$
where *blinding-of-list* ≡ *list-all2*
lift-definition *merge-list* :: $'a_m\ merge \Rightarrow 'a_m\ list_m\ merge$ **is** *merge-R1* ⟨*proof*⟩

lemma *blinding-of-list-mono*:
$[\![\ \bigwedge x\ y.\ bo\ x\ y \longrightarrow bo'\ x\ y\ ]\!] \Longrightarrow$
blinding-of-list bo x y ⟶ *blinding-of-list bo′ x y*
⟨*proof*⟩

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

lemma *blinding-of-list-mono-pred*:
$R \leq R' \Longrightarrow$ *blinding-of-list R* ≤ *blinding-of-list R′*
⟨*proof*⟩

lemma *blinding-of-list-simp*: *blinding-of-list* = *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 $('a, 'b_h)$ $fun_h = 'a \Rightarrow 'b_h$
type-notation fun_h (**infixr** $\langle\Rightarrow_h\rangle$ *0*)

type-synonym $('a, 'b_m)$ $fun_m = 'a \Rightarrow 'b_m$
type-notation fun_m (**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'* :: $('a \Rightarrow_m 'b_h, 'a \Rightarrow_h 'b_h)$ *hash* **where**
hash-fun' ≡ *id*

abbreviation (*input*) *hash-fun* :: $('b_m, 'b_h)$ *hash* ⇒ $('a \Rightarrow_m 'b_m, 'a \Rightarrow_h 'b_h)$ *hash*
where *hash-fun* ≡ *comp*

2.6.2 Blinding

abbreviation (*input*) *blinding-of-fun* :: $'b_m$ *blinding-of* ⇒ $('a \Rightarrow_m 'b_m)$ *blinding-of*
where
blinding-of-fun ≡ *rel-fun* (=)

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

lemma *blinding-of-fun-hash*:
assumes *bo* ≤ *vimage2p rh rh* (=)
shows *blinding-of-fun bo* ≤ *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*. *range x* ⊆ *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 $m :: 'b_m$ *merge*
begin

definition *merge-fun* :: (*'a* ⇒$_m$ *'b*$_m$) *merge* **where**
merge-fun f g = (*if* ∀ *x*. *m* (*f x*) (*g x*) ≠ *None then Some* (λ*x*. *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*. *range x* ⊆ *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 ⋀*a b*. ⟦ *a* ∈ *range f*; *b* ∈ *range g* ⟧ ⟹ *m a b* = *m' a b*
shows *merge-fun m f g* = *merge-fun m' f g*
⟨*proof*⟩

lemma *is-none-alt-def*: *Option.is-none x* ⟷ (*case x of None* ⇒ *True* | *Some* - ⇒ *False*)
⟨*proof*⟩

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

lemma *merge-fun-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*A* ===> *B* ===> *rel-option C*) ===> ((=) ===> *A*) ===> ((=) ===> *B*) ===> *rel-option* ((=) ===> *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*$_h$ $= ('a_h \times_h 'b_h$ *list*$_h)$ *blindable*$_h$

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

datatype $('a_m, 'a_h)$ *rose-tree*$_m$ $=$ *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*$_m$
:: $('ma \Rightarrow 'a) \Rightarrow ('mr \Rightarrow 'r) \Rightarrow ('ma, 'ha, 'mr, 'hr)$ *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*$_m$
:: $('a_m, 'a_h)$ *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*$_m$*-alt-def*: *hash-rt-F*$_m$ *h rhm* $=$ *hash-blindable* (*map-prod h* (*map rhm*))
⟨*proof*⟩

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

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

definition *hash-tree*
:: ('a_m, 'a_h) *hash* ⇒ (('a_m, 'a_h) *rose-tree*$_m$, 'a_h *rose-tree*$_h$) *hash* **where**
hash-tree h = *hash-rt-tree' o map-rose-tree*$_m$ *h id*

lemma *blindable*$_m$*-map-compositionality*:
map-blindable$_m$ *f g o map-blindable*$_m$ *f' g'* = *map-blindable*$_m$ (*f o f'*) (*g o g'*)
⟨*proof*⟩

lemma *hash-tree-simps* [*simp*]:
hash-tree h (*Tree*$_m$ *x*) = *Tree*$_h$ (*hash-rt-F*$_m$ *h* (*hash-tree h*) *x*)
⟨*proof*⟩

parametric-constant *hash-rt-F*$_m$*-parametric* [*transfer-rule*]: *hash-rt-F*$_m$*-alt-def*

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

2.7.2 Blinding

abbreviation (*input*) *blinding-of-rt-F*$_m$
:: ('a_m, 'a_h) *hash* ⇒ 'a_m *blinding-of* ⇒ ('b_m, 'b_h) *hash* ⇒ 'b_m *blinding-of*
⇒ ('a_m, 'a_h, 'b_m, 'b_h) *rose-tree-F*$_m$ *blinding-of* **where**
blinding-of-rt-F$_m$ *ha boa hb bob* ≡ *blinding-of-blindable* (*hash-prod ha* (*map hb*)) (*blinding-of-prod boa* (*blinding-of-list bob*))

lemma *blinding-of-rt-F*$_m$*-mono*:
⟦ *boa* ≤ *boa'*; *bob* ≤ *bob'* ⟧ ⟹ *blinding-of-rt-F*$_m$ *ha boa hb bob* ≤ *blinding-of-rt-F*$_m$ *ha boa' hb bob'*
⟨*proof*⟩

lemma *blinding-of-rt-F*$_m$*-mono-inductive*:
assumes ⋀*x y*. *boa x y* ⟶ *boa' x y* ⋀*x y*. *bob x y* ⟶ *bob' x y*
shows *blinding-of-rt-F*$_m$ *ha boa hb bob x y* ⟶ *blinding-of-rt-F*$_m$ *ha boa' hb bob' x y*
⟨*proof*⟩

context
fixes *h* :: ('a_m, 'a_h) *hash*
and *bo* :: 'a_m *blinding-of*
begin

inductive *blinding-of-tree* :: ('a_m, 'a_h) *rose-tree*$_m$ *blinding-of* **where**
blinding-of-tree (*Tree*$_m$ *t1*) (*Tree*$_m$ *t2*)
if *blinding-of-rt-F*$_m$ *h bo* (*hash-tree h*) *blinding-of-tree t1 t2*
monos *blinding-of-rt-F*$_m$*-mono-inductive*

end

inductive-simps *blinding-of-tree-simps* [*simp*]:
blinding-of-tree h bo (*Tree*$_m$ *t1*) (*Tree*$_m$ *t2*)

lemma *blinding-of-rt-F*$_m$*-hash*:
assumes *boa* $\leq$ *vimage2p ha ha* (=) *bob* $\leq$ *vimage2p hb hb* (=)
shows *blinding-of-rt-F*$_m$ *ha boa hb bob* $\leq$ *vimage2p* (*hash-rt-F*$_m$ *ha hb*) (*hash-rt-F*$_m$ *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*$_m$ :: ('a_m, 'a_h, 'b_h, 'b_m) *rose-tree-F*$_m$ $\Rightarrow$ 'a_m *set*
where
set1-rt-F$_m$ *x* $\equiv$ *set1-blindable*$_m$ *x* $\gg=$ *fsts*

abbreviation (*input*) *set3-rt-F*$_m$ :: ('a_m, 'a_h, 'b_m, 'b_h) *rose-tree-F*$_m$ $\Rightarrow$ 'b_m *set*
where
set3-rt-F$_m$ *x* $\equiv$ (*set1-blindable*$_m$ *x* $\gg=$ *snds*) $\gg=$ *set*

lemma *set-rt-F*$_m$*-eq*:
{*x*. *set1-rt-F*$_m$ *x* $\subseteq$ *A* $\wedge$ *set3-rt-F*$_m$ *x* $\subseteq$ *B*} =
{*x*. *set1-blindable*$_m$ *x* $\subseteq$ {*x*. *fsts x* $\subseteq$ *A* $\wedge$ *snds x* $\subseteq$ {*x*. *set x* $\subseteq$ *B*}}}
⟨*proof*⟩

lemma *hash-blindable-map*: *hash-blindable f* $\circ$ *map-blindable*$_m$ *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*. *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'* $\Longrightarrow$ *blinding-of-tree h bo* $\leq$ *blinding-of-tree h bo'*
⟨*proof*⟩

2.7.3 Merging

definition *merge-rt-F*$_m$
:: (*'a*$_m$, *'a*$_h$) *hash* ⇒ *'a*$_m$ *merge* ⇒ (*'b*$_m$, *'b*$_h$) *hash* ⇒ *'b*$_m$ *merge* ⇒
(*'a*$_m$, *'a*$_h$, *'b*$_m$, *'b*$_h$) *rose-tree-F*$_m$ *merge*
where
merge-rt-F$_m$ *ha ma hr mr* ≡ *merge-blindable* (*hash-prod ha* (*hash-list hr*)) (*merge-prod ma* (*merge-list mr*))

lemma *merge-rt-F*$_m$*-cong* [*fundef-cong*]:
assumes ⋀*a b*. ⟦ *a* ∈ *set1-rt-F*$_m$ *x*; *b* ∈ *set1-rt-F*$_m$ *y* ⟧ ⟹ *ma a b* = *ma' a b*
and ⋀*a b*. ⟦ *a* ∈ *set3-rt-F*$_m$ *x*; *b* ∈ *set3-rt-F*$_m$ *y* ⟧ ⟹ *mm a b* = *mm' a b*
shows *merge-rt-F*$_m$ *ha ma hm mm x y* = *merge-rt-F*$_m$ *ha ma' hm mm' x y*
⟨*proof*⟩

lemma *in-set1-blindable*$_m$*-iff*: *x* ∈ *set1-blindable*$_m$ *y* ⟷ *y* = *Unblinded x*
⟨*proof*⟩

context
fixes *h* :: (*'a*$_m$, *'a*$_h$) *hash*
and *ma* :: *'a*$_m$ *merge*
notes *in-set1-blindable*$_m$*-iff*[*simp*]
begin
fun *merge-tree* :: (*'a*$_m$, *'a*$_h$) *rose-tree*$_m$ *merge* **where**
merge-tree (*Tree*$_m$ *x*) (*Tree*$_m$ *y*) = *map-option Tree*$_m$ (
merge-rt-F$_m$ *h ma* (*hash-tree h*) *merge-tree x y*)
end

lemma *merge-on-tree* [*locale-witness*]:
assumes *merge-on A h bo m*
shows *merge-on* {*x*. *set1-rose-tree*$_m$ *x* ⊆ *A*} (*hash-tree h*) (*blinding-of-tree h bo*) (*merge-tree h m*)
(**is** *merge-on ?A ?h ?bo ?m*)
⟨*proof*⟩

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

lemma *option-bind-comm*:
((*x* :: *'a option*) ⤜ (λ*y*. *c* ⤜ (λ*z*. *f y z*))) = (*c* ⤜ (λ*y*. *x* ⤜ (λ*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 bo m*
shows *merkle-interface* (*hash-tree h*) (*blinding-of-tree h bo*) (*merge-tree h m*)
⟨*proof*⟩

lemma *merge-tree-cong* [*fundef-cong*]:
assumes $\bigwedge a\ b.\ [\![\ a \in \textit{set1-rose-tree}_m\ x;\ b \in \textit{set1-rose-tree}_m\ y\ ]\!] \Longrightarrow m\ a\ b = m'\ a\ b$
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) $('a, 'b)\ F$, where $'a$ is the type of "source" data, and $'b$ 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 $('a, 'b)\ F$ [*wits*: $'a \Rightarrow ('a, 'b)\ F$]

context notes [[*typedef-overloaded*]] **begin**
datatype $'a\ T = T\ ('a, 'a\ T)\ F$
end

3.1.2 Base Merkle functor

This type captures the ADS hashes.

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

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

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

3.1.3 Least fixpoint

context notes [[*typedef-overloaded*]] **begin**
datatype $'a_h$ T_h = T_h $('a_h, 'a_h$ $T_h)$ F_h
end

context notes [[*typedef-overloaded*]] **begin**
datatype $('a_m, 'a_h)$ T_m = T_m (*the-*T_m: $('a_m, 'a_h, ('a_m, 'a_h)$ $T_m, 'a_h$ $T_h)$ F_m)
end

3.1.4 Composition

Finally, we show how to compose two Merkle functors. For simplicity, we reuse $('a, 'b)$ F and $'a$ T.

context notes [[*typedef-overloaded*]] **begin**

datatype $('a, 'b)$ G = G $('a$ $T, 'b)$ F

datatype $('a_h, 'b_h)$ G_h = G_h (*the-*G_h: $('a_h$ $T_h, 'b_h)$ F_h)

datatype $('a_m, 'a_h, 'b_m, 'b_h)$ G_m = G_m (*the-*G_m: $(('a_m, 'a_h)$ $T_m, 'a_h$ $T_h, 'b_m, 'b_h)$ 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 $('a', 'b')$ F_h. (Actually, we want to use an abstract datatype for root hashes, but we omit this distinction here for simplicity.)

consts *root-hash-*F' :: $(('a_h, 'a_h, 'b_h, 'b_h)$ $F_m, ('a_h, 'b_h)$ $F_h)$ *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 $('a_m, 'a_h, 'b_m, 'b_h)$ *hash-F* =
$('a_m, 'a_h)$ *hash* ⇒ $('b_m, 'b_h)$ *hash* ⇒ $(('a_m, 'a_h, 'b_m, 'b_h)\ F_m, ('a_h, 'b_h)\ F_h)$ *hash*
definition *root-hash-F* :: $('a_m, 'a_h, 'b_m, 'b_h)$ *hash-F* **where**
root-hash-F rha rhb = *root-hash-F'* ∘ *map-F*$_m$ *rha id rhb id*

3.2.2 Least fixpoint

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

definition *root-hash-T* :: $('a_m, 'a_h)$ *hash* ⇒ $(('a_m, 'a_h)\ T_m, 'a_h\ T_h)$ *hash* **where**
root-hash-T rha = *root-hash-T'* ∘ *map-T*$_m$ *rha id*

lemma *root-hash-T-simps* [*simp*]:
root-hash-T rha $(T_m\ x)$ = T_h (*root-hash-F rha* (*root-hash-T rha*) *x*)
⟨*proof*⟩

3.2.3 Composition

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

definition *root-hash-G* :: $('a_m, 'a_h)$ *hash* ⇒ $('b_m, 'b_h)$ *hash* ⇒ $(('a_m, 'a_h, 'b_m, 'b_h)\ G_m, ('a_h, 'b_h)\ G_h)$ *hash* **where**
root-hash-G rha rhb = *root-hash-G'* ∘ *map-G*$_m$ *rha id rhb id*

lemma *root-hash-G-unfold*:
root-hash-G rha rhb = G_h ∘ *root-hash-F* (*root-hash-T rha*) *rhb* ∘ *the-G*$_m$
⟨*proof*⟩

lemma *root-hash-G-simps* [*simp*]:
root-hash-G rha rhb $(G_m\ x)$ = G_h (*root-hash-F* (*root-hash-T rha*) *rhb x*)
⟨*proof*⟩

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 $('a_m, 'a_h, 'b_m, 'b_h)$ *blinding-of-F* =
$('a_m, 'a_h)$ *hash* ⇒ $'a_m$ *blinding-of* ⇒ $('b_m, 'b_h)$ *hash* ⇒ $'b_m$ *blinding-of* ⇒ $('a_m, 'a_h, 'b_m, 'b_h)\ F_m$ *blinding-of*

— Computes whether a partially blinded ADS is a blinding of another one
axiomatization *blinding-of-F* :: $('a_m, 'a_h, 'b_m, 'b_h)$ *blinding-of-F* **where**
blinding-of-F-mono: ⟦ *boa* ≤ *boa'*; *bob* ≤ *bob'* ⟧
⟹ *blinding-of-F rha boa rhb bob* ≤ *blinding-of-F rha boa' rhb 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*]:
⟦ *blinding-respects-hashes rha boa*; *blinding-respects-hashes rhb bob* ⟧
⟹ *blinding-respects-hashes* (*root-hash-F rha rhb*) (*blinding-of-F rha boa rhb bob*)
and *blinding-of-on-F* [*locale-witness*]:
⟦ *blinding-of-on A rha boa*; *blinding-of-on B rhb bob* ⟧
⟹ *blinding-of-on* {*x*. *set1-F*$_m$ *x* ⊆ *A* ∧ *set3-F*$_m$ *x* ⊆ *B*} (*root-hash-F rha rhb*) (*blinding-of-F rha boa rhb bob*)

lemma *blinding-of-F-mono-inductive*:
assumes *a*: ⋀*x y*. *boa x y* ⟶ *boa' x y*
and *b*: ⋀*x y*. *bob x y* ⟶ *bob' x y*
shows *blinding-of-F rha boa rhb bob x y* ⟶ *blinding-of-F rha boa' rhb bob' x y*
⟨*proof*⟩

3.3.2 Blinding on least fixpoints

context
fixes *rh* :: $('a_m, 'a_h)$ *hash*
and *bo* :: $'a_m$ *blinding-of*
begin

inductive *blinding-of-T* :: $('a_m, 'a_h)$ T_m *blinding-of* **where**
blinding-of-T (T_m *x*) (T_m *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 ⟹ *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*. *set1-T*$_m$ *x* ⊆ *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*$_m$ *the-G*$_m$ (*blinding-of-F* (*root-hash-T rha*) (*blinding-of-T rha boa*) *rhb bob*)
⟨*proof*⟩

end

lemma *blinding-of-G-mono*:
assumes *boa* ≤ *boa*′ *bob* ≤ *bob*′
shows *blinding-of-G rha boa rhb bob* ≤ *blinding-of-G rha boa*′ *rhb bob*′
⟨*proof*⟩

lemma *blinding-of-G-root-hash*:
assumes *boa* ≤ *vimage2p rha rha* (=) **and** *bob* ≤ *vimage2p rhb rhb* (=)
shows *blinding-of-G rha boa rhb bob* ≤ *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*. *set1-G*$_m$ *x* ⊆ *A* ∧ *set3-G*$_m$ *x* ⊆ *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*]:
$[\![\bigwedge a\ b.\ a \in$ *set1-F*$_m$ $x \Longrightarrow$ *ma a b* $=$ *ma'* *a b*; $\bigwedge a\ b.\ a \in$ *set3-F*$_m$ $x \Longrightarrow$ *mb a b* $=$ *mb'* *a b* $]\!]$
$\Longrightarrow$ *merge-F rha ma rhb mb x y* $=$ *merge-F rha ma' rhb mb' x y*
and
merge-on-F [*locale-witness*]:
$[\![$ *merge-on A rha boa ma*; *merge-on B rhb bob mb* $]\!]$
$\Longrightarrow$ *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*$_m$ $y \Longrightarrow y \neq$ *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* $\{x.\ set1\text{-}T_m\ x \subseteq A\}$ (*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 $\bigwedge a\ b.\ a \in set1\text{-}T_m\ x \Longrightarrow m\ a\ b = m'\ a\ b$
shows *merge-T rh m x y* = *merge-T rh m' x y*
⟨*proof*⟩

3.4.3 Merging and composition

context
fixes *rha* :: $('a_m, 'a_h)$ *hash*
fixes *ma* :: $'a_m$ *merge*
fixes *rhb* :: $('b_m, 'b_h)$ *hash*
fixes *mb* :: $'b_m$ *merge*
begin

primrec *merge-G* :: $('a_m, 'a_h, 'b_m, 'b_h)\ G_m$ *merge* **where**
merge-G (G_m *x*) *y'* = (*case y' of* G_m *y* ⇒
map-option G_m (*merge-F* (*root-hash-T rha*) (*merge-T rha ma*) *rhb mb x y*))

lemma *merge-G-simps* [*simp*]:
merge-G (G_m *x*) (G_m *y*) = *map-option* G_m (*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* $\{x.\ set1\text{-}G_m\ x \subseteq A \wedge set3\text{-}G_m\ x \subseteq B\}$ (*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*]:
⟦ $\bigwedge a\ b.\ a \in set1\text{-}G_m\ x \Longrightarrow ma\ a\ b = ma'\ a\ b$; $\bigwedge a\ b.\ a \in set3\text{-}G_m\ x \Longrightarrow mb\ a\ b = mb'\ a\ b$ ⟧

$\Longrightarrow$ *merge-G rha ma rhb mb x y = merge-G rha ma′ rhb mb′ x y*
⟨*proof*⟩

end

theory *Inclusion-Proof-Construction* **imports**
ADS-Construction
begin

primrec *blind-blindable* :: $('a_m \Rightarrow 'a_h) \Rightarrow ('a_m, 'a_h)\ blindable_m \Rightarrow ('a_m, 'a_h)\ blindable_m$ **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 $h :: 'a \Rightarrow 'a_h$ **begin**
fun *hash-source-tree* :: $'a$ *rose-tree* $\Rightarrow 'a_h$ $rose\text{-}tree_h$ **where**
hash-source-tree (*Tree* (*data*, *subtrees*)) = $Tree_h$ (*Content* (*h data*, *map hash-source-tree subtrees*))
end

context fixes $e :: 'a \Rightarrow 'a_m$ **begin**
fun *embed-source-tree* :: $'a$ *rose-tree* $\Rightarrow ('a_m, 'a_h)$ $rose\text{-}tree_m$ **where**
embed-source-tree (*Tree* (*data*, *subtrees*)) =
$Tree_m$ (*Unblinded* (*e data*, *map embed-source-tree subtrees*))
end

context fixes $h :: 'a \Rightarrow 'a_h$ **begin**
fun *blind-source-tree* :: $'a$ *rose-tree* $\Rightarrow ('a_m, 'a_h)$ $rose\text{-}tree_m$ **where**
blind-source-tree (*Tree* (*data*, *subtrees*)) = $Tree_m$ (*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* :: $('a_m, 'a_h)$ $rose\text{-}tree_m \Rightarrow$ *bool* **where**
is-blinded ($Tree_m$ (*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* ∘ *e*) *st*
⟨*proof*⟩

lemma *blinded-embedded-same-hash*:
hash-tree h'' (*blind-source-tree* (*h o 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 o e*) *st*) (*embed-source-tree e st*)
⟨*proof*⟩

fun *embed-hash-tree* :: *'ha rose-tree*$_h$ ⇒ (*'a*, *'ha*) *rose-tree*$_m$ **where**
embed-hash-tree (*Tree*$_h$ *h*) = *Tree*$_m$ (*Blinded h*)

3.5.2 Auxiliary definitions: selectors and list splits

fun *children* :: *'a rose-tree* ⇒ *'a rose-tree list* **where**
children (*Tree* (*data*, *subtrees*)) = *subtrees*

fun *children*$_m$ :: (*'a*, *'a*$_h$) *rose-tree*$_m$ ⇒ (*'a*, *'a*$_h$) *rose-tree*$_m$ *list* **where**
children$_m$ (*Tree*$_m$ (*Unblinded* (*data*, *subtrees*))) = *subtrees*
| *children*$_m$ - = *undefined*

fun *splits* :: *'a list* ⇒ (*'a list* × *'a* × *'a list*) *list* **where**
splits [] = []
| *splits* (*x*#*xs*) = ([], *x*, *xs*) # *map* (λ(*l*, *y*, *r*). (*x* # *l*, *y*, *r*)) (*splits xs*)

lemma *splits-iff*: (*l*, *a*, *r*) ∈ *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* × *'a rose-tree list* × *'a rose-tree list*
type-synonym *'a path* = *'a path-elem list*
type-synonym *'a zipper* = *'a path* × *'a rose-tree*

definition *zipper-of-tree* :: *'a rose-tree* ⇒ *'a zipper* **where**
zipper-of-tree t ≡ ([], *t*)

fun *tree-of-zipper* :: *'a zipper* ⇒ *'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* ⇒ *'a zipper list* **where**
zipper-children (*p*, *Tree* (*a*, *ts*)) = *map* (λ(*l*, *t*, *r*). ((*a*, *l*, *r*) # *p*, *t*)) (*splits ts*)

lemma *zipper-children-same-tree*:
assumes *z'* ∈ *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 × ('a_m, 'a_h) *rose-tree*$_m$ *list* × ('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*$_m$ × ('a_m, 'a_h) *rose-tree*$_m$

definition *zipper-of-tree*$_m$ :: ('a_m, 'a_h) *rose-tree*$_m$ ⇒ ('a_m, 'a_h) *zipper*$_m$ **where**
zipper-of-tree$_m$ *t* ≡ ([], *t*)

fun *tree-of-zipper*$_m$ :: ('a_m, 'a_h) *zipper*$_m$ ⇒ ('a_m, 'a_h) *rose-tree*$_m$ **where**
tree-of-zipper$_m$ ([], *t*) = *t*
| *tree-of-zipper*$_m$ ((*m*, *l*, *r*) # *z*, *t*) = *tree-of-zipper*$_m$ (*z*, *Tree*$_m$ (*Unblinded* (*m*, *l* @ *t* # *r*)))

lemma *tree-of-zipper*$_m$*-append*:
tree-of-zipper$_m$ (*p* @ *p'*, *t*) = *tree-of-zipper*$_m$ (*p'*, *tree-of-zipper*$_m$ (*p*, *t*))
⟨*proof*⟩

fun *zipper-children*$_m$ :: ('a_m, 'a_h) *zipper*$_m$ ⇒ ('a_m, 'a_h) *zipper*$_m$ *list* **where**
zipper-children$_m$ (*p*, *Tree*$_m$ (*Unblinded* (*a*, *ts*))) = *map* (λ(*l*, *t*, *r*). ((*a*, *l*, *r*) # *p*, *t*)) (*splits ts*)
| *zipper-children*$_m$ - = []

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

fun *blind-path-elem* :: ('*a* ⇒ 'a_m) ⇒ ('a_m ⇒ 'a_h) ⇒ *'a path-elem* ⇒ ('a_m, 'a_h) *path-elem*$_m$ **where**
blind-path-elem e h (*x*, *l*, *r*) = (*e x*, *map* (*blind-source-tree* (*h* ∘ *e*)) *l*, *map* (*blind-source-tree* (*h* ∘ *e*)) *r*)

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

definition *blind-path* :: $('a \Rightarrow 'a_m) \Rightarrow ('a_m \Rightarrow 'a_h) \Rightarrow 'a\ path \Rightarrow ('a_m, 'a_h)\ path_m$
where
blind-path e h ≡ *map* (*blind-path-elem e h*)

fun *embed-path-elem* :: $('a \Rightarrow 'a_m) \Rightarrow 'a\ path\text{-}elem \Rightarrow ('a_m, 'a_h)\ path\text{-}elem_m$ **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* :: $('a \Rightarrow 'a_m) \Rightarrow 'a\ path \Rightarrow ('a_m, 'a_h)\ path_m$ **where**
embed-path embed-elem ≡ *map* (*embed-path-elem embed-elem*)

lemma *hash-tree-of-zipper-same-path*:
hash-tree h (*tree-of-zipper*$_m$ (*p*, *v*)) = *hash-tree h* (*tree-of-zipper*$_m$ (*p*, *v'*))
⟷ *hash-tree h v* = *hash-tree h v'*
⟨*proof*⟩

fun *hash-path-elem* :: $('a_m \Rightarrow 'a_h) \Rightarrow ('a_m, 'a_h)\ path\text{-}elem_m \Rightarrow ('a_h \times 'a_h\ rose\text{-}tree_h\ list \times 'a_h\ rose\text{-}tree_h\ list)$ **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*$_m$ (*p*, *v*)) = *hash-tree h'* (*tree-of-zipper*$_m$ (*p'*, *v'*))
⟨*proof*⟩

lemma *blind-embed-path-same-hash*:
hash-tree h (*tree-of-zipper*$_m$ (*blind-path e h p*, *t*)) = *hash-tree h* (*tree-of-zipper*$_m$ (*embed-path e p*, *t*))
⟨*proof*⟩

lemma *tree-of-embed-commute*:
tree-of-zipper$_m$ (*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$_m$ (*embed-path e p*, *embed-source-tree e* (*Tree* (*d*, *ts*)))
= *tree-of-zipper*$_m$ (*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*$_m$ (*p*, *t*)) (*tree-of-zipper*$_m$ (*p*, *t'*))
⟷ *blinding-of-tree h bo t t'*
⟨*proof*⟩

lemma *zipper-children-size-change* [*termination-simp*]: $(a, b) \in set\ (zipper\text{-}children\ (p, v)) \Longrightarrow size\ b < size\ v$
⟨*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* :: $'a\ zipper \Rightarrow ('a_m, 'a_h)\ zipper_m\ list$ **where**
zippers-rose-tree $(p, t) = (blind\text{-}path\ e\ h\ p, embed\text{-}source\text{-}tree\ e\ t) \#$
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 $z \in set\ (zippers\text{-}rose\text{-}tree\ e\ h\ (p, t))$
shows $hash\text{-}tree\ h\ (tree\text{-}of\text{-}zipper_m\ z) =$
$hash\text{-}tree\ h\ (tree\text{-}of\text{-}zipper_m\ (embed\text{-}path\ e\ p, embed\text{-}source\text{-}tree\ e\ t))$
⟨*proof*⟩

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

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

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

context *merkle-interface* **begin**

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

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

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

lemma *Merge-empty* [*simp*]: *Merge* {} = *None*
⟨*proof*⟩

lemma *Merge-infinite* [*simp*]: *infinite A* ⟹ *Merge A* = *None*
⟨*proof*⟩

lemma *Merge-cong-start*:
Finite-Set.fold (λ*x yo*. *yo* ⤜ *m x*) (*Some x*) *A* = *Finite-Set.fold* (λ*x yo*. *yo* ⤜ *m x*) (*Some y*) *A* (**is** *?lhs* = *?rhs*)
if *x* ∈ *A y* ∈ *A finite A*
⟨*proof*⟩

lemma *Merge-insert* [*simp*]: *Merge* (*insert x A*) = (*if A* = {} *then Some x else Merge A* ⤜ *m x*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Merge-insert-alt*:
Merge (*insert x A*) = *Finite-Set.fold* (λ*x yo*. *yo* ⤜ *m x*) (*Some x*) *A* (**is** *?lhs* = *?rhs*) **if** *finite A*
⟨*proof*⟩

lemma *Merge-None* [*simp*]: *Finite-Set.fold* (λ*x yo*. *yo* ⤜ *m x*) *None A* = *None*
⟨*proof*⟩

lemma *Merge-union*:
Merge (*A* ∪ *B*) = (*if A* = {} *then Merge B else if B* = {} *then Merge A else* (*Merge A* ⤜ (λ*a*. *Merge B* ⤜ *m a*)))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Merge-upper*:
assumes *m*: *Merge A* = *Some x* **and** *y*: *y* ∈ *A*
shows *bo y x*
⟨*proof*⟩

lemma *Merge-least*:
assumes *m*: *Merge A* = *Some x* **and** *u*[*rule-format*]: ∀ *a*∈*A*. *bo a u*
shows *bo x u*
⟨*proof*⟩

lemma *Merge-defined*:
assumes *finite A A* ≠ {} ∀ *a*∈*A*. ∀ *b* ∈ *A*. *h a* = *h b*
shows *Merge A* ≠ *None*
⟨*proof*⟩

lemma *Merge-hash*:
assumes *Merge A* = *Some x a* ∈ *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 $\mathit{view\text{-}metadata}_h$ = *view-metadata* $\mathit{blindable}_h$
type-synonym $\mathit{view\text{-}data}_h$ = *view-data* $\mathit{blindable}_h$

datatype view_h = View_h (($\mathit{view\text{-}metadata}_h$ $\times_h$ $\mathit{view\text{-}data}_h$) $\times_h$ view_h list_h) $\mathit{blindable}_h$

type-synonym $\mathit{view\text{-}metadata}_m$ = (*view-metadata*, *view-metadata*) $\mathit{blindable}_m$
type-synonym $\mathit{view\text{-}data}_m$ = (*view-data*, *view-data*) $\mathit{blindable}_m$

datatype view_m = View_m
(($\mathit{view\text{-}metadata}_m$ $\times_m$ $\mathit{view\text{-}data}_m$) $\times_m$ view_m list_m,
($\mathit{view\text{-}metadata}_h$ $\times_h$ $\mathit{view\text{-}data}_h$) $\times_h$ view_h list_h) $\mathit{blindable}_m$

abbreviation (*input*) *hash-view-data* :: ($\mathit{view\text{-}data}_m$, $\mathit{view\text{-}data}_h$) *hash* **where**
hash-view-data ≡ *hash-blindable id*
abbreviation (*input*) *blinding-of-view-data* :: $\mathit{view\text{-}data}_m$ *blinding-of* **where**
blinding-of-view-data ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-view-data* :: $\mathit{view\text{-}data}_m$ *merge* **where**
merge-view-data ≡ *merge-blindable id merge-discrete*

lemma *merkle-view-data*:
merkle-interface hash-view-data blinding-of-view-data merge-view-data
⟨*proof*⟩

abbreviation (*input*) *hash-view-metadata* :: ($\mathit{view\text{-}metadata}_m$, $\mathit{view\text{-}metadata}_h$) *hash* **where**
hash-view-metadata ≡ *hash-blindable id*

abbreviation (*input*) *blinding-of-view-metadata* :: $view\text{-}metadata_m$ *blinding-of* **where**
blinding-of-view-metadata ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-view-metadata* :: $view\text{-}metadata_m$ *merge* **where**
merge-view-metadata ≡ *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* × *view-data*
type-synonym $view\text{-}content_h$ = $view\text{-}metadata_h$ ×$_h$ $view\text{-}data_h$
type-synonym $view\text{-}content_m$ = $view\text{-}metadata_m$ ×$_m$ $view\text{-}data_m$

locale *view-merkle* **begin**

type-synonym $view_h{}'$ = $view\text{-}content_h$ $rose\text{-}tree_h$

primrec $from\text{-}view_h$ :: $view_h$ ⇒ $view_h{}'$ **where**
$from\text{-}view_h$ ($View_h$ *x*) = $Tree_h$ ($map\text{-}blindable_h$ (*map-prod id* (*map* $from\text{-}view_h$)) *x*)

primrec $to\text{-}view_h$ :: $view_h{}'$ ⇒ $view_h$ **where**
$to\text{-}view_h$ ($Tree_h$ *x*) = $View_h$ ($map\text{-}blindable_h$ (*map-prod id* (*map* $to\text{-}view_h$)) *x*)

lemma $from\text{-}to\text{-}view_h$ [*simp*]: $from\text{-}view_h$ ($to\text{-}view_h$ *x*) = *x*
⟨*proof*⟩

lemma $to\text{-}from\text{-}view_h$ [*simp*]: $to\text{-}view_h$ ($from\text{-}view_h$ *x*) = *x*
⟨*proof*⟩

lemma $iso\text{-}view_h$: *type-definition* $from\text{-}view_h$ $to\text{-}view_h$ *UNIV*
⟨*proof*⟩

setup-lifting $iso\text{-}view_h$

lemma $cr\text{-}view_h$-*Grp*: $cr\text{-}view_h$ = *Grp UNIV* $to\text{-}view_h$
⟨*proof*⟩

lemma $View_h$-*transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
($rel\text{-}blindable_h$ (*rel-prod* (=) (*list-all2* $pcr\text{-}view_h$)) ===> $pcr\text{-}view_h$) $Tree_h$ $View_h$
⟨*proof*⟩

type-synonym $view_m{}'$ = ($view\text{-}content_m$, $view\text{-}content_h$) $rose\text{-}tree_m$

primrec $from\text{-}view_m$ :: $view_m$ ⇒ $view_m{}'$ **where**
$from\text{-}view_m$ ($View_m$ *x*) = $Tree_m$ ($map\text{-}blindable_m$ (*map-prod id* (*map* $from\text{-}view_m$)) (*map-prod id* (*map* $from\text{-}view_h$)) *x*)

primrec $to\text{-}view_m$ :: $view_m{}'$ ⇒ $view_m$ **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*$_m$ [*simp*]: *from-view*$_m$ (*to-view*$_m$ *x*) = *x*
⟨*proof*⟩

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

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

setup-lifting *iso-view*$_m$

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

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

end

code-datatype *View*$_h$
code-datatype *View*$_m$

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

abbreviation (*input*) *hash-view-content* :: (*view-content*$_m$, *view-content*$_h$) *hash*
where
hash-view-content ≡ *hash-prod hash-view-metadata hash-view-data*

abbreviation (*input*) *blinding-of-view-content* :: *view-content*$_m$ *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*$_m$ *merge* **where**
merge-view-content ≡ *merge-prod merge-view-metadata merge-view-data*

lift-definition *hash-view* :: (*view*$_m$, *view*$_h$) *hash* **is**
hash-tree hash-view-content ⟨*proof*⟩

lift-definition *blinding-of-view* :: *view*$_m$ *blinding-of* **is**
blinding-of-tree hash-view-content blinding-of-view-content ⟨*proof*⟩

lift-definition *merge-view* :: *view*$_m$ *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*) ⟷
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 ⋀*x y*. *blinding-of-blindable* (*hash-prod hash-view-content* (*hash-list hash-view*))
(*blinding-of-prod blinding-of-view-content* (*blinding-of-list* (λ*x y*. *blinding-of-view x y* ∧ *P x y*))) *x y*
⟹ *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*) *blindable*$_m$

type-synonym *participant-metadata*$_h$ = *participant-metadata blindable*$_h$
type-synonym *participant-metadata*$_m$ = (*participant-metadata*, *participant-metadata*) *blindable*$_m$

datatype *transaction*$_h$ = *Transaction*$_h$
(*the-Transaction*$_h$: ((*common-metadata*$_h$ ×$_h$ *participant-metadata*$_h$) ×$_h$ *view*$_h$ *list*$_h$) *blindable*$_h$)

datatype *transaction*$_m$ = *Transaction*$_m$
(*the-Transaction*$_m$: ((*common-metadata*$_m$ ×$_m$ *participant-metadata*$_m$) ×$_m$ *view*$_m$

$list_m$,
(($common\text{-}metadata_h$ $\times_h$ $participant\text{-}metadata_h$) $\times_h$ $view_h$ $list_h$) $blindable_m$)

abbreviation (*input*) *hash-common-metadata* :: ($common\text{-}metadata_m$, $common\text{-}metadata_h$) *hash* **where**
hash-common-metadata ≡ *hash-blindable id*
abbreviation (*input*) *blinding-of-common-metadata* :: $common\text{-}metadata_m$ *blinding-of* **where**
blinding-of-common-metadata ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-common-metadata* :: $common\text{-}metadata_m$ *merge* **where**
merge-common-metadata ≡ *merge-blindable id merge-discrete*

abbreviation (*input*) *hash-participant-metadata* :: ($participant\text{-}metadata_m$, $participant\text{-}metadata_h$) *hash* **where**
hash-participant-metadata ≡ *hash-blindable id*
abbreviation (*input*) *blinding-of-participant-metadata* :: $participant\text{-}metadata_m$ *blinding-of* **where**
blinding-of-participant-metadata ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-participant-metadata* :: $participant\text{-}metadata_m$ *merge* **where**
merge-participant-metadata ≡ *merge-blindable id merge-discrete*

locale *transaction-merkle* **begin**

lemma $iso\text{-}transaction_h$: *type-definition the-*$Transaction_h$ $Transaction_h$ *UNIV*
⟨*proof*⟩

setup-lifting $iso\text{-}transaction_h$

lemma $Transaction_h$*-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((=) ===> $pcr\text{-}transaction_h$) *id* $Transaction_h$
⟨*proof*⟩

lemma $iso\text{-}transaction_m$: *type-definition the-*$Transaction_m$ $Transaction_m$ *UNIV*
⟨*proof*⟩

setup-lifting $iso\text{-}transaction_m$

lemma $Transaction_m$*-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((=) ===> $pcr\text{-}transaction_m$) *id* $Transaction_m$
⟨*proof*⟩

end

code-datatype $Transaction_h$
code-datatype $Transaction_m$

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

lift-definition *hash-transaction* :: $(transaction_m,\ transaction_h)$ *hash* **is**
hash-blindable (*hash-prod* (*hash-prod hash-common-metadata hash-participant-metadata*) (*hash-list hash-view*)) ⟨*proof*⟩

lift-definition *blinding-of-transaction* :: $transaction_m$ *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* :: $transaction_m$ *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*$_m$
× *view-data*$_m$ **where**
embed-view-content ≡ *map-prod Unblinded Unblinded*

lift-definition *embed-view* :: *view* ⇒ *view*$_m$ **is** *embed-source-tree embed-view-content*
⟨*proof*⟩

lemma *embed-view-simps* [*simp*]:
embed-view (*View vm vd vs*) = *View*$_m$ (*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*

⇒ *transaction* **where**
  Transaction′ = (λ((*cm*, *pm*), *views*). *Transaction cm pm views*)

lemma *Transaction-Transaction′*: *Transaction* = (λ*cm pm views*. *Transaction′* ((*cm*, *pm*), *views*))
  ⟨*proof*⟩

lemma *the-Transaction-inverse* [*simp*]: *Transaction′* (*the-Transaction x*) = *x*
  ⟨*proof*⟩

lemma *Transaction′-inverse* [*simp*]: *the-Transaction* (*Transaction′ x*) = *x*
  ⟨*proof*⟩

lemma *iso-transaction*: *type-definition the-Transaction Transaction′ UNIV*
  ⟨*proof*⟩

setup-lifting *iso-transaction*

lemma *Transaction′-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
  ((=) ===> *pcr-transaction*) *id Transaction′*
  ⟨*proof*⟩

end

code-datatype *Transaction*

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

lift-definition *embed-transaction* :: *transaction* ⇒ *transaction*$_m$ **is**
  Unblinded ∘ *map-prod* (*map-prod Unblinded Unblinded*) (*map embed-view*) ⟨*proof*⟩

lemma *embed-transaction-simps* [*simp*]:
  embed-transaction (*Transaction cm pm views*) =
    Transaction$_m$ (*Unblinded* ((*Unblinded cm*, *Unblinded pm*), *map embed-view views*))
  for *views* ⟨*proof*⟩

end

4.3.1 Inclusion proof for the mediator

primrec *mediator-view* :: *view* ⇒ *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* ⇒ *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*$_m$ = (*view-metadata*$_m$ ×$_m$ *view-data*$_m$) × *view*$_m$ *list*$_m$ × *view*$_m$ *list*$_m$
type-synonym *view-path*$_m$ = *view-path-elem*$_m$ *list*
type-synonym *view-zipper*$_m$ = *view-path*$_m$ × *view*$_m$

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

lemma *view-of-zipper*$_m$*-Nil* [*simp*]: *view-of-zipper*$_m$ ([], *t*) = *t*
⟨*proof*⟩

lift-definition *blind-view-path-elem* :: *view-path-elem* ⇒ *view-path-elem*$_m$ **is**
blind-path-elem embed-view-content hash-view-content ⟨*proof*⟩

lift-definition *blind-view-path* :: *view-path* ⇒ *view-path*$_m$ **is**
blind-path embed-view-content hash-view-content ⟨*proof*⟩

lift-definition *embed-view-path-elem* :: *view-path-elem* ⇒ *view-path-elem*$_m$ **is**
embed-path-elem embed-view-content ⟨*proof*⟩

lift-definition *embed-view-path* :: *view-path* ⇒ *view-path*$_m$ **is**
embed-path embed-view-content ⟨*proof*⟩

lift-definition *hash-view-path-elem* :: *view-path-elem*$_m$ ⇒ (*view-content*$_h$ × *view*$_h$ *list* × *view*$_h$ *list*) **is**
hash-path-elem hash-view-content ⟨*proof*⟩

lift-definition *zippers-view* :: *view-zipper* ⇒ *view-zipper*$_m$ *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*$_m$ *z*) = *hash-view* (*view-of-zipper*$_m$ (*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*$_m$ *z*) (*view-of-zipper*$_m$ (*blind-view-path p*, *embed-view t*))
⟨*proof*⟩

end

primrec *blind-view* :: *view* ⇒ *view*$_m$ **where**
blind-view (*View vm vd subviews*) =
View$_m$ (*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*$_m$ **where**
blind-transaction (*Transaction cm pm views*) =
Transaction$_m$ (*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* :: $view_m \Rightarrow participant\ set$ **where**
view-recipients (*View*$_m$ (*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* ⇒ $view_m$ *list* **where**
view-trees-for view =
map view-of-zipper$_m$
(*filter* (λ(-, *t*). *participant* ∈ *view-recipients t*)
(*zippers-view* ([], *view*)))

primrec *transaction-views-for* :: *transaction* ⇒ $transaction_m$ *list* **where**
transaction-views-for (*Transaction cm pm views*) =
map (λ$view_m$. *Transaction*$_m$ (*Unblinded* ((*Unblinded cm*, *Unblinded pm*), $view_m$)))
(*concat* (*map* (λ(*l*, *v*, *r*). *map* (λv_m. *map blind-view l* @ [v_m] @ *map 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_m ∈ *set* (*transaction-views-for t*) ⟹ *hash-transaction* t_m = *hash-transaction* (*embed-transaction t*)
⟨*proof*⟩

definition *transaction-projection-for* :: *transaction* ⇒ $transaction_m$ **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_m ∈ *set* (*transaction-views-for t*)
shows *blinding-of-transaction* t_m (*transaction-projection-for t*)
⟨*proof*⟩

end

end