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*

attribute-setup *locale-witness* = ‹*Scan.succeed Locale.witness-add*›

lemma *vimage2p-mono'*: $R \le S \Longrightarrow$ *vimage2p f g R* $\le$ *vimage2p f g S*
by(*auto simp add*: *vimage2p-def le-fun-def*)

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*)
by(*simp add*: *vimage2p-def prod.rel-map*)

lemma *vimage2p-map-list-all2*:
vimage2p (*map f*) (*map g*) (*list-all2 A*) = *list-all2* (*vimage2p f g A*)

by(*simp add*: *vimage2p-def list.rel-map*)

lemma *equivclp-least*:
assumes *le*: $r \leq s$ **and** *s*: *equivp s*
shows *equivclp* $r \leq s$
apply(*rule predicate2I*)
subgoal by(*induction rule*: *equivclp-induct*)(*auto 4 3 intro*: *equivp-reflp*[*OF s*] *equivp-transp*[*OF s*] *equivp-symp*[*OF s*] *le*[*THEN predicate2D*])
done

lemma *reflp-eq-onp*: *reflp* $R \longleftrightarrow$ *eq-onp* ($\lambda x.$ *True*) $\leq R$
by(*auto simp add*: *reflp-def eq-onp-def*)

lemma *eq-onpE*:
assumes *eq-onp P x y*
obtains $x = y$ *P y*
using *assms* **by**(*auto simp add*: *eq-onp-def*)

lemma *case-unit-parametric* [*transfer-rule*]: *rel-fun A* (*rel-fun* (=) *A*) *case-unit case-unit*
by(*simp add*: *rel-fun-def split*: *unit.split*)

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

lemma *reflp*: *reflp bo*
unfolding *bo-def* **by**(*rule reflpI*)(*simp add*: *idem*)

lemma *antisymp*: *antisymp bo*

unfolding *bo-def* **by**(*rule antisympI*)(*simp add*: *commute*)

lemma *transp*: *transp bo*
apply(*rule transpI*)
subgoal for *x y z* **using** *assoc*[*of x y z*] **by**(*simp add*: *commute bo-def*)
done

lemma *hash*: *bo* $\leq$ *vimage2p h h* (=)
unfolding *bo-def* **by**(*auto simp add*: *vimage2p-def merge-respects-hashes*)

lemma *join*: $m\ a\ b = Some\ ab \longleftrightarrow bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u)$
unfolding *bo-def*
by (*smt* (*verit*) *Option.bind-cong bind.bind-lunit commute idem merkle-interface.assoc merkle-interface-axioms*)

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(*rule antisym*)
show *?lhs* $\leq$ *?rhs* **by**(*rule equivclp-least*[*OF hash*])(*rule equivp-vimage2p*[*OF identity-equivp*])
show *?rhs* $\leq$ *?lhs* **unfolding** *vimage2p-def*
proof(*rule predicate2I*)
fix *x y*
assume $h\ x = h\ y$
then obtain *xy* **where** $m\ x\ y = Some\ xy$ **unfolding** *merge-respects-hashes* **..**
hence *bo x xy bo y xy* **unfolding** *join* **by** *blast*+
hence *equivclp bo x xy equivclp bo xy y* **by**(*blast*)+
thus *equivclp bo x y* **by**(*rule equivclp-trans*)
qed
qed

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*: $bo\ x\ y \Longrightarrow h\ x = h\ y$
  by(*drule hash*[*THEN predicate2D*])(*simp add*: *vimage2p-def*)

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*: $x \in A \Longrightarrow bo\ x\ x$
    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* $(\lambda x.\ x \in A) \leq bo$
  by(*auto elim*!: *eq-onpE intro*: *refl*)

lemma *blinding-respects-hashes*: *blinding-respects-hashes h bo* **..**
lemmas *hash* = *hash*

lemma *trans-pointfree*: *eq-onp* $(\lambda x.\ x \in A)$ *OO bo OO bo* $\leq bo$
  by(*auto elim*!: *eq-onpE intro*: *trans*)

lemma *antisym-pointfree*: *inf* (*eq-onp* $(\lambda x.\ x \in A)$ *OO bo*) $bo^{-1-1} \leq (=)$
  by(*auto elim*!: *eq-onpE dest*: *antisym*)

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*: $\llbracket$ $h\ a = h\ b$; $a \in A$ $\rrbracket$
$\Longrightarrow \exists\, ab.\ m\ a\ b = Some\ ab \wedge bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall\, u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u)$
and *undefined*: $\llbracket$ $h\ a \neq h\ b$; $a \in A$ $\rrbracket \Longrightarrow m\ a\ b = None$
begin

lemma *same*: $a \in A \Longrightarrow m\ a\ a = Some\ a$
using *join*[*of a a*] *refl*[*of a*] **by**(*auto 4 3 intro*: *antisym*)

lemma *blinding-of-antisym-on*: *blinding-of-on UNIV h bo* **..**

lemma *transp*: *transp bo*
by(*auto intro*: *transpI trans*)

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

lemma *respects-hashes*:
$a \in A \Longrightarrow h\ a = h\ b \longleftrightarrow (\exists\, ab.\ m\ a\ b = Some\ ab)$
using *join undefined*
by *fastforce*

lemma *join'*:
$a \in A \Longrightarrow \forall\, ab.\ m\ a\ b = Some\ ab \longleftrightarrow bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall\, u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u)$
using *join undefined*
by (*metis* (*full-types*) *hash local.antisym option.distinct*(*1*) *option.sel predicate2D vimage2p-def*)

lemma *merge-on-subset*:
$B \subseteq A \Longrightarrow$ *merge-on B h bo m*
by *unfold-locales* (*auto dest*: *same join undefined*)

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
show *?rhs* **if** *?lhs*
proof
interpret *merkle-interface h bo m* **by**(*fact that*)
show $bo \leq$ *vimage2p h h* (=) **by**(*fact hash*)
show *bo x x* **for** *x* **using** *reflp* **by**(*simp add*: *reflp-def*)
show *bo x z* **if** *bo x y bo y z* **for** *x y z* **using** *transp that* **by**(*rule transpD*)

    show $x = y$ **if** *bo x y bo y x* **for** *x y* **using** *antisymp that* **by**(*rule antisympD*)
    show $\exists\, ab.\ m\ a\ b = Some\ ab \wedge bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall\, u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u)$ **if** $h\ a = h\ b$ **for** *a b*
      using *that* **by**(*simp add*: *merge-respects-hashes join*)
    show $m\ a\ b = None$ **if** $h\ a \neq h\ b$ **for** *a b* **using** *that* **by**(*simp add*: *merge-respects-hashes*)
  qed

  show *?lhs* **if** *?rhs*
  proof
    interpret *merge-on UNIV h bo m* **by**(*fact that*)
    show *eq*: $h\ a = h\ b \longleftrightarrow (\exists\, ab.\ m\ a\ b = Some\ ab)$ **for** *a b* **by**(*simp add*: *respects-hashes*)
    show *idem*: $m\ a\ a = Some\ a$ **for** *a* **by**(*simp add*: *same*)
    show *commute*: $m\ a\ b = m\ b\ a$ **for** *a b*
      using *join*[*of a b*] *join*[*of b a*] *undefined antisym* **by**(*cases m a b*) *force+*
    have *undefined-partitioned*: $m\ a\ c = None$ **if** $m\ a\ b = None$ $m\ b\ c = Some\ bc$ **for** *a b c bc*
      using *that eq* **by** (*metis option.distinct*(*1*) *option.exhaust*)
    have *merge-twice*: $m\ a\ b = Some\ c \Longrightarrow m\ a\ c = Some\ c$ **for** *a b c* **by** (*simp add*: *join′*)
    show $m\ a\ b \ggg m\ c = m\ b\ c \ggg m\ a$ **for** *a b c*
    proof(*simp split*: *Option.bind-split*; *safe*)
      show $None = m\ a\ d$ **if** $m\ a\ b = None$ $m\ b\ c = Some\ d$ **for** *d* **using** *that*
        by(*metis undefined-partitioned merge-twice*)
      show $m\ c\ d = None$ **if** $m\ a\ b = Some\ d$ $m\ b\ c = None$ **for** *d* **using** *that*
        by(*metis commute merge-twice undefined-partitioned*)
    next
      fix *ab bc*
      assume *assms*: $m\ a\ b = Some\ ab$ $m\ b\ c = Some\ bc$
      then obtain *cab* **and** *abc* **where** *cab*: $m\ c\ ab = Some\ cab$ **and** *abc*: $m\ a\ bc = Some\ abc$
        using *eq*[*THEN iffD2*, *OF exI*] *eq*[*THEN iffD1*] **by** (*metis merge-twice*)
      thus $m\ c\ ab = m\ a\ bc$ **using** *assms*
        by(*clarsimp simp add*: *join′*)(*metis UNIV-I abc cab local.antisym local.trans*)
    qed
    show $bo\ a\ b \longleftrightarrow m\ a\ b = Some\ b$ **for** *a b* **using** *idem join′* **by** *auto*
  qed
qed

lemma *merkle-interfaceI* [*locale-witness*]:
  assumes *merge-on UNIV h bo m*
  shows *merkle-interface h bo m*
  using *assms* **unfolding** *merkle-interface-aux* **by** *auto*

lemma (**in** *merkle-interface*) *merkle-interfaceD*: *merge-on UNIV h bo m*
  using *merkle-interface-aux*[*of h bo m*, *symmetric*]
  by *simp unfold-locales*

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*]:
$((A1 ===> A2) ===> (A1 ===> A1 ===> (\longleftrightarrow)) ===> (\longleftrightarrow))$
blinding-respects-hashes blinding-respects-hashes
if [*transfer-rule*]: *bi-unique A2 bi-total A1*
by(*rule blinding-respects-hashes-parametric-aux that le-fun-parametric* | *simp add*: *rel-fun-eq*)+

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* (=)
by(*auto simp add*: *vimage2p-def*)

lemma *blinding-of-on-discrete* [*locale-witness*]:
blinding-of-on UNIV hash-discrete blinding-of-discrete
by(*unfold-locales*)(*simp-all add*: *OO-eq eq-onp-def blinding-of-discrete-hash*)

lemma *merge-on-discrete* [*locale-witness*]:
merge-on UNIV hash-discrete blinding-of-discrete merge-discrete
by *unfold-locales*(*auto simp add*: *merge-discrete-def*)

lemma *merkle-discrete* [*locale-witness*]:
merkle-interface hash-discrete blinding-of-discrete merge-discrete
..

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* (=)
by(*fact blinding-of-discrete-hash*)

lemma *blinding-of-on-unit*:
blinding-of-on UNIV hash-unit blinding-of-unit
by(*fact blinding-of-on-discrete*)

lemma *merge-on-unit*:
merge-on UNIV hash-unit blinding-of-unit merge-unit
by(*fact merge-on-discrete*)

lemma *merkle-interface-unit*:
merkle-interface hash-unit blinding-of-unit merge-unit
by(*intro merkle-interfaceI merge-on-unit*)

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'* ∘ $map\text{-}blindable_m$ *h id*

lemma *hash-blindable-simps* [*simp*]:
hash-blindable h (*Unblinded x*) = *Content* (*h x*)
hash-blindable h (*Blinded y*) = *y*
by(*simp-all add*: *hash-blindable-def* $blindable_h$*.map-id*)

lemma *hash-map-blindable-simp*:
hash-blindable f ($map\text{-}blindable_m$ *f' id x*) = *hash-blindable* (*f o f'*) *x*
by(*cases x*) (*simp-all add*: *hash-blindable-def* $blindable_h$*.map-comp*)

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 $bo \leq bo'$
shows *blinding-of-blindable h bo* $\leq$ *blinding-of-blindable h bo'*
apply(*rule predicate2I*)
apply(*erule blinding-of-blindable.cases*; *hypsubst*)
subgoal by(*rule blinding-of-blindable.intros*)(*rule assms*[*THEN predicate2D*])
subgoal by(*rule blinding-of-blindable.intros*) *simp*
done

lemma *blinding-of-blindable-hash*:
assumes $bo \leq$ *vimage2p h h* (=)
shows *blinding-of-blindable h bo* $\leq$ *vimage2p* (*hash-blindable h*) (*hash-blindable h*) (=)
apply(*rule predicate2I vimage2pI*)+
apply(*erule blinding-of-blindable.cases*; *hypsubst*)
subgoal using *assms*[*THEN predicate2D*] **by**(*simp add*: *vimage2p-def*)
subgoal by *simp*
done

lemma *blinding-of-on-blindable* [*locale-witness*]:
assumes *blinding-of-on A h bo*
shows *blinding-of-on* $\{x.$ *set1-blindable*$_m$ $x \subseteq A\}$ (*hash-blindable h*) (*blinding-of-blindable h bo*)
(**is** *blinding-of-on ?A ?h ?bo*)
proof −
interpret *blinding-of-on A h bo* **by** *fact*
show *?thesis*
proof
show *?bo* $\leq$ *vimage2p ?h ?h* (=)
by(*rule blinding-of-blindable-hash*)(*rule hash*)
show *?bo x x* **if** $x \in$ *?A* **for** *x* **using** *that* **by**(*cases x*)(*auto simp add*: *refl*)
show *?bo x z* **if** *?bo x y ?bo y z* $x \in$ *?A* **for** *x y z* **using** *that*
by(*auto elim*!: *blinding-of-blindable.cases dest*: *trans blinding-hash-eq*)
show $x = y$ **if** *?bo x y ?bo y x* $x \in$ *?A* **for** *x y* **using** *that*
by(*auto elim*!: *blinding-of-blindable.cases dest*: *antisym*)
qed
qed

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*$_m$ $x \subseteq A\}$ (*hash-blindable h*) (*blinding-of-blindable h bo*) *merge-blindable*
(**is** *merge-on ?A ?h ?bo ?m*)
proof −
interpret *merge-on A h bo m* **by** *fact*
show *?thesis*
proof
show $\exists$ *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*) **if** *?h a* = *?h b a* $\in$ *?A* **for** *a b*
using *that* **by**(*cases* (*a*, *b*) *rule*: *merge-blindable.cases*)(*auto simp add*: *refl dest*!: *join*)
show *?m a b* = *None* **if** *?h a* $\neq$ *?h b a* $\in$ *?A* **for** *a b*
using *that* **by**(*cases* (*a*, *b*) *rule*: *merge-blindable.cases*)(*auto simp add*: *dest*!: *undefined*)
qed
qed

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*) $\Rightarrow$ *map-option Unblinded* (*m x y*)
| (*Blinded x*, *Unblinded y*) $\Rightarrow$ (*if Content* (*h y*) = *x then Some* (*Unblinded y*) *else None*)
| (*Unblinded y*, *Blinded x*) $\Rightarrow$ (*if Content* (*h y*) = *x then Some* (*Unblinded y*) *else*

None)
| (*Blinded t*, *Blinded u*) ⇒ (*if t* = *u then Some* (*Blinded u*) *else None*))
by(*simp split*: $blindable_m$*.split* $blindable_h$*.split*)

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

lemma *merge-blindable-cong* [*fundef-cong*]:
assumes $\bigwedge a\ b.\ [\![\ a \in set1\text{-}blindable_m\ x;\ b \in set1\text{-}blindable_m\ y\ ]\!] \Longrightarrow m\ a\ b = m'\ a\ b$
shows *merge-blindable h m x y* = *merge-blindable h m′ x y*
by(*auto simp add*: *merge-blindable-alt-def split*: $blindable_m$*.split intro*: *assms intro*!: *arg-cong*[**where** *f*=*map-option* -])

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 −
interpret *merge-on UNIV h bo m* **using** *assms* **by**(*simp add*: *merkle-interface-aux*)
show *?thesis* **unfolding** *merkle-interface-aux* **..**
qed

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
..

lemma *merkle-nr-blindable*:
merkle-interface hash-nr-blindable blinding-of-nr-blindable merge-nr-blindable
..

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' $\equiv$ *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*) (=)
using *assms* **by**(*auto simp add*: *vimage2p-def elim*!: *rel-sum.cases*)

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*)
proof −
interpret *a*: *blinding-of-on A rha boa* **by** *fact*
interpret *b*: *blinding-of-on B rhb bob* **by** *fact*
show *?thesis*
proof
show *?bo x x* **if** $x \in$ *?A* **for** *x* **using** *that* **by**(*intro sum.rel-refl-strong*)(*auto intro*: *a.refl b.refl*)
show *?bo x z* **if** *?bo x y ?bo y z* $x \in$ *?A* **for** *x y z*
using *that* **by**(*auto elim*!: *rel-sum.cases dest*: *a.trans b.trans*)
show $x = y$ **if** *?bo x y ?bo y x* $x \in$ *?A* **for** *x y*

using *that* **by**(*auto elim*!: *rel-sum.cases dest*: *a.antisym b.antisym*)
qed(*rule blinding-of-sum-hash a.hash b.hash*)+
qed

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 −
interpret *a*: *merge-on A rha boa ma* **by** *fact*
interpret *b*: *merge-on B rhb bob mb* **by** *fact*
show *?thesis*
proof
show $\exists\, ab.\ ?m\ a\ b = Some\ ab \wedge ?bo\ a\ ab \wedge ?bo\ b\ ab \wedge (\forall\, u.\ ?bo\ a\ u \longrightarrow ?bo\ b\ u \longrightarrow ?bo\ ab\ u)$
if $?h\ a = ?h\ b\ a \in ?A$ **for** *a b* **using** *that*
by(*cases* (*a*, *b*) *rule*: *merge-sum.cases*)(*auto dest*!: *a.join b.join elim*!: *rel-sum.cases*)
show $?m\ a\ b = None$ **if** $?h\ a \neq ?h\ b\ a \in ?A$ **for** *a b* **using** *that*
by(*cases* (*a*, *b*) *rule*: *merge-sum.cases*)(*auto dest*!: *a.undefined b.undefined*)
qed
qed

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*)
by(*simp add*: *split*: *sum.split*)

end

lemma *merge-sum-cong*[*fundef-cong*]:
  $\llbracket$ $x = x'$; $y = y'$;
    $\bigwedge xl\ yl.$ $\llbracket$ $x = Inl\ xl$; $y = Inl\ yl$ $\rrbracket \Longrightarrow ma\ xl\ yl = ma'\ xl\ yl$;
    $\bigwedge xr\ yr.$ $\llbracket$ $x = Inr\ xr$; $y = Inr\ yr$ $\rrbracket \Longrightarrow mb\ xr\ yr = mb'\ xr\ yr$ $\rrbracket \Longrightarrow$
    merge-sum $ma\ mb\ x\ y$ = *merge-sum* $ma'\ mb'\ x'\ y'$
  by(*cases x*; *simp-all*; *cases y*; *auto*)

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 −
  interpret *a*: *merge-on UNIV rha boa ma* **unfolding** *merkle-interface-aux*[*symmetric*] **by** *fact*
  interpret *b*: *merge-on UNIV rhb bob mb* **unfolding** *merkle-interface-aux*[*symmetric*] **by** *fact*
  show *?thesis* **unfolding** *merkle-interface-aux*[*symmetric*] **..**
qed

2.4 Building Block: Products

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

type-synonym $('a_h, 'b_h)\ prod_h = 'a_h \times 'b_h$
type-notation $prod_h$ (‹(- $\times_h$/ -)› [*21*, *20*] *20*)

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

2.4.1 Hashes

abbreviation (*input*) *hash-prod'* :: $('a_h \times_h 'b_h, 'a_h \times_h 'b_h)$ *hash* **where**
  hash-prod' ≡ *id*

abbreviation (*input*) *hash-prod* :: $('a_m, 'a_h)$ *hash* ⇒ $('b_m, 'b_h)$ *hash* ⇒ $('a_m \times_m 'b_m, 'a_h \times_h 'b_h)$ *hash*
  where *hash-prod* ≡ *map-prod*

2.4.2 Blinding

abbreviation (*input*) *blinding-of-prod* :: $'a_m$ *blinding-of* ⇒ $'b_m$ *blinding-of* ⇒ $('a_m \times_m 'b_m)$ *blinding-of* **where**
  blinding-of-prod ≡ *rel-prod*

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

lemma *blinding-of-prod-hash*:
assumes $boa \leq vimage2p\ rha\ rha\ (=)$ $bob \leq vimage2p\ rhb\ rhb\ (=)$
shows $blinding\text{-}of\text{-}prod\ boa\ bob \leq vimage2p\ (hash\text{-}prod\ rha\ rhb)\ (hash\text{-}prod\ rha\ rhb)\ (=)$
using *assms* **by**(*auto simp add*: *vimage2p-def*)

lemma *blinding-of-on-prod* [*locale-witness*]:
assumes *blinding-of-on A rha boa blinding-of-on B rhb bob*
shows $blinding\text{-}of\text{-}on\ \{x.\ fsts\ x \subseteq A \land snds\ x \subseteq B\}\ (hash\text{-}prod\ rha\ rhb)\ (blinding\text{-}of\text{-}prod\ boa\ bob)$
(**is** *blinding-of-on ?A ?h ?bo*)
proof −
interpret *a*: *blinding-of-on A rha boa* **by** *fact*
interpret *b*: *blinding-of-on B rhb bob* **by** *fact*
show *?thesis*
proof
show $?bo\ x\ x$ **if** $x \in ?A$ **for** *x* **using** *that* **by**(*cases x*)(*auto intro*: *a.refl b.refl*)
show $?bo\ x\ z$ **if** $?bo\ x\ y$ $?bo\ y\ z$ $x \in ?A$ **for** *x y z* **using** *that*
by(*auto elim*!: *rel-prod.cases dest*: *a.trans b.trans*)
show $x = y$ **if** $?bo\ x\ y$ $?bo\ y\ x$ $x \in ?A$ **for** *x y* **using** *that*
by(*auto elim*!: *rel-prod.cases dest*: *a.antisym b.antisym*)
qed(*rule blinding-of-prod-hash a.hash b.hash*)+
qed

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 $merge\text{-}prod :: ('a_m \times_m 'b_m)\ merge$ **where**
$merge\text{-}prod\ (x,\ y)\ (x',\ y') = Option.bind\ (ma\ x\ x')\ (\lambda x''.\ map\text{-}option\ (Pair\ x'')\ (mb\ y\ y'))$

lemma *merge-on-prod* [*locale-witness*]:
assumes *merge-on A rha boa ma merge-on B rhb bob mb*
shows $merge\text{-}on\ \{x.\ fsts\ x \subseteq A \land snds\ x \subseteq B\}\ (hash\text{-}prod\ rha\ rhb)\ (blinding\text{-}of\text{-}prod\ boa\ bob)\ merge\text{-}prod$
(**is** *merge-on ?A ?h ?bo ?m*)
proof −
interpret *a*: *merge-on A rha boa ma* **by** *fact*
interpret *b*: *merge-on B rhb bob mb* **by** *fact*

show *?thesis*
proof
show $\exists\, ab.$ *?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*$)$
if *?h a* $=$ *?h b a* $\in$ *?A* **for** *a b* **using** *that*
by(*cases* (*a*, *b*) *rule*: *merge-prod.cases*)(*auto dest*!: *a.join b.join*)
show *?m a b* $=$ *None* **if** *?h a* $\neq$ *?h b a* $\in$ *?A* **for** *a b* **using** *that*
by(*cases* (*a*, *b*) *rule*: *merge-prod.cases*)(*auto dest*!: *a.undefined b.undefined*)
qed
qed

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

lemma *merge-prod-alt-def*:
merge-prod $=$ $(\lambda(x,\ y)\ (x',\ y').$ *Option.bind* $(ma\ x\ x')$ $(\lambda x''.$ *map-option* $(Pair\ x'')$ $(mb\ y\ y')))$
by(*simp add*: *fun-eq-iff*)

end

lemma *merge-prod-cong*[*fundef-cong*]:
assumes $\bigwedge a\ b.\ [\![\ a \in fsts\ p1;\ b \in fsts\ p2\ ]\!] \Longrightarrow ma\ a\ b = ma'\ a\ b$
and $\bigwedge a\ b.\ [\![\ a \in snds\ p1;\ b \in snds\ p2\ ]\!] \Longrightarrow mb\ a\ b = mb'\ a\ b$
shows *merge-prod ma mb p1 p2* $=$ *merge-prod ma' mb' p1 p2*
using *assms* **by**(*cases p1*; *cases p2*) *auto*

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 −
interpret *a*: *merge-on UNIV rha boa ma* **unfolding** *merkle-interface-aux*[*symmetric*] **by** *fact*
interpret *b*: *merge-on UNIV rhb bob mb* **unfolding** *merkle-interface-aux*[*symmetric*] **by** *fact*
show *?thesis* **unfolding** *merkle-interface-aux*[*symmetric*] **..**
qed

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* $+$ $'a \times 'b$

abbreviation (*input*) *set-base-F*$_m$ $\equiv \lambda x.$ *setr* $x \gg=$ *fsts*
abbreviation (*input*) *set-rec-F*$_m$ $\equiv \lambda A.$ *setr* $A \gg=$ *snds*
abbreviation (*input*) *map-F* $\equiv \lambda$*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* $F = l \longleftrightarrow F =$ *unR* l
  by *auto*

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

lemma *list-R1-induct*[*case-names list-R1*]:
  assumes $\bigwedge F.$ $[\![\bigwedge l'.\ l' \in$ *set-rec-F*$_m$ $F \implies P\ l']\!] \implies P$ (*list-R1* F)
  shows $P\ l$
  apply(*rule list-R1.induct*)
  apply(*auto intro*!: *assms*)
  done

lemma *set-list-R1-eq*:
  $\{x.$ *set-base-F*$_m$ $x \subseteq A \wedge$ *set-rec-F*$_m$ $x \subseteq B\} =$
   $\{x.$ *setl* $x \subseteq$ *UNIV* $\wedge$ *setr* $x \subseteq \{x.$ *fsts* $x \subseteq A \wedge$ *snds* $x \subseteq B\}\}$
  by(*auto simp add*: *bind-UNION*)

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*
  by(*simp-all split*: *unit.split*)

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) $= x$
  by(*induct* x) (*auto*)

lemma *list-of-R1*: *list-to-list-R1* (*list-R1-to-list* x) $= x$
  apply(*induct* x)
  subgoal for x

by(*cases x*) (*auto*)
done

lemma *list-R1-def*: *type-definition list-to-list-R1 list-R1-to-list UNIV*
by(*unfold-locales*)(*auto intro*: *R1-of-list list-of-R1*)

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*)
by(*induction xs*) *auto*

lemma *list-R1-map-trans* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(((=) ===> (=)) ===> *pcr-list* (=) ===> *pcr-list* (=)) *map-list-R1 map*
by(*auto 4 3 simp add*: *list.pcr-cr-eq rel-fun-eq cr-list-def map-list-R1-list-to-list-R1*)

lemma *set-list-R1-list-to-list-R1*: *set-list-R1* (*list-to-list-R1 xs*) = *set xs*
by(*induction xs*) *auto*

lemma *list-R1-set-trans* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*pcr-list* (=) ===> (=)) *set-list-R1 set*
by(*auto simp add*: *list.pcr-cr-eq cr-list-def set-list-R1-list-to-list-R1*)

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
define *xs'* **and** *ys'* **where** *xs'* = *list-to-list-R1 xs* **and** *ys'* = *list-to-list-R1 ys*
assume *rel-list-R1 R xs' ys'*
then have *list-all2 R* (*list-R1-to-list xs'*) (*list-R1-to-list ys'*)
by *induction*(*auto elim*!: *rel-sum.cases*)
thus *?rhs* **by**(*simp add*: *xs'-def ys'-def R1-of-list*)
next
show *?lhs* **if** *?rhs* **using** *that* **by** *induction auto*
qed

lemma *list-R1-rel-trans*[*transfer-rule*]: **includes** *lifting-syntax* **shows**
(((=) ===> (=) ===> (=)) ===> *pcr-list* (=) ===> *pcr-list* (=) ===> (=)) *rel-list-R1 list-all2*
by(*auto 4 4 simp add*: *list.pcr-cr-eq rel-fun-eq cr-list-def rel-list-R1-list-to-list-R1*)

2.5.2 Hashes

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

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

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*) (=)
apply(*rule predicate2I vimage2pI*)+
apply(*auto simp add*: *predicate2D-vimage2p*[*OF assms*] *elim*!: *list-R1.rel-induct rel-sum.cases rel-prod.cases*)
done

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 −
interpret *a*: *blinding-of-on A h bo* **by** *fact*
show *?thesis*
proof
show *hash*: *?bo* ≤ *vimage2p ?h ?h* (=) **using** *a.hash* **by**(*rule blinding-of-hash-R1*)

have *?bo x x* ∧ (*?bo x y* ⟶ *?bo y z* ⟶ *?bo x z*) ∧ (*?bo x y* ⟶ *?bo y x* ⟶ *x* = *y*) **if** *x* ∈ *?A* **for** *x y z* **using** *that*
proof(*induction x arbitrary*: *y z*)
case (*list-R1 x y' z'*)
from *list-R1.prems* **have** *s1*: *set-base-F$_m$* *x* ⊆ *A* **by**(*fastforce*)

from *list-R1.prems* **have** *s3*: *set-rec-*F_m *x* ⪢= *set-list-R1* ⊆ *A* **by**(*fastforce intro*: *rev-bexI*)

interpret *F*: *blinding-of-on* {*y*. *set-base-*F_m *y* ⊆ *A* ∧ *set-rec-*F_m *y* ⊆ *set-rec-*F_m *x*}
hash-F h (*hash-R1 h*) *blinding-of-F bo* (*blinding-of-R1 bo*)
unfolding *hash-F-def blinding-of-F-def set-list-R1-eq*
proof
let *?A′* = *setr x* ⪢= *snds* **and** *?bo′* = *rel-list-R1 bo*
show *?bo′ x x* **if** *x* ∈ *?A′* **for** *x* **using** *that list-R1* **by**(*force simp add*: *eq-onp-def*)
show *?bo′ x z* **if** *?bo′ x y ?bo′ y z x* ∈ *?A′* **for** *x y z*
using *that list-R1.IH*[*of - x y z*] *list-R1.prems*
by(*force simp add*: *bind-UNION prod-set-defs*)
show *x* = *y* **if** *?bo′ x y ?bo′ y x x* ∈ *?A′* **for** *x y*
using *that list-R1.IH*[*of - x y*] *list-R1.prems*
by(*force simp add*: *prod-set-defs*)
qed(*rule hash*)
show *?case* **using** *list-R1.prems*
apply(*intro conjI*)
subgoal using *F.refl*[*of x*] *s1* **unfolding** *blinding-of-F-def* **by**(*auto intro*: *list-R1.rel-intros*)
subgoal using *s1* **by**(*auto elim*!: *list-R1.rel-cases F.trans*[*unfolded blinding-of-F-def*] *intro*: *list-R1.rel-intros*)
subgoal using *s1* **by**(*auto elim*!: *list-R1.rel-cases dest*: *F.antisym*[*unfolded blinding-of-F-def*])
done
qed
then show *x* ∈ *?A* ⟹ *?bo x x*
and ⟦ *?bo x y*; *?bo y z*; *x* ∈ *?A* ⟧ ⟹ *?bo x z*
and ⟦ *?bo x y*; *?bo y x*; *x* ∈ *?A* ⟧ ⟹ *x* = *y*
for *x y z* **by** *blast+*
qed
qed

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*
using *assms*
apply(*cases x*; *cases y*)
apply(*simp-all add*: *merge-F-def*)
apply(*rule arg-cong*[**where** *f*=*map-option* -])
apply(*blast intro*: *merge-prod-cong*)
done

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 −
interpret *a*: *merge-on A h bo m* **by** *fact*
show *?thesis*
proof
have (*?h a* = *?h b* ⟶ (∃ *ab. ?m a b* = *Some ab* ∧ *?bo a ab* ∧ *?bo b ab* ∧ (∀ *u. ?bo a u* ⟶ *?bo b u* ⟶ *?bo ab u*))) ∧
(*?h a* ≠ *?h b* ⟶ *?m a b* = *None*)
if *a* ∈ *?A* **for** *a b* **using** *that* **unfolding** *mem-Collect-eq*
proof(*induction a arbitrary*: *b rule*: *list-R1-induct*)
case *wfInd*: (*list-R1 l*)
interpret *merge-on* {*y. set-base-F*$_m$ *y* ⊆ *A* ∧ *set-rec-F*$_m$ *y* ⊆ *set-rec-F*$_m$ *l*}
hash-F h ?h blinding-of-F bo ?bo merge-F m ?m
unfolding *set-list-R1-eq hash-F-def merge-F-def blinding-of-F-def*
proof
fix *a*
assume *a*: *a* ∈ *set-rec-F*$_m$ *l*
with *wfInd.prems* **have** *a′*: *set-list-R1 a* ⊆ *A*
by *fastforce*

show *hash-R1 h a* = *hash-R1 h b*
⟹ ∃ *ab. ?m a b* = *Some ab* ∧ *?bo a ab* ∧ *?bo b ab* ∧
(∀ *u. ?bo a u* ⟶ *?bo b u* ⟶ *?bo ab u*)
and *?h a* ≠ *?h b* ⟹ *?m a b* = *None* **for** *b*
using *wfInd.IH*[*OF a a′*, *rule-format*, *of b*]
by(*auto dest*: *sym*)

qed
show *?case* **using** *wfInd.prems*
apply(*intro conjI strip*)
subgoal
by(*auto 4 4 dest*!: *join*[*unfolded hash-F-def*]
simp add: *blinding-of-F-def UN-subset-iff list-R1.rel-sel*)
subgoal **by**(*auto 4 3 intro*!: *undefined*[*simplified hash-F-def*])
done
qed
then show
$?h\ a = ?h\ b \Longrightarrow \exists\, ab.\ ?m\ a\ b = Some\ ab \wedge ?bo\ a\ ab \wedge ?bo\ b\ ab \wedge (\forall\, u.\ ?bo\ a\ u \longrightarrow ?bo\ b\ u \longrightarrow ?bo\ ab\ u)$
$?h\ a \neq ?h\ b \Longrightarrow ?m\ a\ b = None$
if $a \in ?A$ **for** *a b* **using** *that* **by** *blast+*
qed
qed

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 −
interpret *merge-on UNIV h bo m* **using** *assms* **by**(*unfold merkle-interface-aux*)
show *?thesis* **unfolding** *merkle-interface-aux*[*symmetric*] **..**
qed

lemma *merge-R1-cong* [*fundef-cong*]:
assumes $\bigwedge a\ b.\ [\![\ a \in set\text{-}list\text{-}R1\ x;\ b \in set\text{-}list\text{-}R1\ y\ ]\!] \Longrightarrow m\ a\ b = m'\ a\ b$
shows *merge-R1 m x y* = *merge-R1 m′ x y*
using *assms*
apply(*induction x y rule*: *merge-R1.induct*)
apply(*simp del*: *merge-cases*)
apply(*rule arg-cong*[**where** *f*=*map-option* -])
apply(*blast intro*: *merge-F-cong*[*unfolded bind-UNION*])
done

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
apply(*intro rel-funI*)
subgoal premises *prems* [*transfer-rule*] **for** *m1 m2 xs1 xs2 ys1 ys2* **using** *prems*(*2*, *3*)

apply(*induction xs1 ys1 arbitrary*: *xs2 ys2 rule*: *merge-R1.induct*)
apply(*elim list-R1.rel-cases rel-sum.cases*; *clarsimp simp add*: *option.rel-map merge-F-def merge-discrete-def*)
apply(*elim meta-allE*; (*erule meta-impE*, *simp*)+)
subgoal premises [*transfer-rule*] **by** *transfer-prover*
done
done

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* **.**

abbreviation (*input*) *hash-list* :: ($'a_m$, $'a_h$) *hash* ⇒ ($'a_m$ $list_m$, $'a_h$ $list_h$) *hash*
where *hash-list* ≡ *map*
abbreviation (*input*) *blinding-of-list* :: $'a_m$ *blinding-of* ⇒ $'a_m$ $list_m$ *blinding-of*
where *blinding-of-list* ≡ *list-all2*
lift-definition *merge-list* :: $'a_m$ *merge* ⇒ $'a_m$ $list_m$ *merge* **is** *merge-R1* **.**

lemma *blinding-of-list-mono*:
$\llbracket \bigwedge x\ y.\ bo\ x\ y \longrightarrow bo'\ x\ y \rrbracket \Longrightarrow$
blinding-of-list bo x y ⟶ *blinding-of-list bo′ x y*
by (*transfer*) (*blast intro*: *list-R1.rel-mono-strong*)

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* $R \leq$ *blinding-of-list* R'
by(*transfer*) (*rule list-R1.rel-mono*)

lemma *blinding-of-list-simp*: *blinding-of-list* = *list-all2*
by(*transfer*) (*rule refl*)

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*)
by(*transfer fixing*: *h bo m*) *unfold-locales*

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** ‹$\Rightarrow_h$› *0*)

type-synonym $('a, 'b_m)$ fun_m $=$ $'a \Rightarrow 'b_m$
type-notation fun_m (**infixr** ‹$\Rightarrow_m$› *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*) (=)
using *assms* **by**(*auto simp add*: *vimage2p-def rel-fun-def le-fun-def*)

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 −
interpret *a*: *blinding-of-on A rh bo* **by** *fact*
show *?thesis*
proof
show *?bo x x* **if** *x* ∈ *?A* **for** *x* **using** *that* **by**(*auto simp add*: *rel-fun-def intro*: *a.refl*)
show *?bo x z* **if** *?bo x y ?bo y z x* ∈ *?A* **for** *x y z* **using** *that*

by(*auto 4 3 simp add*: *rel-fun-def intro*: *a.trans*)
show $x = y$ **if** *?bo x y ?bo y x x* ∈ *?A* **for** *x y* **using** *that*
by(*fastforce simp add*: *fun-eq-iff rel-fun-def intro*: *a.antisym*)
qed(*rule blinding-of-fun-hash a.hash*)+
qed

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* $\Rightarrow_m$ *'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 −
interpret *a*: *merge-on A rh bo m* **by** *fact*
show *?thesis*
proof
show ∃ *ab*. *?m a b* = *Some ab* ∧ *?bo a ab* ∧ *?bo b ab* ∧ (∀ *u*. *?bo a u* ⟶ *?bo b u* ⟶ *?bo ab u*)
if *?h a* = *?h b a* ∈ *?A* **for** *a b*
using *that*(*1*)[*THEN fun-cong*, *unfolded o-apply*, *THEN a.join*, *OF that*(*2*)[*unfolded mem-Collect-eq*, *THEN subsetD*, *OF rangeI*]]
by *atomize*(*subst* (*asm*) *choice-iff*; *auto simp add*: *merge-fun-def rel-fun-def*)
show *?m a b* = *None* **if** *?h a* ≠ *?h b a* ∈ *?A* **for** *a b* **using** *that*
by(*auto simp add*: *merge-fun-def fun-eq-iff dest*: *a.undefined*)
qed
qed

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*
using *assms*[*OF rangeI rangeI*] **by**(*clarsimp simp add*: *merge-fun-def*)

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

by(*auto simp add*: *Option.is-none-def split*: *option.splits*)

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(*intro rel-funI*)
fix m :: $'a$ *merge* **and** m' :: $'b$ *merge* **and** f :: $'c \Rightarrow 'a$ **and** f' :: $'c \Rightarrow 'b$ **and** g :: $'c \Rightarrow 'a$ **and** g' :: $'c \Rightarrow 'b$
assume *m*: (*A* ===> *B* ===> *rel-option C*) m m'
and *f*: ((=) ===> *A*) f f' **and** *g*: ((=) ===> *B*) g g'
note [*transfer-rule*] = *this*
have *cond* [*unfolded Option.is-none-def*]: ($\forall x. \neg$ *Option.is-none* (m (f x) (g x))) $\longleftrightarrow$ ($\forall x. \neg$ *Option.is-none* (m' (f' x) (g' x)))
by *transfer-prover*
moreover
have ((=) ===> *C*) ($\lambda x.$ *the* (m (f x) (g x))) ($\lambda x.$ *the* (m' (f' x) (g' x))) **if** $*$: $\forall x. \neg$ m (f x) (g x) = *None*
proof −
obtain *fg* fg' **where** *m*: m (f x) (g x) = *Some* (*fg* x) **and** m': m' (f' x) (g' x) = *Some* (fg' x) **for** x
using $*$ $*$[*simplified cond*]
by(*simp*)(*subst* (*asm*) (*1 2*) *choice-iff*; *clarsimp*)
have *rel-option C* (*Some* (*fg* x)) (*Some* (fg' x)) **for** x **unfolding** *m*[*symmetric*] m'[*symmetric*] **by** *transfer-prover*
then show *?thesis* **by**(*simp add*: *rel-fun-def m* m')
qed
ultimately show *rel-option* ((=) ===> *C*) (*merge-fun m f g*) (*merge-fun* m' f' g')
unfolding *merge-fun-def* **by**(*simp*)
qed

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 −
interpret *a*: *merge-on UNIV rh bo m* **unfolding** *merkle-interface-aux*[*symmetric*] **by** *fact*
show *?thesis* **unfolding** *merkle-interface-aux*[*symmetric*] **..**
qed

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*))
by(*simp add*: *hash-rt-F*$_m$*-def fun-eq-iff hash-map-blindable-simp*)

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* $\Rightarrow$ $(('a_m, 'a_h)$ *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'*)
by(*rule ext*) (*simp add*: *blindable*$_m$*.map-comp*)

lemma *hash-tree-simps* [*simp*]:
hash-tree h (*Tree*$_m$ *x*) = *Tree*$_h$ (*hash-rt-F*$_m$ *h* (*hash-tree h*) *x*)
by(*simp add*: *hash-tree-def hash-rt-F*$_m$*-def*
map-prod.comp map-sum.comp rose-tree$_h$*.map-comp blindable*$_m$*.map-comp*
prod.map-id0 rose-tree$_h$*.map-id0*)

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

lemma *blinding-of-rt-F*$_m$*-mono*:
$[\![$ *boa* $\leq$ *boa'*; *bob* $\leq$ *bob'* $]\!]$ $\Longrightarrow$ *blinding-of-rt-F*$_m$ *ha boa hb bob* $\leq$ *blinding-of-rt-F*$_m$ *ha boa' hb bob'*
by(*intro blinding-of-blindable-mono prod.rel-mono list.rel-mono*)

lemma *blinding-of-rt-F*$_m$*-mono-inductive*:
assumes $\bigwedge x\ y.$ *boa x y* $\longrightarrow$ *boa' x y* $\bigwedge x\ y.$ *bob x y* $\longrightarrow$ *bob' x y*
shows *blinding-of-rt-F*$_m$ *ha boa hb bob x y* $\longrightarrow$ *blinding-of-rt-F*$_m$ *ha boa' hb bob' x y*
apply(*rule impI*)
apply(*erule blinding-of-rt-F*$_m$*-mono*[*THEN predicate2D*, *rotated* -1])
using *assms* **by** *blast*+

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*) (=)
apply(*rule order-trans*)
apply(*rule blinding-of-blindable-hash*)
apply(*fold relator-eq*)
apply(*unfold vimage2p-map-rel-prod vimage2p-map-list-all2*)
apply(*rule prod.rel-mono assms list.rel-mono*)+
apply(*simp only*: *hash-rt-*F_m*-def vimage2p-comp o-apply hash-blindable-def blindable*$_m$*.map-id0 id-def*[*symmetric*] *vimage2p-id id-apply*)
done

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*) (=)
apply(*rule predicate2I vimage2pI*)+
apply(*erule blinding-of-tree.induct*)
apply(*simp*)
apply(*erule blinding-of-rt-*F_m*-hash*[*OF assms*, *THEN predicate2D-vimage2p*, *rotated 1*])
apply(*blast intro*: *vimage2pI*)
done

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*}}}
by *force*

lemma *hash-blindable-map*: *hash-blindable f* $\circ$ *map-blindable*$_m$ *g id* = *hash-blindable* (*f* $\circ$ *g*)
by(*rule ext*) (*simp add*: *hash-blindable-def blindable*$_m$*.map-comp*)

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 −
  interpret *a*: *blinding-of-on A h bo* **by** *fact*
  show *?thesis*
  proof
    show *?bo* ≤ *vimage2p ?h ?h* (=) **using** *a.hash* **by**(*rule blinding-of-tree-hash*)
    have *?bo x x* ∧ (*?bo x y* ⟶ *?bo y z* ⟶ *?bo x z*) ∧ (*?bo x y* ⟶ *?bo y x* ⟶ *x* = *y*) **if** *x* ∈ *?A* **for** *x y z* **using** *that*
    proof(*induction x arbitrary*: *y z*)
      case ($Tree_m$ *x*)
      have [*locale-witness*]: *blinding-of-on* (*set3-rt-*F_m *x*) (*hash-tree h*) (*blinding-of-tree h bo*)
        apply *unfold-locales*
        subgoal by(*rule blinding-of-tree-hash*)(*rule a.hash*)
        subgoal using $Tree_m$*.IH* $Tree_m$*.prems* **by**(*fastforce simp add*: *eq-onp-def*)
        subgoal for *x y z* **using** $Tree_m$*.IH*[*of - - x y z*] $Tree_m$*.prems* **by** *fastforce*
        subgoal for *x y* **using** $Tree_m$*.IH*[*of - - x y*] $Tree_m$*.prems* **by** *fastforce*
        done
      interpret *blinding-of-on*
        {*a. set1-rt-*F_m *a* ⊆ *A* ∧ *set3-rt-*F_m *a* ⊆ *set3-rt-*F_m *x*}
        hash-rt-F_m *h ?h blinding-of-rt-*F_m *h bo ?h ?bo*
        unfolding *set-rt-*F_m*-eq hash-rt-*F_m*-alt-def* **..**
      from $Tree_m$*.prems* **show** *?case*
        apply(*intro conjI*)
      subgoal by(*fastforce intro*!: *blinding-of-tree.intros refl*[*unfolded hash-rt-*F_m*-alt-def*])
      subgoal by(*fastforce elim*!: *blinding-of-tree.cases trans*[*unfolded hash-rt-*F_m*-alt-def*]

                intro!: *blinding-of-tree.intros*)
      subgoal by(*fastforce elim*!: *blinding-of-tree.cases antisym*[*unfolded hash-rt-*F_m*-alt-def*])
        done
    qed
    then show *x* ∈ *?A* ⟹ *?bo x x*
      and ⟦ *?bo x y*; *?bo y z*; *x* ∈ *?A* ⟧ ⟹ *?bo x z*
      and ⟦ *?bo x y*; *?bo y x*; *x* ∈ *?A* ⟧ ⟹ *x* = *y*
      for *x y z* **by** *blast*+
  qed
qed

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

lemma *blinding-of-tree-mono*:
  bo ≤ *bo′* ⟹ *blinding-of-tree h bo* ≤ *blinding-of-tree h bo′*
  apply(*rule predicate2I*)
  apply(*erule blinding-of-tree.induct*)
  apply(*rule blinding-of-tree.intros*)
  apply(*erule blinding-of-rt-*F_m*-mono*[*THEN predicate2D*, *rotated* −*1*])
  apply(*blast*)+

done

2.7.3 Merging

definition *merge-rt-F_m*
:: $('a_m, 'a_h)$ *hash* $\Rightarrow$ $'a_m$ *merge* $\Rightarrow$ $('b_m, 'b_h)$ *hash* $\Rightarrow$ $'b_m$ *merge* $\Rightarrow$
$('a_m, 'a_h, 'b_m, 'b_h)$ *rose-tree-F_m merge*
where
merge-rt-F_m ha ma hr mr $\equiv$ *merge-blindable* (*hash-prod ha* (*hash-list hr*)) (*merge-prod ma* (*merge-list mr*))

lemma *merge-rt-F_m-cong* [*fundef-cong*]:
assumes $\bigwedge a\ b.$ ⟦ $a \in$ *set1-rt-F_m x*; $b \in$ *set1-rt-F_m y* ⟧ $\Longrightarrow$ *ma a b* = *ma' a b*
and $\bigwedge a\ b.$ ⟦ $a \in$ *set3-rt-F_m x*; $b \in$ *set3-rt-F_m y* ⟧ $\Longrightarrow$ *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*
using *assms*
apply(*cases x*; *cases y*; *simp add*: *merge-rt-F_m-def bind-UNION*)
apply(*rule arg-cong*[**where** *f*=*map-option* -])
apply(*blast intro*: *merge-prod-cong merge-list-cong*)
done

lemma *in-set1-blindable$_m$-iff*: $x \in$ *set1-blindable$_m$ y* $\longleftrightarrow$ *y* = *Unblinded x*
by(*cases y*) *auto*

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* $\subseteq A\}$ (*hash-tree h*) (*blinding-of-tree h bo*) (*merge-tree h m*)
(**is** *merge-on ?A ?h ?bo ?m*)
proof −
interpret *a*: *merge-on A h bo m* **by** *fact*
show *?thesis*
proof
have (*?h a* = *?h b* $\longrightarrow$ ($\exists$ *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*))) $\wedge$
(*?h a* $\neq$ *?h b* $\longrightarrow$ *?m a b* = *None*)
if $a \in$ *?A* **for** *a b* **using** *that* **unfolding** *mem-Collect-eq*
proof(*induction a arbitrary*: *b rule*: *rose-tree$_m$.induct*)
case (*Tree$_m$ x y*)

```
      interpret merge-on
        {y. set1-rt-F_m y ⊆ A ∧ set3-rt-F_m y ⊆ set3-rt-F_m x}
        hash-rt-F_m h ?h
        blinding-of-rt-F_m h bo ?h ?bo
        merge-rt-F_m h m ?h ?m
        unfolding set-rt-F_m-eq hash-rt-F_m-alt-def merge-rt-F_m-def
      proof
        fix a
        assume a: a ∈ set3-rt-F_m x
        with Tree_m.prems have a': set1-rose-tree_m a ⊆ A
          by(force simp add: bind-UNION)

        from a obtain l and ab where a'': ab ∈ set1-blindable_m x l ∈ snds ab a
∈ set l
          by(clarsimp simp add: bind-UNION)

        fix b
        from Tree_m.IH[OF a'' a', rule-format, of b]
        show hash-tree h a = hash-tree h b
            ⟹ ∃ ab. merge-tree h m a b = Some ab ∧ blinding-of-tree h bo a ab ∧
blinding-of-tree h bo b ab ∧
                      (∀ u. blinding-of-tree h bo a u ⟶ blinding-of-tree h bo b u ⟶
blinding-of-tree h bo ab u)
          and hash-tree h a ≠ hash-tree h b ⟹ merge-tree h m a b = None
          by(auto dest: sym)
      qed
      show ?case using Tree_m.prems
        apply(intro conjI strip)
        subgoal by(cases y)(fastforce dest!: join simp add: blinding-of-tree.simps)
        subgoal by (cases y) (fastforce dest!: undefined)
        done
    qed
    then show
      ?h a = ?h b ⟹ ∃ ab. ?m a b = Some ab ∧ ?bo a ab ∧ ?bo b ab ∧ (∀ u. ?bo a
u ⟶ ?bo b u ⟶ ?bo ab u)
      ?h a ≠ ?h b ⟹ ?m a b = None
      if a ∈ ?A for a b using that by blast+
  qed
qed

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

lemma option-bind-comm:
 ((x :: 'a option) ≫= (λy. c ≫= (λz. f y z))) = (c ≫= (λy. x ≫= (λz. f z y)))
  by(cases x; cases c; auto)

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 −
  interpret *merge-on UNIV h bo m* **using** *assms* **unfolding** *merkle-interface-aux*
.
  show *?thesis* **unfolding** *merkle-interface-aux*[*symmetric*] **..**
qed

lemma *merge-tree-cong* [*fundef-cong*]:
  assumes $\bigwedge a\ b.\ [\![\ a \in \mathit{set1\text{-}rose\text{-}tree}_m\ x;\ b \in \mathit{set1\text{-}rose\text{-}tree}_m\ y\ ]\!] \Longrightarrow m\ a\ b = m'$
$a\ b$
  shows $\mathit{merge\text{-}tree}\ h\ m\ x\ y = \mathit{merge\text{-}tree}\ h\ m'\ x\ y$
  using *assms*
  apply(*induction x y rule*: *merge-tree.induct*)
  apply(*simp add*: *bind-UNION*)
  apply(*rule arg-cong*[**where** *f*=*map-option* -])
  apply(*rule merge-rt-*F_m*-cong*; *simp add*: *bind-UNION*; *blast*)
  done

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 $('a_m, 'a_h)\ T_m = T_m\ (\mathit{the}\text{-}T_m\text{: } ('a_m, 'a_h, ('a_m, 'a_h)\ T_m, 'a_h\ T_h)\ F_m)$
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 $('a_h,\ 'b_h)\ G_h = G_h\ (\textit{the-}G_h\text{: } ('a_h\ T_h,\ 'b_h)\ F_h)$

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

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 $\textit{root-hash-F}' :: (('a_h,\ 'a_h,\ 'b_h,\ 'b_h)\ F_m,\ ('a_h,\ 'b_h)\ F_h)\ \textit{hash}$
— 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 $('a_m,\ 'a_h,\ 'b_m,\ 'b_h)\ \textit{hash-F} =$
$('a_m,\ 'a_h)\ \textit{hash} \Rightarrow ('b_m,\ 'b_h)\ \textit{hash} \Rightarrow (('a_m,\ 'a_h,\ 'b_m,\ 'b_h)\ F_m,\ ('a_h,\ 'b_h)\ F_h)\ \textit{hash}$
definition $\textit{root-hash-F} :: ('a_m,\ 'a_h,\ 'b_m,\ 'b_h)\ \textit{hash-F}$ **where**
$\textit{root-hash-F rha rhb} = \textit{root-hash-F}' \circ \textit{map-}F_m\ \textit{rha id rhb id}$

3.2.2 Least fixpoint

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

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

lemma *root-hash-T-simps* [*simp*]:
$\textit{root-hash-T rha}\ (T_m\ x) = T_h\ (\textit{root-hash-F rha}\ (\textit{root-hash-T rha})\ x)$
by(*simp add*: $\textit{root-hash-T-def}\ F_m\textit{.map-comp root-hash-F-def}\ T_h\textit{.map-id0}$)

3.2.3 Composition

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

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

lemma *root-hash-G-unfold*:
$\textit{root-hash-G rha rhb} = G_h \circ \textit{root-hash-F}\ (\textit{root-hash-T rha})\ \textit{rhb} \circ \textit{the-}G_m$

apply(*rule ext*)
subgoal for x
by(*cases x*)(*simp add*: *root-hash-G-def fun-eq-iff root-hash-F-def root-hash-T-def* F_m*.map-comp* T_m*.map-comp o-def* T_h*.map-id id-def*[*symmetric*])
done

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*)
by(*simp add*: *root-hash-G-def root-hash-T-def* F_m*.map-comp root-hash-F-def* T_h*.map-id0*)

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: ⟦ $boa \le boa'$; $bob \le bob'$ ⟧
⟹ *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* $\{x.\ set1\text{-}F_m\ x \subseteq A \wedge set3\text{-}F_m\ x \subseteq B\}$ (*root-hash-F rha rhb*) (*blinding-of-F rha boa rhb bob*)

lemma *blinding-of-F-mono-inductive*:
assumes *a*: $\bigwedge x\ y.\ boa\ x\ y \longrightarrow boa'\ x\ y$
and *b*: $\bigwedge x\ y.\ bob\ x\ y \longrightarrow bob'\ x\ y$
shows *blinding-of-F rha boa rhb bob x y* ⟶ *blinding-of-F rha boa' rhb bob' x y*
using *assms* **by**(*blast intro*: *blinding-of-F-mono*[*THEN predicate2D*, *OF predicate2I predicate2I*])

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 $bo \leq bo'$
shows *blinding-of-T rh bo* $\leq$ *blinding-of-T rh bo'*
by(*rule predicate2I*; *erule blinding-of-T.induct*)
(*blast intro*: *blinding-of-T.intros blinding-of-F-mono*[*THEN predicate2D*, *OF assms*, *rotated* -1])

lemma *blinding-of-T-root-hash*:
assumes $bo \leq$ *vimage2p rh rh* $(=)$
shows *blinding-of-T rh bo* $\leq$ *vimage2p* (*root-hash-T rh*) (*root-hash-T rh*) $(=)$
apply(*rule predicate2I vimage2pI*)+
apply(*erule blinding-of-T.induct*)
apply *simp*
apply(*drule blinding-respects-hashes-F*[*unfolded blinding-respects-hashes-def*, *THEN predicate2D*, *rotated* -1])
apply(*rule assms*)
apply(*blast intro*: *vimage2pI*)
apply(*simp add*: *vimage2p-def*)
done

lemma *blinding-respects-hashes-T* [*locale-witness*]:
blinding-respects-hashes rh bo $\Longrightarrow$ *blinding-respects-hashes* (*root-hash-T rh*) (*blinding-of-T rh bo*)
unfolding *blinding-respects-hashes-def* **by**(*rule blinding-of-T-root-hash*)

lemma *blinding-of-on-T* [*locale-witness*]:
assumes *blinding-of-on A rh bo*
shows *blinding-of-on* $\{x.\ set1\text{-}T_m\ x \subseteq A\}$ (*root-hash-T rh*) (*blinding-of-T rh bo*)
(**is** *blinding-of-on ?A ?h ?bo*)
proof $-$
interpret *a*: *blinding-of-on A rh bo* **by** *fact*
show *?thesis*
proof
have $?bo\ x\ x \wedge (?bo\ x\ y \longrightarrow ?bo\ y\ z \longrightarrow ?bo\ x\ z) \wedge (?bo\ x\ y \longrightarrow ?bo\ y\ x \longrightarrow x = y)$
if $x \in ?A$ **for** *x y z* **using** *that*
proof(*induction x arbitrary*: *y z*)
case $(T_m\ x)$
interpret *blinding-of-on*
$\{a.\ set1\text{-}F_m\ a \subseteq A \wedge set3\text{-}F_m\ a \subseteq set3\text{-}F_m\ x\}$
root-hash-F rh ?h
blinding-of-F rh bo ?h ?bo

apply(*rule blinding-of-on-F*[*OF assms*])
apply *unfold-locales*
subgoal using T_m*.IH* T_m*.prems* **by**(*force simp add*: *eq-onp-def*)
subgoal for *a b c* **using** T_m*.IH*[*of a b c*] T_m*.prems* **by** *auto*
subgoal for *a b* **using** T_m*.IH*[*of a b*] T_m*.prems* **by** *auto*
done
show *?case* **using** T_m*.prems*
apply(*intro conjI*)
subgoal by(*auto intro*: *blinding-of-T.intros refl*)
subgoal by(*auto elim*!: *blinding-of-T.cases trans intro*!: *blinding-of-T.intros*)
subgoal by(*auto elim*!: *blinding-of-T.cases dest*: *antisym*)
done
qed
then show $x \in$ *?A* $\Longrightarrow$ *?bo x x*
and $[\![$ *?bo x y*; *?bo y z*; $x \in$ *?A* $]\!] \Longrightarrow$ *?bo x z*
and $[\![$ *?bo x y*; *?bo y x*; $x \in$ *?A* $]\!] \Longrightarrow x = y$
for *x y z* **by** *blast+*
qed
qed

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*)
apply(*rule ext*)+
subgoal for *x y* **by**(*cases x*; *cases y*)(*simp-all add*: *blinding-of-G.simps fun-eq-iff vimage2p-def*)
done

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'*

unfolding *blinding-of-G-unfold*
by(*rule vimage2p-mono′ blinding-of-F-mono blinding-of-T-mono assms*)+

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*) (=)
unfolding *blinding-of-G-unfold root-hash-G-unfold vimage2p-comp o-apply*
apply(*rule vimage2p-mono′*)
apply(*rule order-trans*)
apply(*rule blinding-respects-hashes-F*[*unfolded blinding-respects-hashes-def*])
apply(*rule blinding-of-T-root-hash*)
apply(*rule assms*)+
apply(*rule vimage2p-mono′*)
apply(*simp add*: *vimage2p-def*)
done

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 −
interpret *a*: *blinding-of-on A rha boa* **by** *fact*
interpret *b*: *blinding-of-on B rhb bob* **by** *fact*
interpret *FT*: *blinding-of-on*
{*x*. *set1-*F_m *x* ⊆ {*x*. *set1-*T_m *x* ⊆ *A*} ∧ *set3-*F_m *x* ⊆ *B*}
root-hash-F (*root-hash-T rha*) *rhb*
blinding-of-F (*root-hash-T rha*) (*blinding-of-T rha boa*) *rhb bob*
..
show *?thesis*
proof
show *?bo* ≤ *vimage2p ?h ?h* (=)
using *a.hash b.hash*
by(*rule blinding-of-G-root-hash*)
show *?bo x x* **if** *x* ∈ *?A* **for** *x* **using** *that*
by(*cases x*; *hypsubst*)(*rule blinding-of-G.intros*; *rule FT.refl*; *auto*)
show *?bo x z* **if** *?bo x y ?bo y z x* ∈ *?A* **for** *x y z* **using** *that*
by(*fastforce elim*!: *blinding-of-G.cases intro*!: *blinding-of-G.intros elim*!: *FT.trans*)
show *x* = *y* **if** *?bo x y ?bo y x x* ∈ *?A* **for** *x y* **using** *that*
by(*clarsimp elim*!: *blinding-of-G.cases*)(*erule* (*1*) *FT.antisym*; *auto*)
qed
qed

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*]:
$[\![\bigwedge a\ b.\ a \in set1\text{-}F_m\ x \Longrightarrow ma\ a\ b = ma'\ a\ b;\ \bigwedge a\ b.\ a \in set3\text{-}F_m\ x \Longrightarrow mb\ a\ b = mb'\ a\ b\ ]\!]$
$\Longrightarrow merge\text{-}F\ rha\ ma\ rhb\ mb\ x\ y = merge\text{-}F\ rha\ ma'\ rhb\ mb'\ x\ y$
and
merge-on-F [*locale-witness*]:
$[\![\ merge\text{-}on\ A\ rha\ boa\ ma;\ merge\text{-}on\ B\ rhb\ bob\ mb\ ]\!]$
$\Longrightarrow merge\text{-}on\ \{x.\ set1\text{-}F_m\ x \subseteq A \wedge set3\text{-}F_m\ x \subseteq B\}\ (root\text{-}hash\text{-}F\ rha\ rhb)\ (blinding\text{-}of\text{-}F\ rha\ boa\ rhb\ bob)\ (merge\text{-}F\ rha\ ma\ rhb\ mb)$

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* $(\lambda x\ y.\ x \in set3\text{-}F_m\ (the\text{-}T_m\ y))$
apply(*rule wfpUNIVI*)
subgoal premises *IH*[*rule-format*] **for** *P x*
by(*induct x*)(*auto intro*: *IH*)
done

lemma *irrefl-subterm-T*: $x \in set3\text{-}F_m\ y \Longrightarrow y \neq the\text{-}T_m\ x$
using *wfP-subterm-T* **by** (*auto simp*: *wfp-def elim*!: *wf-irrefl*)

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**
$merge\text{-}T\ (T_m\ x)\ (T_m\ y) = map\text{-}option\ T_m\ (merge\text{-}F\ rh\ m\ (root\text{-}hash\text{-}T\ rh)\ merge\text{-}T\ x\ y)$
by *pat-completeness auto*
termination
apply(*relation* $\{(x, y).\ x \in set3\text{-}F_m\ (the\text{-}T_m\ y)\}$ <*lex*> $\{(x, y).\ x \in set3\text{-}F_m\ (the\text{-}T_m\ y)\}$)
apply(*auto simp add*: *wfp-def*[*symmetric*] *wfP-subterm-T*)
done

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 −
  interpret *a*: *merge-on A rh bo m* **by** *fact*
  show *?thesis*
  proof
    have $(?h\ a = ?h\ b \longrightarrow (\exists\, ab.\ ?m\ a\ b = Some\ ab \wedge ?bo\ a\ ab \wedge ?bo\ b\ ab \wedge (\forall\, u.\ ?bo\ a\ u \longrightarrow ?bo\ b\ u \longrightarrow ?bo\ ab\ u))) \wedge$
      $(?h\ a \neq ?h\ b \longrightarrow ?m\ a\ b = None)$
      if $a \in ?A$ **for** *a b* **using** *that* **unfolding** *mem-Collect-eq*
    proof(*induction a arbitrary*: *b*)
      case $(T_m\ x\ y)$
      interpret *merge-on* $\{y.\ set1\text{-}F_m\ y \subseteq A \wedge set3\text{-}F_m\ y \subseteq set3\text{-}F_m\ x\}$
        root-hash-F rh ?h blinding-of-F rh bo ?h ?bo merge-F rh m ?h ?m
      proof
        fix *a*
        assume *a*: $a \in set3\text{-}F_m\ x$
        with T_m*.prems* **have** a': $set1\text{-}T_m\ a \subseteq A$ **by** *auto*

        fix *b*
        from T_m*.IH*[*OF a* a', *rule-format*, *of b*]
        show *root-hash-T rh a = root-hash-T rh b*
         $\Longrightarrow \exists\, ab.$ *merge-T a b = Some ab* $\wedge$ *blinding-of-T rh bo a ab* $\wedge$ *blinding-of-T rh bo b ab* $\wedge$
                    $(\forall\, u.$ *blinding-of-T rh bo a u* $\longrightarrow$ *blinding-of-T rh bo b u* $\longrightarrow$ *blinding-of-T rh bo ab u*)
          and *root-hash-T rh a* $\neq$ *root-hash-T rh b* $\Longrightarrow$ *merge-T a b = None*
          by(*auto dest*: *sym*)
      qed
      show *?case* **using** T_m*.prems*
        apply(*intro conjI strip*)
        subgoal by(*cases y*)(*auto dest*!: *join simp add*: *blinding-of-T.simps*)
        subgoal by(*cases y*)(*auto dest*!: *undefined*)
        done
    qed
    then show
      $?h\ a = ?h\ b \Longrightarrow \exists\, ab.\ ?m\ a\ b = Some\ ab \wedge ?bo\ a\ ab \wedge ?bo\ b\ ab \wedge (\forall\, u.\ ?bo\ a\ u \longrightarrow ?bo\ b\ u \longrightarrow ?bo\ ab\ u)$
      $?h\ a \neq ?h\ b \Longrightarrow ?m\ a\ b = None$
      if $a \in ?A$ **for** *a b* **using** *that* **by** *blast*+
  qed
qed

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 $merge\text{-}T\ rh\ m\ x\ y = merge\text{-}T\ rh\ m'\ x\ y$
  using *assms*
  apply(*induction x y rule*: *merge-T.induct*)
  apply *simp*
  apply(*rule arg-cong*[**where** *f*=*map-option* -])
  apply(*blast intro*: *merge-F-cong*)
  done

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

lemma *merge-G-simps* [*simp*]:
  $merge\text{-}G\ (G_m\ x)\ (G_m\ y) = map\text{-}option\ G_m\ (merge\text{-}F\ (root\text{-}hash\text{-}T\ rha)\ (merge\text{-}T\ rha\ ma)\ rhb\ mb\ x\ y)$
  by(*simp*)

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 $merge\text{-}on\ \{x.\ set1\text{-}G_m\ x \subseteq A \wedge set3\text{-}G_m\ x \subseteq B\}\ (root\text{-}hash\text{-}G\ rha\ rhb)\ (blinding\text{-}of\text{-}G\ rha\ boa\ rhb\ bob)\ merge\text{-}G$
  (**is** *merge-on ?A ?h ?bo ?m*)
proof −
  interpret *a*: *merge-on A rha boa ma* **by** *fact*
  interpret *b*: *merge-on B rhb bob mb* **by** *fact*
  interpret *F*: *merge-on*
    $\{x.\ set1\text{-}F_m\ x \subseteq \{x.\ set1\text{-}T_m\ x \subseteq A\} \wedge set3\text{-}F_m\ x \subseteq B\}$
    root-hash-F (*root-hash-T rha*) *rhb*
    blinding-of-F (*root-hash-T rha*) (*blinding-of-T rha boa*) *rhb bob*
    merge-F (*root-hash-T rha*) (*merge-T rha ma*) *rhb mb*
    ..
  show *?thesis*
  proof
    show $\exists ab.\ ?m\ a\ b = Some\ ab \wedge ?bo\ a\ ab \wedge ?bo\ b\ ab \wedge (\forall u.\ ?bo\ a\ u \longrightarrow ?bo\ b$

$u \longrightarrow$ *?bo ab u*)
if *?h a* = *?h b a* ∈ *?A* **for** *a b* **using** *that*
by(*cases a*; *cases b*)(*auto dest*!: *F.join simp add*: *blinding-of-G.simps*)
show *?m a b* = *None* **if** *?h a* ≠ *?h b a* ∈ *?A* **for** *a b* **using** *that*
by(*cases a*; *cases b*)(*auto dest*!: *F.undefined*)
qed
qed

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

end

lemma *merge-G-cong* [*fundef-cong*]:
$\llbracket \bigwedge a\ b.\ a \in set1\text{-}G_m\ x \Longrightarrow ma\ a\ b = ma'\ a\ b;\ \bigwedge a\ b.\ a \in set3\text{-}G_m\ x \Longrightarrow mb\ a\ b = mb'\ a\ b\ \rrbracket$
$\Longrightarrow$ *merge-G rha ma rhb mb x y* = *merge-G rha ma′ rhb mb′ x y*
apply(*cases x*; *cases y*; *simp*)
apply(*rule arg-cong*[**where** *f*=*map-option* -])
apply(*blast intro*: *merge-F-cong merge-T-cong*)
done

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*
by(*cases x*) *simp-all*

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* :: $'a\ rose\text{-}tree \Rightarrow 'a_h\ rose\text{-}tree_h$ **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* :: $'a\ rose\text{-}tree \Rightarrow ('a_m, 'a_h)\ rose\text{-}tree_m$ **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-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-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*
by(*cases st rule*: *blind-source-tree.cases*)(*simp-all add*: *hash-rt-F*$_m$*-def*)

lemma *hash-embedded-simp*:
hash-tree h (*embed-source-tree e st*) = *hash-source-tree* ($h \circ e$) *st*
by(*induction st rule*: *embed-source-tree.induct*)(*simp add*: *hash-rt-F*$_m$*-def*)

lemma *blinded-embedded-same-hash*:
hash-tree h'' (*blind-source-tree* (*h o e*) *st*) = *hash-tree h* (*embed-source-tree e st*)
by(*simp add*: *hash-blinded-simp hash-embedded-simp*)

lemma *blinding-blinds* [*simp*]:
is-blinded (*blind-source-tree h t*)
by(*simp add*: *blind-source-tree-cases split*: *rose-tree.split*)

lemma *blinded-blinds-embedded*:
blinding-of-tree h bo (*blind-source-tree* (*h o e*) *st*) (*embed-source-tree e st*)
by(*cases st rule*: *blind-source-tree.cases*)(*simp-all add*: *hash-embedded-simp*)

fun *embed-hash-tree* :: *'ha rose-tree*$_h$ $\Rightarrow$ (*'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* $\Rightarrow$ *'a rose-tree list* **where**
children (*Tree* (*data*, *subtrees*)) = *subtrees*

fun *children*$_m$ :: (*'a*, $'a_h$) *rose-tree*$_m$ $\Rightarrow$ (*'a*, $'a_h$) *rose-tree*$_m$ *list* **where**
children$_m$ (*Tree*$_m$ (*Unblinded* (*data*, *subtrees*))) = *subtrees*
| *children*$_m$ - = *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*: $(l, a, r) \in set\ (splits\ ll) = (ll = l\ @\ a\ \#\ r)$
by(*induction ll arbitrary*: *l a r*)(*auto simp add*: *Cons-eq-append-conv*)

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 $'a\ path\text{-}elem = 'a \times 'a\ rose\text{-}tree\ list \times 'a\ rose\text{-}tree\ list$
type-synonym $'a\ path = 'a\ path\text{-}elem\ list$
type-synonym $'a\ zipper = 'a\ path \times 'a\ rose\text{-}tree$

definition *zipper-of-tree* :: $'a\ rose\text{-}tree \Rightarrow 'a\ zipper$ **where**
$zipper\text{-}of\text{-}tree\ t \equiv ([], t)$

fun *tree-of-zipper* :: $'a\ zipper \Rightarrow 'a\ rose\text{-}tree$ **where**
$tree\text{-}of\text{-}zipper\ ([], t) = t$
$|\ tree\text{-}of\text{-}zipper\ ((a, l, r)\ \#\ z, t) = tree\text{-}of\text{-}zipper\ (z, (Tree\ (a, (l\ @\ t\ \#\ r))))$

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

lemma *tree-of-zipper-id*[*iff*]: $tree\text{-}of\text{-}zipper\ (zipper\text{-}of\text{-}tree\ t) = t$
by(*simp add*: *zipper-of-tree-def*)

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

lemma *zipper-children-same-tree*:
assumes $z' \in set\ (zipper\text{-}children\ z)$
shows $tree\text{-}of\text{-}zipper\ z' = tree\text{-}of\text{-}zipper\ z$
proof−
obtain *p a ts* **where** *z*: $z = (p, Tree\ (a, ts))$
using *assms*
by(*cases z rule*: *zipper-children.cases*) (*simp-all*)

then obtain *l t r* **where** *ltr*: $z' = ((a, l, r)\ \#\ p, t)$ **and** $(l, t, r) \in set\ (splits\ ts)$
using *assms*
by(*auto*)

with *z* **show** *?thesis*
by(*simp add*: *splits-iff*)
qed

type-synonym $('a_m, 'a_h)\ path\text{-}elem_m = 'a_m \times ('a_m, 'a_h)\ rose\text{-}tree_m\ list \times ('a_m, 'a_h)\ rose\text{-}tree_m\ list$
type-synonym $('a_m, 'a_h)\ path_m = ('a_m, 'a_h)\ path\text{-}elem_m\ list$

type-synonym $('a_m,\ 'a_h)$ $zipper_m$ = $('a_m,\ 'a_h)$ $path_m \times ('a_m,\ 'a_h)$ $rose\text{-}tree_m$

definition $zipper\text{-}of\text{-}tree_m$ :: $('a_m,\ 'a_h)$ $rose\text{-}tree_m \Rightarrow ('a_m,\ 'a_h)$ $zipper_m$ **where**
$zipper\text{-}of\text{-}tree_m\ t \equiv ([],\ t)$

fun $tree\text{-}of\text{-}zipper_m$ :: $('a_m,\ 'a_h)$ $zipper_m \Rightarrow ('a_m,\ 'a_h)$ $rose\text{-}tree_m$ **where**
$tree\text{-}of\text{-}zipper_m\ ([],\ t) = t$
| $tree\text{-}of\text{-}zipper_m\ ((m,\ l,\ r)\ \#\ z,\ t) = tree\text{-}of\text{-}zipper_m\ (z,\ Tree_m\ (Unblinded\ (m,\ l$ $@\ t\ \#\ r)))$

lemma $tree\text{-}of\text{-}zipper_m\text{-}append$:
$tree\text{-}of\text{-}zipper_m\ (p\ @\ p',\ t) = tree\text{-}of\text{-}zipper_m\ (p',\ tree\text{-}of\text{-}zipper_m\ (p,\ t))$
by(*induction p arbitrary*: *p′ t*) *auto*

fun $zipper\text{-}children_m$ :: $('a_m,\ 'a_h)$ $zipper_m \Rightarrow ('a_m,\ 'a_h)$ $zipper_m$ *list* **where**
$zipper\text{-}children_m\ (p,\ Tree_m\ (Unblinded\ (a,\ ts))) = map\ (\lambda(l,\ t,\ r).\ ((a,\ l,\ r)\ \#\ p,$ $t))\ (splits\ ts)$
| $zipper\text{-}children_m$ - = []

lemma $zipper\text{-}children\text{-}same\text{-}tree_m$:
assumes $z' \in set\ (zipper\text{-}children_m\ z)$
shows $tree\text{-}of\text{-}zipper_m\ z' = tree\text{-}of\text{-}zipper_m\ z$
proof−
obtain *p a ts* **where** *z*: $z = (p,\ Tree_m\ (Unblinded\ (a,\ ts)))$
using *assms*
by(*cases z rule*: $zipper\text{-}children_m.cases$) (*simp-all*)

then obtain *l t r* **where** *ltr*: $z' = ((a,\ l,\ r)\ \#\ p,\ t)$ **and** $(l,\ t,\ r) \in set\ (splits\ ts)$
using *assms*
by(*auto*)

with *z* **show** *?thesis*
by(*simp add*: *splits-iff*)
qed

fun *blind-path-elem* :: $('a \Rightarrow 'a_m) \Rightarrow ('a_m \Rightarrow 'a_h) \Rightarrow 'a$ *path-elem* $\Rightarrow ('a_m,\ 'a_h)$ $path\text{-}elem_m$ **where**
blind-path-elem e h $(x,\ l,\ r) = (e\ x,\ map\ (blind\text{-}source\text{-}tree\ (h \circ e))\ l,\ map$ $(blind\text{-}source\text{-}tree\ (h \circ e))\ r)$

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-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 $\equiv$ *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'*))
$\longleftrightarrow$ *hash-tree h v* = *hash-tree h v'*
by(*induction p arbitrary*: *v v'*)(*auto simp add*: *hash-rt-F*$_m$*-def*)

fun *hash-path-elem* :: $('a_m \Rightarrow 'a_h) \Rightarrow ('a_m, 'a_h)$ *path-elem*$_m \Rightarrow ('a_h \times 'a_h$ *rose-tree*$_h$ *list* $\times$ $'a_h$ *rose-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'* ⟧ $\Longrightarrow$
hash-tree h (*tree-of-zipper*$_m$ (*p*, *v*)) = *hash-tree h'* (*tree-of-zipper*$_m$ (*p'*, *v'*))
by(*induction p arbitrary*: *p' v v'*)(*auto simp add*: *hash-rt-F*$_m$*-def*)

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 −
have *hash-path-elem h* $\circ$ *blind-path-elem e h* = *hash-path-elem h* $\circ$ *embed-path-elem e*
by(*clarsimp simp add*: *hash-blinded-simp hash-embedded-simp fun-eq-iff intro*!: *arg-cong2*[**where** *f*=*hash-source-tree*, *OF* - *refl*])
then show *?thesis*
by(*intro hash-view-zipper-eqI*)(*simp-all add*: *embed-path-def blind-path-def list.map-comp*)
qed

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*))
by(*induction* (*p*, *t*) *arbitrary*: *p t rule*: *tree-of-zipper.induct*)(*simp-all add*: *embed-path-def*)

lemma *childz-same-tree*:
(*l*, *t*, *r*) $\in$ *set* (*splits ts*) $\Longrightarrow$
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*)
by(*simp add*: *tree-of-embed-commute splits-iff del*: *embed-source-tree.simps*)

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'*))
$\longleftrightarrow$ *blinding-of-tree h bo t t'*
proof −

interpret *a*: *blinding-of-on UNIV h bo* **by** *fact*
interpret *tree*: *blinding-of-on UNIV hash-tree h blinding-of-tree h bo* **..**
show *?thesis*
by(*induction p arbitrary*: *t t'*)(*auto simp add*: *list-all2-append list.rel-refl a.refl tree.refl*)
qed

lemma *zipper-children-size-change* [*termination-simp*]: $(a, b) \in$ *set* (*zipper-children* $(p, v)) \Longrightarrow$ *size b* $<$ *size v*
by(*cases v*)(*clarsimp simp add*: *splits-iff Set.image-iff*)

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-path e h p*, *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 $z \in$ *set* (*zippers-rose-tree e h* (p, t))
shows *hash-tree h* ($tree\text{-}of\text{-}zipper_m$ *z*) =
hash-tree h ($tree\text{-}of\text{-}zipper_m$ (*embed-path e p*, *embed-source-tree e t*))
using *assms*(*1*)
proof(*induction* (p, t) *arbitrary*: *p t rule*: *zippers-rose-tree.induct*)
case (*1 p t*)
from *1.prems*[*unfolded zippers-rose-tree.simps*]
consider (*find*) *z* = (*blind-path e h p*, *embed-source-tree e t*)
| (*rec*) *x ts l t' r* **where** *t* = *Tree* (x, ts) $(l, t', r) \in$ *set* (*splits ts*) $z \in$ *set* (*zippers-rose-tree e h* $((x, l, r)$ # $p, t')$)
by(*cases t*)(*auto simp add*: *zipper-children.simps*)
then show *?case*
proof *cases*
case *rec*
then show *?thesis*
apply(*subst 1.hyps*[*of* (x, l, r) # *p t'*])
apply(*simp-all add*: *rev-image-eqI zipper-children.simps*)
by (*metis* (*no-types*) *childz-same-tree comp-apply embed-source-tree.simps rec*(*2*))
qed(*simp add*: *blind-embed-path-same-hash*)
qed

lemma *zippers-rose-tree-blinding-of*:
assumes *blinding-of-on UNIV h bo*
and *z*: $z \in$ *set* (*zippers-rose-tree e h* (p, t))

shows *blinding-of-tree h bo* (*tree-of-zipper*$_m$ *z*) (*tree-of-zipper*$_m$ (*blind-path e h p*, *embed-source-tree e t*))
using *z*
proof(*induction* (*p*, *t*) *arbitrary*: *p t rule*: *zippers-rose-tree.induct*)
case (*1 p t*)

interpret *a*: *blinding-of-on UNIV h bo* **by** *fact*
interpret *rt*: *blinding-of-on UNIV hash-tree h blinding-of-tree h bo* **..**

from *1.prems*[*unfolded zippers-rose-tree.simps*]
consider (*find*) *z* = (*blind-path e h p*, *embed-source-tree e t*)
| (*rec*) *x ts l t′ r* **where** *t* = *Tree* (*x*, *ts*) (*l*, *t′*, *r*) ∈ *set* (*splits ts*) *z* ∈ *set* (*zippers-rose-tree e h* ((*x*, *l*, *r*) # *p*, *t′*))
by(*cases t*)(*auto simp add*: *zipper-children.simps*)
then show *?case*
proof *cases*
case *find*
then show *?thesis* **by**(*simp add*: *rt.refl*)
next
case *rec*
then have *blinding-of-tree h bo*
(*tree-of-zipper*$_m$ *z*)
(*tree-of-zipper*$_m$ (*blind-path e h* ((*x*, *l*, *r*) # *p*), *embed-source-tree e t′*))
by(*intro 1*)(*simp add*: *rev-image-eqI zipper-children.simps*)
also have *blinding-of-tree h bo*
(*tree-of-zipper*$_m$ (*blind-path e h* ((*x*, *l*, *r*) # *p*), *embed-source-tree e t′*))
(*tree-of-zipper*$_m$ (*blind-path e h p*, *embed-source-tree e* (*Tree* (*x*, *ts*))))
using *rec*
by(*simp add*: *blind-path-def splits-iff blinding-of-same-path*[*OF assms*(*1*)] *a.refl list-all2-append list-all2-same list.rel-map blinded-blinds-embedded rt.refl*)
finally (*rt.trans*) **show** *?thesis* **using** *rec* **by** *simp*
qed
qed

lemma *zippers-rose-tree-neq-Nil*: *zippers-rose-tree e h* (*p*, *t*) ≠ []
by(*simp add*: *zippers-rose-tree.simps*)

lemma (**in** *comp-fun-idem*) *fold-set-union*:
assumes *finite A finite B*
shows *Finite-Set.fold f z* (*A* ∪ *B*) = *Finite-Set.fold f* (*Finite-Set.fold f z A*) *B*
using *assms*(*2*,*1*) **by** *induct simp-all*

context *merkle-interface* **begin**

lemma *comp-fun-idem-merge*: *comp-fun-idem* (λ*x yo*. *yo* ≫= *m x*)
apply(*unfold-locales*; *clarsimp simp add*: *fun-eq-iff split*: *bind-split*)
subgoal by (*metis assoc bind.bind-lunit bind.bind-lzero idem option.distinct*(*1*))
subgoal by (*simp add*: *join*)
done

interpretation *merge*: *comp-fun-idem* λx *yo*. *yo* $\ggg=$ *m x* **by**(*rule comp-fun-idem-merge*)

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

lemma *Merge-empty* [*simp*]: *Merge* {} = *None*
by(*simp add*: *Merge-def*)

lemma *Merge-infinite* [*simp*]: *infinite A* $\Longrightarrow$ *Merge A* = *None*
by(*simp add*: *Merge-def*)

lemma *Merge-cong-start*:
Finite-Set.fold (λx *yo*. *yo* $\ggg=$ *m x*) (*Some x*) *A* = *Finite-Set.fold* (λx *yo*. *yo* $\ggg=$ *m x*) (*Some y*) *A* (**is** *?lhs* = *?rhs*)
if *x* $\in$ *A y* $\in$ *A finite A*
proof −
have *?lhs* = *Finite-Set.fold* (λx *yo*. *yo* $\ggg=$ *m x*) (*Some x*) (*insert y A*) **using** *that* **by**(*simp add*: *insert-absorb*)
also have ... = *Finite-Set.fold* (λx *yo*. *yo* $\ggg=$ *m x*) (*m x y*) *A* **using** *that*
by(*simp only*: *merge.fold-insert-idem2*)(*simp add*: *commute*)
also have ... = *Finite-Set.fold* (λx *yo*. *yo* $\ggg=$ *m x*) (*Some y*) (*insert x A*) **using** *that*
by(*simp only*: *merge.fold-insert-idem2*)(*simp*)
also have ... = *?rhs* **using** *that* **by**(*simp add*: *insert-absorb*)
finally show *?thesis* **.**
qed

lemma *Merge-insert* [*simp*]: *Merge* (*insert x A*) = (*if A* = {} *then Some x else Merge A* $\ggg=$ *m x*) (**is** *?lhs* = *?rhs*)
proof(*cases finite A* $\wedge$ *A* $\neq$ {})
case *True*
then have *?lhs* = *Finite-Set.fold* (λx *yo*. *yo* $\ggg=$ *m x*) (*Some* (*SOME x*. *x* $\in$ *A*)) (*insert x A*)
unfolding *Merge-def* **by**(*subst Merge-cong-start*[**where** *y*=*SOME x*. *x* $\in$ *A*, *OF someI*])(*auto intro*: *someI*)
also have ... = *?rhs* **using** *True* **by**(*simp add*: *Merge-def*)
finally show *?thesis* **.**
qed(*auto simp add*: *Merge-def idem*)

lemma *Merge-insert-alt*:
Merge (*insert x A*) = *Finite-Set.fold* (λx *yo*. *yo* $\ggg=$ *m x*) (*Some x*) *A* (**is** *?lhs* = *?rhs*) **if** *finite A*
proof −
have *?lhs* = *Finite-Set.fold* (λx *yo*. *yo* $\ggg=$ *m x*) (*Some x*) (*insert x A*) **using** *that*
unfolding *Merge-def* **by**(*subst Merge-cong-start*[**where** *y*=*x*, *OF someI*]) *auto*
also have ... = *?rhs* **using** *that* **by**(*simp only*: *merge.fold-insert-idem2*)(*simp*

add: *idem*)
  finally show *?thesis* **.**
qed

lemma *Merge-None* [*simp*]: *Finite-Set.fold* (λx *yo*. *yo* $\ggg$ *m x*) *None A* = *None*
proof(*cases finite A*)
  case *True*
  then show *?thesis* **by**(*induction*) *auto*
qed *simp*

lemma *Merge-union*:
  Merge ($A \cup B$) = (*if* $A = \{\}$ *then Merge B else if* $B = \{\}$ *then Merge A else* (*Merge A* $\ggg$ (λa. *Merge B* $\ggg$ *m a*)))
  (**is** *?lhs* = *?rhs*)
proof(*cases finite* ($A \cup B$) $\wedge A \neq \{\} \wedge B \neq \{\}$)
  case *True*
  then have *?lhs* = *Finite-Set.fold* (λx *yo*. *yo* $\ggg$ *m x*) (*Some* (*SOME x*. $x \in B$)) ($B \cup A$)
    unfolding *Merge-def* **by**(*subst Merge-cong-start*[**where** *y*=*SOME x*. $x \in B$, *OF someI*])(*auto intro*: *someI simp add*: *Un-commute*)
  also have ... = *Finite-Set.fold* (λx *yo*. *yo* $\ggg$ *m x*) (*Merge B*) *A* **using** *True*
   by(*simp add*: *Merge-def merge.fold-set-union*)
  also have ... = *Merge A* $\ggg$ (λa. *Merge B* $\ggg$ *m a*)
  proof(*cases Merge B*)
   case (*Some b*)
   thus *?thesis* **using** *True*
     by *simp*(*subst Merge-insert-alt*[*symmetric*]; *simp add*: *commute*; *metis commute*)
  qed *simp*
  finally show *?thesis* **using** *True* **by** *simp*
qed *auto*

lemma *Merge-upper*:
  assumes *m*: *Merge A* = *Some x* **and** *y*: $y \in A$
  shows *bo y x*
proof –
  have *Merge A* = *Merge* (*insert y A*) **using** *y* **by**(*simp add*: *insert-absorb*)
  also have ... = *Merge A* $\ggg$ *m y* **using** *y* **by** *auto*
  finally have *m y x* = *Some x* **using** *m* **by** *simp*
  thus *?thesis* **by**(*simp add*: *bo-def*)
qed

lemma *Merge-least*:
  assumes *m*: *Merge A* = *Some x* **and** *u*[*rule-format*]: $\forall a \in A$. *bo a u*
  shows *bo x u*
proof –
  define *a* **where** $a \equiv$ *SOME x*. $x \in A$
  from *m* **have** *A*: *finite A* $A \neq \{\}$
   and $*$: *Finite-Set.fold* (λx *yo*. *yo* $\ggg$ *m x*) (*Some a*) *A* = *Some x*

by(*auto simp add*: *Merge-def a-def split*: *if-splits*)
from A **have** *bo a u* **by**(*auto intro*: *someI u simp add*: *a-def*)
with $A * u$ **show** *?thesis*
proof(*induction A arbitrary*: *a*)
case (*insert x A*)
then show *?case*
by(*cases m x a*; *cases* $A = \{\}$; *simp only*: *merge.fold-insert-idem2*; *simp*)(*auto simp add*: *join*)
qed *simp*
qed

lemma *Merge-defined*:
assumes *finite A* $A \neq \{\}$ $\forall a{\in}A.\ \forall b \in A.\ h\ a = h\ b$
shows *Merge A* $\neq$ *None*
proof
define *a* **where** $a \equiv$ *SOME a.* $a \in A$
have *a*: $a \in A$ **unfolding** *a-def* **using** *assms* **by**(*auto intro*: *someI*)
hence *ha*: $\forall b \in A.\ h\ b = h\ a$ **using** *assms* **by** *blast*

assume *m*: *Merge A* = *None*
hence *Finite-Set.fold* ($\lambda x\ yo.\ yo \ggg= m\ x$) (*Some a*) *A* = *None*
using *assms* **by**(*simp add*: *Merge-def a-def*)
with *assms*(*1*) **show** *False* **using** *ha*
proof(*induction arbitrary*: *a*)
case (*insert x A*)
thus *?case*
apply(*cases m x a*; *use nothing* **in** ‹*simp only*: *merge.fold-insert-idem2*›)
apply(*simp add*: *merge-respects-hashes*)
apply(*fastforce simp add*: *join vimage2p-def dest*: *hash*[*THEN predicate2D*])
done
qed *simp*
qed

lemma *Merge-hash*:
assumes *Merge A* = *Some x* $a \in A$
shows $h\ a = h\ x$
using *Merge-upper*[*OF assms*] *hash* **by**(*auto simp add*: *vimage2p-def*)

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 $\textit{view-metadata}_h$ = *view-metadata* $\textit{blindable}_h$
type-synonym $\textit{view-data}_h$ = *view-data* $\textit{blindable}_h$

datatype $\textit{view}_h = \textit{View}_h\ ((\textit{view-metadata}_h \times_h \textit{view-data}_h) \times_h \textit{view}_h\ \textit{list}_h)\ \textit{blindable}_h$

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

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

abbreviation (*input*) *hash-view-data* :: $(\textit{view-data}_m, \textit{view-data}_h)$ *hash* **where**
hash-view-data ≡ *hash-blindable id*
abbreviation (*input*) *blinding-of-view-data* :: $\textit{view-data}_m$ *blinding-of* **where**
blinding-of-view-data ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-view-data* :: $\textit{view-data}_m$ *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
by *unfold-locales*

abbreviation (*input*) *hash-view-metadata* :: $(\textit{view-metadata}_m, \textit{view-metadata}_h)$ *hash* **where**
hash-view-metadata ≡ *hash-blindable id*
abbreviation (*input*) *blinding-of-view-metadata* :: $\textit{view-metadata}_m$ *blinding-of* **where**
blinding-of-view-metadata ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-view-metadata* :: $\textit{view-metadata}_m$ *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
by *unfold-locales*

type-synonym *view-content* = *view-metadata* × *view-data*
type-synonym $\textit{view-content}_h = \textit{view-metadata}_h \times_h \textit{view-data}_h$
type-synonym $\textit{view-content}_m = \textit{view-metadata}_m \times_m \textit{view-data}_m$

locale *view-merkle* **begin**

type-synonym $view_h{}'$ = *view-content*$_h$ *rose-tree*$_h$

primrec *from-view*$_h$ :: $view_h \Rightarrow view_h{}'$ **where**
from-view$_h$ ($View_h$ x) = $Tree_h$ (*map-blindable*$_h$ (*map-prod id* (*map from-view*$_h$)) x)

primrec *to-view*$_h$ :: $view_h{}' \Rightarrow view_h$ **where**
to-view$_h$ ($Tree_h$ x) = $View_h$ (*map-blindable*$_h$ (*map-prod id* (*map to-view*$_h$)) x)

lemma *from-to-view*$_h$ [*simp*]: *from-view*$_h$ (*to-view*$_h$ x) = x
apply(*induction x*)
apply(*simp add*: *blindable*$_h$*.map-comp o-def prod.map-comp*)
apply(*simp cong*: *blindable*$_h$*.map-cong prod.map-cong list.map-cong add*: *blindable*$_h$*.map-id*[*unfolded id-def*])
done

lemma *to-from-view*$_h$ [*simp*]: *to-view*$_h$ (*from-view*$_h$ x) = x
apply(*induction x*)
apply(*simp add*: *blindable*$_h$*.map-comp o-def prod.map-comp*)
apply(*simp cong*: *blindable*$_h$*.map-cong prod.map-cong list.map-cong add*: *blindable*$_h$*.map-id*[*unfolded id-def*])
done

lemma *iso-view*$_h$: *type-definition from-view*$_h$ *to-view*$_h$ *UNIV*
by *unfold-locales simp-all*

setup-lifting *iso-view*$_h$

lemma *cr-view*$_h$*-Grp*: *cr-view*$_h$ = *Grp UNIV to-view*$_h$
by(*simp add*: *cr-view*$_h$*-def Grp-def fun-eq-iff*)(*transfer*, *auto*)

lemma $View_h$*-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-blindable*$_h$ (*rel-prod* (=) (*list-all2 pcr-view*$_h$)) ===> *pcr-view*$_h$) $Tree_h$ $View_h$
by(*simp add*: *rel-fun-def view*$_h$*.pcr-cr-eq cr-view*$_h$*-Grp list.rel-Grp eq-alt prod.rel-Grp blindable*$_h$*.rel-Grp*)
(*simp add*: *Grp-def*)

type-synonym $view_m{}'$ = (*view-content*$_m$, *view-content*$_h$) *rose-tree*$_m$

primrec *from-view*$_m$ :: $view_m \Rightarrow view_m{}'$ **where**
from-view$_m$ ($View_m$ x) = $Tree_m$ (*map-blindable*$_m$ (*map-prod id* (*map from-view*$_m$)) (*map-prod id* (*map from-view*$_h$)) x)

primrec *to-view*$_m$ :: $view_m{}' \Rightarrow view_m$ **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*
  apply(*induction x*)
  apply(*simp add*: *blindable*$_m$.*map-comp o-def prod.map-comp*)
  apply(*simp cong*: *blindable*$_m$.*map-cong prod.map-cong list.map-cong add*: *blindable*$_m$.*map-id*[*unfolded id-def*])
  done

lemma *to-from-view*$_m$ [*simp*]: *to-view*$_m$ (*from-view*$_m$ *x*) = *x*
  apply(*induction x*)
  apply(*simp add*: *blindable*$_m$.*map-comp o-def prod.map-comp*)
  apply(*simp cong*: *blindable*$_m$.*map-cong prod.map-cong list.map-cong add*: *blindable*$_m$.*map-id*[*unfolded id-def*])
  done

lemma *iso-view*$_m$: *type-definition from-view*$_m$ *to-view*$_m$ *UNIV*
  by *unfold-locales simp-all*

setup-lifting *iso-view*$_m$

lemma *cr-view*$_m$-*Grp*: *cr-view*$_m$ = *Grp UNIV to-view*$_m$
  by(*simp add*: *cr-view*$_m$-*def Grp-def fun-eq-iff*)(*transfer*, *auto*)

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*$_m$
 by(*simp add*: *rel-fun-def view*$_h$.*pcr-cr-eq view*$_m$.*pcr-cr-eq cr-view*$_h$-*Grp cr-view*$_m$-*Grp list.rel-Grp eq-alt prod.rel-Grp blindable*$_m$.*rel-Grp*)
    (*simp add*: *Grp-def*)

end

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

context begin
interpretation *view-merkle* .

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 **.**

lift-definition *blinding-of-view* :: $view_m$ *blinding-of* **is**
blinding-of-tree hash-view-content blinding-of-view-content **.**

lift-definition *merge-view* :: $view_m$ *merge* **is**
merge-tree hash-view-content merge-view-content **.**

lemma *merkle-view* [*locale-witness*]: *merkle-interface hash-view blinding-of-view merge-view*
by *transfer unfold-locales*

lemma *hash-view-simps* [*simp*]:
hash-view ($View_m$ *x*) =
$View_h$ (*hash-blindable* (*hash-prod hash-view-content* (*hash-list hash-view*)) *x*)
by *transfer*(*simp add*: *hash-rt-*F_m*-def prod.map-comp hash-blindable-def blindable*$_m$*.map-id*)

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*
by *transfer simp*

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* (λ*x y*. *blinding-of-view x y* $\wedge$ *P x y*))) *x y*
$\Longrightarrow$ *P* ($View_m$ *x*) ($View_m$ *y*)
shows *P x y*
using *assms* **by** *transfer*(*rule blinding-of-tree.induct*)

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*)
by *transfer simp*

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*$_h$ $\times_h$ *participant-metadata*$_h$) $\times_h$ *view*$_h$ *list*$_h$) *blindable*$_h$)

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

abbreviation (*input*) *hash-common-metadata* :: (*common-metadata*$_m$, *common-metadata*$_h$) *hash* **where**
hash-common-metadata ≡ *hash-blindable id*
abbreviation (*input*) *blinding-of-common-metadata* :: *common-metadata*$_m$ *blinding-of* **where**
blinding-of-common-metadata ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-common-metadata* :: *common-metadata*$_m$ *merge* **where**
merge-common-metadata ≡ *merge-blindable id merge-discrete*

abbreviation (*input*) *hash-participant-metadata* :: (*participant-metadata*$_m$, *participant-metadata*$_h$) *hash* **where**
hash-participant-metadata ≡ *hash-blindable id*
abbreviation (*input*) *blinding-of-participant-metadata* :: *participant-metadata*$_m$ *blinding-of* **where**
blinding-of-participant-metadata ≡ *blinding-of-blindable id* (=)
abbreviation (*input*) *merge-participant-metadata* :: *participant-metadata*$_m$ *merge* **where**
merge-participant-metadata ≡ *merge-blindable id merge-discrete*

locale *transaction-merkle* **begin**

lemma *iso-transaction*$_h$: *type-definition the-Transaction*$_h$ *Transaction*$_h$ *UNIV*
by *unfold-locales simp-all*

setup-lifting *iso-transaction*$_h$

lemma *Transaction*$_h$*-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((=) ===> *pcr-transaction*$_h$) *id Transaction*$_h$
by(*simp add*: *transaction*$_h$*.pcr-cr-eq cr-transaction*$_h$*-def rel-fun-def*)

lemma *iso-transaction*$_m$: *type-definition the-Transaction*$_m$ *Transaction*$_m$ *UNIV*
by *unfold-locales simp-all*

setup-lifting *iso-transaction*$_m$

lemma *Transaction*$_m$*-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((=) ===> *pcr-transaction*$_m$) *id Transaction*$_m$

by(*simp add*: *transaction*$_m$*.pcr-cr-eq cr-transaction*$_m$*-def rel-fun-def*)

end

code-datatype *Transaction*$_h$
code-datatype *Transaction*$_m$

context begin
interpretation *transaction-merkle* **.**

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*)) **.**

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*)) **.**

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*)) **.**

lemma *merkle-transaction* [*locale-witness*]:
merkle-interface hash-transaction blinding-of-transaction merge-transaction
by *transfer unfold-locales*

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
by(*rule merkle-transaction*)

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*
by(*induction x*)(*clarsimp cong*: *map-cong*)

lemma *to-from-view* [*simp*]: *to-view* (*from-view x*) = *x*
by(*induction x*)(*clarsimp cong*: *map-cong*)

lemma *iso-view*: *type-definition from-view to-view UNIV*
by *unfold-locales simp-all*

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*))
by(*simp add*: *View'-def*)

lemma *cr-view-Grp*: *cr-view* = *Grp UNIV to-view*
by(*simp add*: *cr-view-def Grp-def fun-eq-iff*)(*transfer*, *auto*)

lemma *View'-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-prod* (=) (*list-all2 pcr-view*) ===> *pcr-view*) *Tree View'*
by(*simp add*: *view.pcr-cr-eq cr-view-Grp eq-alt prod.rel-Grp rose-tree.rel-Grp list.rel-Grp*)
(*auto simp add*: *Grp-def View'-def*)

end

code-datatype *View*

context begin
interpretation *view-merkle* **.**

abbreviation *embed-view-content* :: *view-metadata* × *view-data* ⇒ $\textit{view-metadata}_m$ × $\textit{view-data}_m$ **where**
embed-view-content ≡ *map-prod Unblinded Unblinded*

lift-definition *embed-view* :: *view* ⇒ $\textit{view}_m$ **is** *embed-source-tree embed-view-content* **.**

lemma *embed-view-simps* [*simp*]:
embed-view (*View vm vd vs*) = $\textit{View}_m$ (*Unblinded* ((*Unblinded vm*, *Unblinded vd*), *map embed-view vs*))

unfolding *View-View′* **by** *transfer simp*

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*))
by(*simp add*: *Transaction′-def*)

lemma *the-Transaction-inverse* [*simp*]: *Transaction′* (*the-Transaction x*) = *x*
by(*cases x*)(*simp add*: *Transaction′-def*)

lemma *Transaction′-inverse* [*simp*]: *the-Transaction* (*Transaction′ x*) = *x*
by(*simp add*: *Transaction′-def split-def*)

lemma *iso-transaction*: *type-definition the-Transaction Transaction′ UNIV*
by *unfold-locales simp-all*

setup-lifting *iso-transaction*

lemma *Transaction′-transfer* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((=) ===> *pcr-transaction*) *id Transaction′*
by(*simp add*: *transaction.pcr-cr-eq cr-transaction-def rel-fun-def*)

end

code-datatype *Transaction*

context begin
interpretation *transaction-merkle* **.**

lift-definition *embed-transaction* :: *transaction* ⇒ $transaction_m$ **is**
Unblinded ∘ *map-prod* (*map-prod Unblinded Unblinded*) (*map embed-view*) **.**

lemma *embed-transaction-simps* [*simp*]:
embed-transaction (*Transaction cm pm views*) =
$Transaction_m$ (*Unblinded* ((*Unblinded cm*, *Unblinded pm*), *map embed-view views*))
for *views* **unfolding** *Transaction-Transaction′* **by** *transfer simp*

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*)
by(*induction view*)(*auto simp add*: *list.rel-map intro*!: *list.rel-refl-strong*)

lemma *blinding-of-mediator-transaction-tree*:
blinding-of-transaction (*mediator-transaction-tree tt*) (*embed-transaction tt*)
by(*cases tt*)(*auto simp add*: *list.rel-map intro*: *list.rel-refl-strong*)

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 $view\text{-}path\text{-}elem_m$ = ($view\text{-}metadata_m$ $\times_m$ $view\text{-}data_m$) × $view_m$ $list_m$ × $view_m$ $list_m$
type-synonym $view\text{-}path_m$ = $view\text{-}path\text{-}elem_m$ *list*
type-synonym $view\text{-}zipper_m$ = $view\text{-}path_m$ × $view_m$

context begin
interpretation *view-merkle* **.**

lift-definition *zipper-of-view* :: *view* ⇒ *view-zipper* **is** *zipper-of-tree* **.**
lift-definition *view-of-zipper* :: *view-zipper* ⇒ *view* **is** *tree-of-zipper* **.**

lift-definition $zipper\text{-}of\text{-}view_m$ :: $view_m$ ⇒ $view\text{-}zipper_m$ **is** $zipper\text{-}of\text{-}tree_m$ **.**
lift-definition $view\text{-}of\text{-}zipper_m$ :: $view\text{-}zipper_m$ ⇒ $view_m$ **is** $tree\text{-}of\text{-}zipper_m$ **.**

lemma $view\text{-}of\text{-}zipper_m\text{-}Nil$ [*simp*]: $view\text{-}of\text{-}zipper_m$ ([], *t*) = *t*
by *transfer simp*

lift-definition *blind-view-path-elem* :: *view-path-elem* ⇒ $view\text{-}path\text{-}elem_m$ **is**
blind-path-elem embed-view-content hash-view-content .

lift-definition *blind-view-path* :: *view-path* ⇒ $view\text{-}path_m$ **is**
blind-path embed-view-content hash-view-content .

lift-definition *embed-view-path-elem* :: *view-path-elem* ⇒ $view\text{-}path\text{-}elem_m$ **is**
embed-path-elem embed-view-content .

lift-definition *embed-view-path* :: *view-path* ⇒ $view\text{-}path_m$ **is**
embed-path embed-view-content .

lift-definition *hash-view-path-elem* :: $view\text{-}path\text{-}elem_m$ ⇒ ($view\text{-}content_h$ × $view_h$ *list* × $view_h$ *list*) **is**
hash-path-elem hash-view-content .

lift-definition *zippers-view* :: *view-zipper* ⇒ $view\text{-}zipper_m$ *list* **is**
zippers-rose-tree embed-view-content hash-view-content .

lemma *embed-view-path-Nil* [*simp*]: *embed-view-path* [] = []
by *transfer*(*simp add*: *embed-path-def*)

lemma *zippers-view-same-hash*:
assumes *z* ∈ *set* (*zippers-view* (*p*, *t*))
shows *hash-view* ($view\text{-}of\text{-}zipper_m$ *z*) = *hash-view* ($view\text{-}of\text{-}zipper_m$ (*embed-view-path p*, *embed-view t*))
using *assms* **by** *transfer*(*rule zippers-rose-tree-same-hash′*)

lemma *zippers-view-blinding-of*:
assumes *z* ∈ *set* (*zippers-view* (*p*, *t*))
shows *blinding-of-view* ($view\text{-}of\text{-}zipper_m$ *z*) ($view\text{-}of\text{-}zipper_m$ (*blind-view-path p*, *embed-view t*))
using *assms* **by** *transfer*(*rule zippers-rose-tree-blinding-of*, *unfold-locales*)

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*)
by(*cases view*) *simp*

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*)
by(*cases transaction*)(*simp add*: *hash-blind-view*)

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

fun *view-recipients* :: $view_m$ ⇒ *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 $view\text{-}of\text{-}zipper_m$
(*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 ($\lambda view_m$. $Transaction_m$ (*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*)
by(*auto simp add*: *view-trees-for-def dest*: *zippers-view-same-hash*)

lemma *transaction-views-for-same-hash*:
t_m ∈ *set* (*transaction-views-for t*) ⟹ *hash-transaction* t_m = *hash-transaction* (*embed-transaction t*)
by(*cases t*)(*clarsimp simp add*: *splits-iff hash-blind-view view-trees-for-same-hash*)

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(*cases transaction-views-for t* = [])

case *True* **thus** *?thesis* **by**(*simp add*: *transaction-projection-for-def Let-def hash-blind-transaction*)
next
case *False*
then have *transaction.Merge* (*set* (*transaction-views-for t*)) $\neq$ *None*
by(*intro transaction.Merge-defined*)(*auto simp add*: *transaction-views-for-same-hash*)
with *False* **show** *?thesis*
apply(*clarsimp simp add*: *transaction-projection-for-def neq-Nil-conv simp del*: *transaction.Merge-insert*)
apply(*drule transaction.Merge-hash*[*symmetric*], *blast*)
apply(*auto intro*: *transaction-views-for-same-hash*)
done
qed

lemma *transaction-projection-for-upper*:
assumes $t_m \in$ *set* (*transaction-views-for t*)
shows *blinding-of-transaction* t_m (*transaction-projection-for t*)
proof −
from *assms* **have** *transaction.Merge* (*set* (*transaction-views-for t*)) $\neq$ *None*
by(*intro transaction.Merge-defined*)(*auto simp add*: *transaction-views-for-same-hash*)
with *assms* **show** *?thesis*
by(*auto simp add*: *transaction-projection-for-def Let-def dest*: *transaction.Merge-upper*)
qed

end

end