

# First-Order Query Evaluation

Martin Raszyk

February 6, 2026

## Abstract

We formalize first-order query evaluation over an infinite domain with equality. We first define the syntax and semantics of first-order logic with equality. Next we define a locale *eval\_fo* abstracting a representation of a potentially infinite set of tuples satisfying a first-order query over finite relations. Inside the locale, we define a function *eval* checking if the set of tuples satisfying a first-order query over a database (an interpretation of the query's predicates) is finite (i.e., deciding *relative safety*) and computing the set of satisfying tuples if it is finite. Altogether the function *eval* solves *capturability* [2] of first-order logic with equality. We also use the function *eval* to prove a code equation for the semantics of first-order logic, i.e., the function checking if a first-order query over a database is satisfied by a variable assignment.

We provide an interpretation of the locale *eval\_fo* based on the approach by Ailamazyan et al. [1]. A core notion in the interpretation is the active domain of a query and a database that contains all domain elements that occur in the database or interpret the query's constants. We prove the main theorem of Ailamazyan et al. [1] relating the satisfaction of a first-order query over an infinite domain to the satisfaction of this query over a finite domain consisting of the active domain and a few additional domain elements (outside the active domain) whose number only depends on the query. In our interpretation of the locale *eval\_fo*, we use a potentially higher number of the additional domain elements, but their number still only depends on the query and thus has no effect on the data complexity [3] of query evaluation. Our interpretation yields an *executable* function *eval*. The time complexity of *eval* on a query is linear in the total number of tuples in the intermediate relations for the subqueries. Specifically, we build a database index to evaluate a conjunction. We also optimize the case of a negated subquery in a conjunction. Finally, we export code for the infinite domain of natural numbers.

## Contents

theory *FO*

imports *Main*

begin

abbreviation *sorted\_distinct xs*  $\equiv$  *sorted xs*  $\wedge$  *distinct xs*

datatype 'a *fo\_term* = *Const* 'a | *Var* nat

type\_synonym 'a *val* = nat  $\Rightarrow$  'a

fun *list\_fo\_term* :: 'a *fo\_term*  $\Rightarrow$  'a list **where**

*list\_fo\_term* (*Const* c) = [c]

| *list\_fo\_term* \_ = []

fun *fv\_fo\_term\_list* :: 'a *fo\_term*  $\Rightarrow$  nat list **where**

*fv\_fo\_term\_list* (*Var* n) = [n]

| *fv\_fo\_term\_list* \_ = []

**fun** *fv\_fo\_term\_set* :: 'a fo\_term ⇒ nat set **where**  
*fv\_fo\_term\_set* (Var n) = {n}  
| *fv\_fo\_term\_set* \_ = {}

**definition** *fv\_fo\_terms\_set* :: ('a fo\_term) list ⇒ nat set **where**  
*fv\_fo\_terms\_set* ts =  $\bigcup$ (set (map *fv\_fo\_term\_set* ts))

**fun** *fv\_fo\_terms\_list\_rec* :: ('a fo\_term) list ⇒ nat list **where**  
*fv\_fo\_terms\_list\_rec* [] = []  
| *fv\_fo\_terms\_list\_rec* (t # ts) = *fv\_fo\_term\_list* t @ *fv\_fo\_terms\_list\_rec* ts

**definition** *fv\_fo\_terms\_list* :: ('a fo\_term) list ⇒ nat list **where**  
*fv\_fo\_terms\_list* ts = *remdups\_adj* (sort (*fv\_fo\_terms\_list\_rec* ts))

**fun** *eval\_term* :: 'a val ⇒ 'a fo\_term ⇒ 'a (infix <·> 60) **where**  
*eval\_term* σ (Const c) = c  
| *eval\_term* σ (Var n) = σ n

**definition** *eval\_terms* :: 'a val ⇒ ('a fo\_term) list ⇒ 'a list (infix <⊙> 60) **where**  
*eval\_terms* σ ts = map (*eval\_term* σ) ts

**lemma** *finite\_set\_fo\_term*: finite (set\_fo\_term t)  
⟨proof⟩

**lemma** *list\_fo\_term\_set*: set (list\_fo\_term t) = set\_fo\_term t  
⟨proof⟩

**lemma** *finite\_fv\_fo\_term\_set*: finite (fv\_fo\_term\_set t)  
⟨proof⟩

**lemma** *fv\_fo\_term\_setD*: n ∈ fv\_fo\_term\_set t ⇒ t = Var n  
⟨proof⟩

**lemma** *fv\_fo\_term\_set\_list*: set (fv\_fo\_term\_list t) = fv\_fo\_term\_set t  
⟨proof⟩

**lemma** *sorted\_distinct\_fv\_fo\_term\_list*: sorted\_distinct (fv\_fo\_term\_list t)  
⟨proof⟩

**lemma** *fv\_fo\_term\_set\_cong*: fv\_fo\_term\_set t = fv\_fo\_term\_set (map\_fo\_term f t)  
⟨proof⟩

**lemma** *fv\_fo\_terms\_setI*: Var m ∈ set ts ⇒ m ∈ fv\_fo\_terms\_set ts  
⟨proof⟩

**lemma** *fv\_fo\_terms\_setD*: m ∈ fv\_fo\_terms\_set ts ⇒ Var m ∈ set ts  
⟨proof⟩

**lemma** *finite\_fv\_fo\_terms\_set*: finite (fv\_fo\_terms\_set ts)  
⟨proof⟩

**lemma** *fv\_fo\_terms\_set\_list*: set (fv\_fo\_terms\_list ts) = fv\_fo\_terms\_set ts  
⟨proof⟩

**lemma** *distinct\_remdups\_adj\_sort*: sorted xs ⇒ distinct (remdups\_adj xs)  
⟨proof⟩

**lemma** *sorted\_distinct\_fv\_fo\_terms\_list*: sorted\_distinct (fv\_fo\_terms\_list ts)

*<proof>*

**lemma** *fv\_fo\_terms\_set\_cong*:  $fv\_fo\_terms\_set\ ts = fv\_fo\_terms\_set\ (map\ (map\_fo\_term\ f)\ ts)$   
*<proof>*

**lemma** *eval\_term\_cong*:  $(\bigwedge n. n \in fv\_fo\_term\_set\ t \implies \sigma\ n = \sigma'\ n) \implies$   
 $eval\_term\ \sigma\ t = eval\_term\ \sigma'\ t$   
*<proof>*

**lemma** *eval\_terms\_fv\_fo\_terms\_set*:  $\sigma \odot ts = \sigma' \odot ts \implies n \in fv\_fo\_terms\_set\ ts \implies \sigma\ n = \sigma'\ n$   
*<proof>*

**lemma** *eval\_terms\_cong*:  $(\bigwedge n. n \in fv\_fo\_terms\_set\ ts \implies \sigma\ n = \sigma'\ n) \implies$   
 $eval\_terms\ \sigma\ ts = eval\_terms\ \sigma'\ ts$   
*<proof>*

**datatype**  $( 'a, 'b )\ fo\_fmla =$   
  *Pred*  $'b\ ( 'a\ fo\_term )\ list$   
| *Bool* *bool*  
| *Eqa*  $'a\ fo\_term\ 'a\ fo\_term$   
| *Neg*  $( 'a, 'b )\ fo\_fmla$   
| *Conj*  $( 'a, 'b )\ fo\_fmla\ ( 'a, 'b )\ fo\_fmla$   
| *Disj*  $( 'a, 'b )\ fo\_fmla\ ( 'a, 'b )\ fo\_fmla$   
| *Exists* *nat*  $( 'a, 'b )\ fo\_fmla$   
| *Forall* *nat*  $( 'a, 'b )\ fo\_fmla$

**fun** *fv\_fo\_fmla\_list\_rec* ::  $( 'a, 'b )\ fo\_fmla \Rightarrow nat\ list$  **where**  
  *fv\_fo\_fmla\_list\_rec* (*Pred*  $\_ ts$ ) = *fv\_fo\_terms\_list*  $ts$   
| *fv\_fo\_fmla\_list\_rec* (*Bool*  $b$ ) = []  
| *fv\_fo\_fmla\_list\_rec* (*Eqa*  $t\ t'$ ) = *fv\_fo\_term\_list*  $t @ fv\_fo\_term\_list\ t'$   
| *fv\_fo\_fmla\_list\_rec* (*Neg*  $\varphi$ ) = *fv\_fo\_fmla\_list\_rec*  $\varphi$   
| *fv\_fo\_fmla\_list\_rec* (*Conj*  $\varphi\ \psi$ ) = *fv\_fo\_fmla\_list\_rec*  $\varphi @ fv\_fo\_fmla\_list\_rec\ \psi$   
| *fv\_fo\_fmla\_list\_rec* (*Disj*  $\varphi\ \psi$ ) = *fv\_fo\_fmla\_list\_rec*  $\varphi @ fv\_fo\_fmla\_list\_rec\ \psi$   
| *fv\_fo\_fmla\_list\_rec* (*Exists*  $n\ \varphi$ ) = *filter*  $(\lambda m. n \neq m)$  (*fv\_fo\_fmla\_list\_rec*  $\varphi$ )  
| *fv\_fo\_fmla\_list\_rec* (*Forall*  $n\ \varphi$ ) = *filter*  $(\lambda m. n \neq m)$  (*fv\_fo\_fmla\_list\_rec*  $\varphi$ )

**definition** *fv\_fo\_fmla\_list* ::  $( 'a, 'b )\ fo\_fmla \Rightarrow nat\ list$  **where**  
  *fv\_fo\_fmla\_list*  $\varphi = remdups\_adj\ (sort\ (fv\_fo\_fmla\_list\_rec\ \varphi))$

**fun** *fv\_fo\_fmla* ::  $( 'a, 'b )\ fo\_fmla \Rightarrow nat\ set$  **where**  
  *fv\_fo\_fmla* (*Pred*  $\_ ts$ ) = *fv\_fo\_terms\_set*  $ts$   
| *fv\_fo\_fmla* (*Bool*  $b$ ) = {}  
| *fv\_fo\_fmla* (*Eqa*  $t\ t'$ ) = *fv\_fo\_term\_set*  $t \cup fv\_fo\_term\_set\ t'$   
| *fv\_fo\_fmla* (*Neg*  $\varphi$ ) = *fv\_fo\_fmla*  $\varphi$   
| *fv\_fo\_fmla* (*Conj*  $\varphi\ \psi$ ) = *fv\_fo\_fmla*  $\varphi \cup fv\_fo\_fmla\ \psi$   
| *fv\_fo\_fmla* (*Disj*  $\varphi\ \psi$ ) = *fv\_fo\_fmla*  $\varphi \cup fv\_fo\_fmla\ \psi$   
| *fv\_fo\_fmla* (*Exists*  $n\ \varphi$ ) = *fv\_fo\_fmla*  $\varphi - \{n\}$   
| *fv\_fo\_fmla* (*Forall*  $n\ \varphi$ ) = *fv\_fo\_fmla*  $\varphi - \{n\}$

**lemma** *finite\_fv\_fo\_fmla*: *finite* (*fv\_fo\_fmla*  $\varphi$ )  
*<proof>*

**lemma** *fv\_fo\_fmla\_list\_set*: *set* (*fv\_fo\_fmla\_list*  $\varphi$ ) = *fv\_fo\_fmla*  $\varphi$   
*<proof>*

**lemma** *sorted\_distinct\_fv\_list*: *sorted\_distinct* (*fv\_fo\_fmla\_list*  $\varphi$ )  
*<proof>*

**lemma** *length\_fv\_fo\_fmula\_list*:  $\text{length } (\text{fv\_fo\_fmula\_list } \varphi) = \text{card } (\text{fv\_fo\_fmula } \varphi)$   
 ⟨proof⟩

**lemma** *fv\_fo\_fmula\_list\_eq*:  $\text{fv\_fo\_fmula } \varphi = \text{fv\_fo\_fmula } \psi \implies \text{fv\_fo\_fmula\_list } \varphi = \text{fv\_fo\_fmula\_list } \psi$   
 ⟨proof⟩

**lemma** *fv\_fo\_fmula\_list\_Conj*:  $\text{fv\_fo\_fmula\_list } (\text{Conj } \varphi \ \psi) = \text{fv\_fo\_fmula\_list } (\text{Conj } \psi \ \varphi)$   
 ⟨proof⟩

**type\_synonym** 'a table = ('a list) set

**type\_synonym** ('t, 'b) fo\_intp = 'b × nat ⇒ 't

**fun** *wf\_fo\_intp* :: ('a, 'b) fo\_fmula ⇒ ('a table, 'b) fo\_intp ⇒ bool **where**  
 | *wf\_fo\_intp* (Pred r ts) I ⇔ finite (I (r, length ts))  
 | *wf\_fo\_intp* (Bool b) I ⇔ True  
 | *wf\_fo\_intp* (Eqa t t') I ⇔ True  
 | *wf\_fo\_intp* (Neg φ) I ⇔ *wf\_fo\_intp* φ I  
 | *wf\_fo\_intp* (Conj φ ψ) I ⇔ *wf\_fo\_intp* φ I ∧ *wf\_fo\_intp* ψ I  
 | *wf\_fo\_intp* (Disj φ ψ) I ⇔ *wf\_fo\_intp* φ I ∧ *wf\_fo\_intp* ψ I  
 | *wf\_fo\_intp* (Exists n φ) I ⇔ *wf\_fo\_intp* φ I  
 | *wf\_fo\_intp* (Forall n φ) I ⇔ *wf\_fo\_intp* φ I

**fun** *sat* :: ('a, 'b) fo\_fmula ⇒ ('a table, 'b) fo\_intp ⇒ 'a val ⇒ bool **where**  
 | *sat* (Pred r ts) I σ ⇔ σ ⊙ ts ∈ I (r, length ts)  
 | *sat* (Bool b) I σ ⇔ b  
 | *sat* (Eqa t t') I σ ⇔ σ · t = σ · t'  
 | *sat* (Neg φ) I σ ⇔ ¬*sat* φ I σ  
 | *sat* (Conj φ ψ) I σ ⇔ *sat* φ I σ ∧ *sat* ψ I σ  
 | *sat* (Disj φ ψ) I σ ⇔ *sat* φ I σ ∨ *sat* ψ I σ  
 | *sat* (Exists n φ) I σ ⇔ (∃ x. *sat* φ I (σ(n := x)))  
 | *sat* (Forall n φ) I σ ⇔ (∀ x. *sat* φ I (σ(n := x)))

**lemma** *sat\_fv\_cong*:  $(\bigwedge n. n \in \text{fv\_fo\_fmula } \varphi \implies \sigma \ n = \sigma' \ n) \implies \text{sat } \varphi \ I \ \sigma \longleftrightarrow \text{sat } \varphi \ I \ \sigma'$   
 ⟨proof⟩

**definition** *proj\_sat* :: ('a, 'b) fo\_fmula ⇒ ('a table, 'b) fo\_intp ⇒ 'a table **where**  
*proj\_sat* φ I = (λσ. map σ (fv\_fo\_fmula\_list φ)) ' {σ. sat φ I σ}

**end**

**theory** *Eval\_FO*

**imports** *HOL-Library.Infinite\_Typeclass FO*

**begin**

**datatype** 'a eval\_res = Fin 'a table | Infin | Wf\_error

**locale** *eval\_fo* =

**fixes** *wf* :: ('a :: infinite, 'b) fo\_fmula ⇒ ('b × nat ⇒ 'a list set) ⇒ 't ⇒ bool  
**and** *abs* :: ('a fo\_term) list ⇒ 'a table ⇒ 't  
**and** *rep* :: 't ⇒ 'a table  
**and** *res* :: 't ⇒ 'a eval\_res  
**and** *eval\_bool* :: bool ⇒ 't  
**and** *eval\_eq* :: 'a fo\_term ⇒ 'a fo\_term ⇒ 't  
**and** *eval\_neg* :: nat list ⇒ 't ⇒ 't  
**and** *eval\_conj* :: nat list ⇒ 't ⇒ nat list ⇒ 't ⇒ 't  
**and** *eval\_ajoin* :: nat list ⇒ 't ⇒ nat list ⇒ 't ⇒ 't

```

and eval_disj :: nat list ⇒ 't ⇒ nat list ⇒ 't ⇒ 't
and eval_exists :: nat ⇒ nat list ⇒ 't ⇒ 't
and eval_forall :: nat ⇒ nat list ⇒ 't ⇒ 't
assumes fo_rep: wf φ I t ⇒ rep t = proj_sat φ I
and fo_res_fin: wf φ I t ⇒ finite (rep t) ⇒ res t = Fin (rep t)
and fo_res_infin: wf φ I t ⇒ ¬finite (rep t) ⇒ res t = Infin
and fo_abs: finite (I (r, length ts)) ⇒ wf (Pred r ts) I (abs ts (I (r, length ts)))
and fo_bool: wf (Bool b) I (eval_bool b)
and fo_eq: wf (Eq trm trm') I (eval_eq trm trm')
and fo_neg: wf φ I t ⇒ wf (Neg φ) I (eval_neg (fv_fo_fmula_list φ) t)
and fo_conj: wf φ I tφ ⇒ wf ψ I tψ ⇒ (case ψ of Neg ψ' ⇒ False | _ ⇒ True) ⇒
  wf (Conj φ ψ) I (eval_conj (fv_fo_fmula_list φ) tφ (fv_fo_fmula_list ψ) tψ)
and fo_ajoin: wf φ I tφ ⇒ wf ψ' I tψ' ⇒
  wf (Conj φ (Neg ψ')) I (eval_ajoin (fv_fo_fmula_list φ) tφ (fv_fo_fmula_list ψ') tψ')
and fo_disj: wf φ I tφ ⇒ wf ψ I tψ ⇒
  wf (Disj φ ψ) I (eval_disj (fv_fo_fmula_list φ) tφ (fv_fo_fmula_list ψ) tψ)
and fo_exists: wf φ I t ⇒ wf (Exists i φ) I (eval_exists i (fv_fo_fmula_list φ) t)
and fo_forall: wf φ I t ⇒ wf (Forall i φ) I (eval_forall i (fv_fo_fmula_list φ) t)
begin

```

```

fun eval_fmula :: ('a, 'b) fo_fmula ⇒ ('a table, 'b) fo_intp ⇒ 't where
  eval_fmula (Pred r ts) I = abs ts (I (r, length ts))
| eval_fmula (Bool b) I = eval_bool b
| eval_fmula (Eq t t') I = eval_eq t t'
| eval_fmula (Neg φ) I = eval_neg (fv_fo_fmula_list φ) (eval_fmula φ I)
| eval_fmula (Conj φ ψ) I = (let nsφ = fv_fo_fmula_list φ; nsψ = fv_fo_fmula_list ψ;
  Xφ = eval_fmula φ I in
  case ψ of Neg ψ' ⇒ let Xψ' = eval_fmula ψ' I in
    eval_ajoin nsφ Xφ (fv_fo_fmula_list ψ') Xψ'
  | _ ⇒ eval_conj nsφ Xφ nsψ (eval_fmula ψ I))
| eval_fmula (Disj φ ψ) I = eval_disj (fv_fo_fmula_list φ) (eval_fmula φ I)
  (fv_fo_fmula_list ψ) (eval_fmula ψ I)
| eval_fmula (Exists i φ) I = eval_exists i (fv_fo_fmula_list φ) (eval_fmula φ I)
| eval_fmula (Forall i φ) I = eval_forall i (fv_fo_fmula_list φ) (eval_fmula φ I)

```

```

lemma eval_fmula_correct:
  fixes φ :: ('a :: infinite, 'b) fo_fmula
  assumes wf_fo_intp φ I
  shows wf φ I (eval_fmula φ I)
  <proof>

```

```

definition eval :: ('a, 'b) fo_fmula ⇒ ('a table, 'b) fo_intp ⇒ 'a eval_res where
  eval φ I = (if wf_fo_intp φ I then res (eval_fmula φ I) else Wf_error)

```

```

lemma eval_fmula_proj_sat:
  fixes φ :: ('a :: infinite, 'b) fo_fmula
  assumes wf_fo_intp φ I
  shows rep (eval_fmula φ I) = proj_sat φ I
  <proof>

```

```

lemma eval_sound:
  fixes φ :: ('a :: infinite, 'b) fo_fmula
  assumes eval φ I = Fin Z
  shows Z = proj_sat φ I
  <proof>

```

```

lemma eval_complete:
  fixes φ :: ('a :: infinite, 'b) fo_fmula

```

```

assumes eval  $\varphi$  I = Infn
shows infinite (proj_sat  $\varphi$  I)
⟨proof⟩

end

end
theory Mapping_Code
imports Containers.Mapping_Impl
begin

lift_definition set_of_idx :: ('a, 'b set) mapping  $\Rightarrow$  'b set is
   $\lambda m. \bigcup (\text{ran } m)$  ⟨proof⟩

lemma set_of_idx_code[code]:
fixes t :: ('a :: ccompare, 'b set) mapping_rbt
shows set_of_idx (RBT_Mapping t) =
  (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "set_of_idx RBT_Mapping: ccompare = None")
  | Some _  $\Rightarrow \bigcup (\text{snd } \text{' set (RBT_Mapping2.entries t)})$ )
  ⟨proof⟩

lemma mapping_combine[code]:
fixes t :: ('a :: ccompare, 'b) mapping_rbt
shows Mapping.combine f (RBT_Mapping t) (RBT_Mapping u) =
  (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "combine RBT_Mapping: ccompare = None")
  | Some _  $\Rightarrow$  RBT_Mapping (RBT_Mapping2.join ( $\lambda_. f$ ) t u))
  ⟨proof⟩

lift_definition mapping_join :: ('b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b)
  mapping is
   $\lambda f m m' x. \text{case } m \ x \text{ of None } \Rightarrow \text{None} \mid \text{Some } y \Rightarrow (\text{case } m' \ x \text{ of None } \Rightarrow \text{None} \mid \text{Some } y' \Rightarrow \text{Some } (f \ y \ y'))$ 
  ⟨proof⟩

lemma mapping_join_code[code]:
fixes t :: ('a :: ccompare, 'b) mapping_rbt
shows mapping_join f (RBT_Mapping t) (RBT_Mapping u) =
  (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "mapping_join RBT_Mapping: ccompare = None")
  | Some _  $\Rightarrow$  RBT_Mapping (RBT_Mapping2.meet ( $\lambda_. f$ ) t u))
  ⟨proof⟩

context fixes dummy :: 'a :: ccompare begin

lift_definition diff ::
  ('a, 'b) mapping_rbt  $\Rightarrow$  ('a, 'b) mapping_rbt  $\Rightarrow$  ('a, 'b) mapping_rbt is rbt_comp_minus ccomp
  ⟨proof⟩

end

context assumes ID_ccompare_neq_None: ID CCOMPARE('a :: ccompare)  $\neq$  None
begin

lemma lookup_diff:
  RBT_Mapping2.lookup (diff (t1 :: ('a, 'b) mapping_rbt) t2) =
  ( $\lambda k. \text{case } \text{RBT\_Mapping2.lookup } t1 \ k \text{ of None } \Rightarrow \text{None} \mid \text{Some } v1 \Rightarrow (\text{case } \text{RBT\_Mapping2.lookup } t2 \ k \text{ of None } \Rightarrow \text{Some } v1 \mid \text{Some } v2 \Rightarrow \text{None})$ )

```

$\langle \text{proof} \rangle$

**end**

**lift\_definition** *mapping\_antijoin* :: ('a, 'b) *mapping*  $\Rightarrow$  ('a, 'b) *mapping*  $\Rightarrow$  ('a, 'b) *mapping* **is**  
 $\lambda m m' x. \text{case } m \ x \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } y \Rightarrow (\text{case } m' \ x \text{ of } \text{None} \Rightarrow \text{Some } y \mid \text{Some } y' \Rightarrow \text{None})$   
 $\langle \text{proof} \rangle$

**lemma** *mapping\_antijoin\_code*[code]:  
**fixes** *t* :: ('a :: *compare*, 'b) *mapping\_rbt*  
**shows** *mapping\_antijoin* (*RBT\_Mapping* *t*) (*RBT\_Mapping* *u*) =  
 $(\text{case } \text{ID } \text{CCOMPARE}('a) \text{ of } \text{None} \Rightarrow \text{Code.abort } (\text{STR } "mapping\_antijoin \text{RBT\_Mapping}: \text{compare} = \text{None}")) (\lambda \_ . \text{mapping\_antijoin } (\text{RBT\_Mapping } t) (\text{RBT\_Mapping } u))$   
 $\mid \text{Some } \_ \Rightarrow \text{RBT\_Mapping } (\text{diff } t \ u)$   
 $\langle \text{proof} \rangle$

**end**  
**theory** *Cluster*  
**imports** *Mapping\_Code*  
**begin**

**lemma** *these\_Un[simp]*: *Option.these* ( $A \cup B$ ) = *Option.these* *A*  $\cup$  *Option.these* *B*  
 $\langle \text{proof} \rangle$

**lemma** *these\_insert[simp]*: *Option.these* (*insert* *x* *A*) = (*case* *x* *of* *Some* *a*  $\Rightarrow$  *insert* *a*  $\mid$  *None*  $\Rightarrow$  *id*)  
(*Option.these* *A*)  
 $\langle \text{proof} \rangle$

**lemma** *these\_image\_Un[simp]*: *Option.these* ( $f \ ' (A \cup B)$ ) = *Option.these* ( $f \ ' A$ )  $\cup$  *Option.these* ( $f \ ' B$ )  
 $\langle \text{proof} \rangle$

**lemma** *these\_imageI*:  $f \ x = \text{Some } y \Longrightarrow x \in X \Longrightarrow y \in \text{Option.these } (f \ ' X)$   
 $\langle \text{proof} \rangle$

**lift\_definition** *cluster* :: ('b  $\Rightarrow$  'a *option*)  $\Rightarrow$  'b *set*  $\Rightarrow$  ('a, 'b *set*) *mapping* **is**  
 $\lambda f \ Y \ x. \text{if } \text{Some } x \in f \ ' Y \text{ then } \text{Some } \{y \in Y. f \ y = \text{Some } x\} \text{ else } \text{None}$   $\langle \text{proof} \rangle$

**lemma** *set\_of\_idx\_cluster*: *set\_of\_idx* (*cluster* (*Some*  $\circ$  *f*) *X*) = *X*  
 $\langle \text{proof} \rangle$

**lemma** *lookup\_cluster'*: *Mapping.lookup* (*cluster* (*Some*  $\circ$  *h*) *X*) *y* = (*if* *y*  $\notin$   $h \ ' X$  *then* *None* *else* *Some*  $\{x \in X. h \ x = y\}$ )  
 $\langle \text{proof} \rangle$

**context** *ord*  
**begin**

**definition** *add\_to\_rbt* :: 'a  $\times$  'b  $\Rightarrow$  ('a, 'b *set*) *rbt*  $\Rightarrow$  ('a, 'b *set*) *rbt* **where**  
 $\text{add\_to\_rbt} = (\lambda(a, b) \ t. \text{case } \text{rbt\_lookup } t \ a \text{ of } \text{Some } X \Rightarrow \text{rbt\_insert } a \ (\text{insert } b \ X) \ t \mid \text{None} \Rightarrow \text{rbt\_insert } a \ \{b\} \ t)$

**abbreviation** *add\_option\_to\_rbt* *f*  $\equiv$  ( $\lambda b \ \_ \ t. \text{case } f \ b \text{ of } \text{Some } a \Rightarrow \text{add\_to\_rbt } (a, b) \ t \mid \text{None} \Rightarrow t$ )

**definition** *cluster\_rbt* :: ('b  $\Rightarrow$  'a *option*)  $\Rightarrow$  ('b, *unit*) *rbt*  $\Rightarrow$  ('a, 'b *set*) *rbt* **where**  
 $\text{cluster\_rbt } f \ t = \text{RBT\_Impl.fold } (\text{add\_option\_to\_rbt } f) \ t \ \text{RBT\_Impl.Empty}$

**end**

**context** *linorder*

**begin**

**lemma** *is\_rbt\_add\_to\_rbt*:  $is\_rbt\ t \implies is\_rbt\ (add\_to\_rbt\ ab\ t)$

*<proof>*

**lemma** *is\_rbt\_fold\_add\_to\_rbt*:  $is\_rbt\ t' \implies$

$is\_rbt\ (RBT\_Impl.fold\ (add\_option\_to\_rbt\ f)\ t\ t')$

*<proof>*

**lemma** *is\_rbt\_cluster\_rbt*:  $is\_rbt\ (cluster\_rbt\ f\ t)$

*<proof>*

**lemma** *rbt\_insert\_entries\_None*:  $is\_rbt\ t \implies rbt\_lookup\ t\ k = None \implies$

$set\ (RBT\_Impl.entries\ (rbt\_insert\ k\ v\ t)) = insert\ (k, v)\ (set\ (RBT\_Impl.entries\ t))$

*<proof>*

**lemma** *rbt\_insert\_entries\_Some*:  $is\_rbt\ t \implies rbt\_lookup\ t\ k = Some\ v' \implies$

$set\ (RBT\_Impl.entries\ (rbt\_insert\ k\ v\ t)) = insert\ (k, v)\ (set\ (RBT\_Impl.entries\ t) - \{(k, v')\})$

*<proof>*

**lemma** *keys\_add\_to\_rbt*:  $is\_rbt\ t \implies set\ (RBT\_Impl.keys\ (add\_to\_rbt\ (a, b)\ t)) = insert\ a\ (set\ (RBT\_Impl.keys\ t))$

*<proof>*

**lemma** *keys\_fold\_add\_to\_rbt*:  $is\_rbt\ t' \implies set\ (RBT\_Impl.keys\ (RBT\_Impl.fold\ (add\_option\_to\_rbt\ f)\ t\ t')) =$

$Option.these\ (f\ 'set\ (RBT\_Impl.keys\ t)) \cup set\ (RBT\_Impl.keys\ t')$

*<proof>*

**lemma** *rbt\_lookup\_add\_to\_rbt*:  $is\_rbt\ t \implies rbt\_lookup\ (add\_to\_rbt\ (a, b)\ t)\ x = (if\ a = x\ then\ Some\ (case\ rbt\_lookup\ t\ x\ of\ None \Rightarrow \{b\} \mid Some\ Y \Rightarrow insert\ b\ Y)\ else\ rbt\_lookup\ t\ x)$

*<proof>*

**lemma** *rbt\_lookup\_fold\_add\_to\_rbt*:  $is\_rbt\ t' \implies rbt\_lookup\ (RBT\_Impl.fold\ (add\_option\_to\_rbt\ f)\ t\ t')\ x =$

$(if\ x \in Option.these\ (f\ 'set\ (RBT\_Impl.keys\ t)) \cup set\ (RBT\_Impl.keys\ t')\ then\ Some\ (\{y \in set\ (RBT\_Impl.keys\ t). f\ y = Some\ x\}$

$\cup (case\ rbt\_lookup\ t'\ x\ of\ None \Rightarrow \{\} \mid Some\ Y \Rightarrow Y))\ else\ None)$

*<proof>*

**end**

**context**

**fixes**  $c :: 'a\ comparator$

**begin**

**definition** *add\_to\_rbt\_comp* ::  $'a \times 'b \Rightarrow ('a, 'b\ set)\ rbt \Rightarrow ('a, 'b\ set)\ rbt$  **where**

$add\_to\_rbt\_comp = (\lambda(a, b)\ t.\ case\ rbt\_comp\_lookup\ c\ t\ a\ of\ None \Rightarrow rbt\_comp\_insert\ c\ a\ \{b\}\ t$   
 $\mid Some\ X \Rightarrow rbt\_comp\_insert\ c\ a\ (insert\ b\ X)\ t)$

**abbreviation** *add\_option\_to\_rbt\_comp*  $f \equiv (\lambda b\ _\ t.\ case\ f\ b\ of\ Some\ a \Rightarrow add\_to\_rbt\_comp\ (a, b)\ t$   
 $\mid None \Rightarrow t)$

**definition** *cluster\_rbt\_comp* ::  $('b \Rightarrow 'a\ option) \Rightarrow ('b, unit)\ rbt \Rightarrow ('a, 'b\ set)\ rbt$  **where**

$cluster\_rbt\_comp\ f\ t = RBT\_Impl.fold\ (add\_option\_to\_rbt\_comp\ f)\ t\ RBT\_Impl.Empty$

**context**

```

assumes c: comparator c
begin

lemma add_to_rbt_comp: add_to_rbt_comp = ord.add_to_rbt (lt_of_comp c)
  <proof>

lemma cluster_rbt_comp: cluster_rbt_comp = ord.cluster_rbt (lt_of_comp c)
  <proof>

end

end

lift_definition mapping_of_cluster :: ('b ⇒ 'a :: ccompare option) ⇒ ('b, unit) rbt ⇒ ('a, 'b set)
mapping_rbt is
  cluster_rbt_comp ccomp
  <proof>

lemma cluster_code[code]:
  fixes f :: 'b :: ccompare ⇒ 'a :: ccompare option and t :: ('b, unit) mapping_rbt
  shows cluster f (RBT_set t) = (case ID CCOMPARE('a) of None ⇒
    Code.abort (STR "cluster: ccompare = None") (λ_. cluster f (RBT_set t))
  | Some c ⇒ (case ID CCOMPARE('b) of None ⇒
    Code.abort (STR "cluster: ccompare = None") (λ_. cluster f (RBT_set t))
  | Some c' ⇒ (RBT_Mapping (mapping_of_cluster f (RBT_Mapping2.impl_of t))))))
  <proof>

end

theory Ailamazyan
  imports Eval_FO Cluster Mapping_Code
begin

fun SP :: ('a, 'b) fo_fm ⇒ nat set where
  SP (Eqa (Var n) (Var n')) = (if n ≠ n' then {n, n'} else {})
  | SP (Neg φ) = SP φ
  | SP (Conj φ ψ) = SP φ ∪ SP ψ
  | SP (Disj φ ψ) = SP φ ∪ SP ψ
  | SP (Exists n φ) = SP φ - {n}
  | SP (Forall n φ) = SP φ - {n}
  | SP _ = {}

lemma SP_fv: SP φ ⊆ fv_fm φ
  <proof>

lemma finite_SP: finite (SP φ)
  <proof>

fun SP_list_rec :: ('a, 'b) fo_fm ⇒ nat list where
  SP_list_rec (Eqa (Var n) (Var n')) = (if n ≠ n' then [n, n'] else [])
  | SP_list_rec (Neg φ) = SP_list_rec φ
  | SP_list_rec (Conj φ ψ) = SP_list_rec φ @ SP_list_rec ψ
  | SP_list_rec (Disj φ ψ) = SP_list_rec φ @ SP_list_rec ψ
  | SP_list_rec (Exists n φ) = filter (λm. n ≠ m) (SP_list_rec φ)
  | SP_list_rec (Forall n φ) = filter (λm. n ≠ m) (SP_list_rec φ)
  | SP_list_rec _ = []

definition SP_list :: ('a, 'b) fo_fm ⇒ nat list where
  SP_list φ = remdups_adj (sort (SP_list_rec φ))

```

**lemma** *SP\_list\_set*:  $set (SP\_list \varphi) = SP \varphi$   
 ⟨proof⟩

**lemma** *sorted\_distinct\_SP\_list*:  $sorted\_distinct (SP\_list \varphi)$   
 ⟨proof⟩

**fun** *d* :: ('a, 'b) fo\_fmula  $\Rightarrow$  nat **where**  
 | *d* (Eqn (Var n) (Var n')) = (if n  $\neq$  n' then 2 else 1)  
 | *d* (Neg  $\varphi$ ) = *d*  $\varphi$   
 | *d* (Conj  $\varphi \psi$ ) = max (*d*  $\varphi$ ) (max (*d*  $\psi$ ) (card (SP (Conj  $\varphi \psi$ ))))  
 | *d* (Disj  $\varphi \psi$ ) = max (*d*  $\varphi$ ) (max (*d*  $\psi$ ) (card (SP (Disj  $\varphi \psi$ ))))  
 | *d* (Exists n  $\varphi$ ) = *d*  $\varphi$   
 | *d* (Forall n  $\varphi$ ) = *d*  $\varphi$   
 | *d* \_ = 1

**lemma** *d\_pos*:  $1 \leq d \varphi$   
 ⟨proof⟩

**lemma** *card\_SP\_d*:  $card (SP \varphi) \leq d \varphi$   
 ⟨proof⟩

**fun** *eval\_eterm* :: ('a + 'c) val  $\Rightarrow$  'a fo\_term  $\Rightarrow$  'a + 'c (**infix** <·e> 60) **where**  
 | *eval\_eterm*  $\sigma$  (Const c) = Inl c  
 | *eval\_eterm*  $\sigma$  (Var n) =  $\sigma$  n

**definition** *eval\_eterms* :: ('a + 'c) val  $\Rightarrow$  ('a fo\_term) list  $\Rightarrow$   
 ('a + 'c) list (**infix** <⊙e> 60) **where**  
 | *eval\_eterms*  $\sigma$  ts = map (*eval\_eterm*  $\sigma$ ) ts

**lemma** *eval\_eterm\_cong*:  $(\bigwedge n. n \in fv\_fo\_term\_set t \Rightarrow \sigma n = \sigma' n) \Rightarrow$   
 $eval\_eterm \sigma t = eval\_eterm \sigma' t$   
 ⟨proof⟩

**lemma** *eval\_eterms\_fv\_fo\_terms\_set*:  $\sigma \odot e ts = \sigma' \odot e ts \Rightarrow n \in fv\_fo\_terms\_set ts \Rightarrow \sigma n = \sigma' n$   
 ⟨proof⟩

**lemma** *eval\_eterms\_cong*:  $(\bigwedge n. n \in fv\_fo\_terms\_set ts \Rightarrow \sigma n = \sigma' n) \Rightarrow$   
 $eval\_eterms \sigma ts = eval\_eterms \sigma' ts$   
 ⟨proof⟩

**lemma** *eval\_terms\_eterms*:  $map Inl (\sigma \odot ts) = (Inl \circ \sigma) \odot e ts$   
 ⟨proof⟩

**fun** *ad\_equiv\_pair* :: 'a set  $\Rightarrow$  ('a + 'c)  $\times$  ('a + 'c)  $\Rightarrow$  bool **where**  
 | *ad\_equiv\_pair* X (a, a')  $\longleftrightarrow (a \in Inl ' X \rightarrow a = a') \wedge (a' \in Inl ' X \rightarrow a = a')$

**fun** *sp\_equiv\_pair* :: 'a  $\times$  'b  $\Rightarrow$  'a  $\times$  'b  $\Rightarrow$  bool **where**  
 | *sp\_equiv\_pair* (a, b) (a', b')  $\longleftrightarrow (a = a' \longleftrightarrow b = b')$

**definition** *ad\_equiv\_list* :: 'a set  $\Rightarrow$  ('a + 'c) list  $\Rightarrow$  ('a + 'c) list  $\Rightarrow$  bool **where**  
 | *ad\_equiv\_list* X xs ys  $\longleftrightarrow length xs = length ys \wedge (\forall x \in set (zip xs ys). ad\_equiv\_pair X x)$

**definition** *sp\_equiv\_list* :: ('a + 'c) list  $\Rightarrow$  ('a + 'c) list  $\Rightarrow$  bool **where**  
 | *sp\_equiv\_list* xs ys  $\longleftrightarrow length xs = length ys \wedge pairwise sp\_equiv\_pair (set (zip xs ys))$

**definition** *ad\_agr\_list* :: 'a set  $\Rightarrow$  ('a + 'c) list  $\Rightarrow$  ('a + 'c) list  $\Rightarrow$  bool **where**  
 | *ad\_agr\_list* X xs ys  $\longleftrightarrow length xs = length ys \wedge ad\_equiv\_list X xs ys \wedge sp\_equiv\_list xs ys$

**lemma** *ad\_equiv\_pair\_refl*[simp]: *ad\_equiv\_pair* *X* (*a*, *a*)  
 ⟨*proof*⟩

**declare** *ad\_equiv\_pair.simps*[simp del]

**lemma** *ad\_equiv\_pair\_comm*: *ad\_equiv\_pair* *X* (*a*, *a'*)  $\longleftrightarrow$  *ad\_equiv\_pair* *X* (*a'*, *a*)  
 ⟨*proof*⟩

**lemma** *ad\_equiv\_pair\_mono*:  $X \subseteq Y \implies \text{ad\_equiv\_pair } Y (a, a') \implies \text{ad\_equiv\_pair } X (a, a')$   
 ⟨*proof*⟩

**lemma** *sp\_equiv\_pair\_comm*: *sp\_equiv\_pair* *x y*  $\longleftrightarrow$  *sp\_equiv\_pair* *y x*  
 ⟨*proof*⟩

**definition** *sp\_equiv* :: ('*a* + '*c*) *val*  $\Rightarrow$  ('*a* + '*c*) *val*  $\Rightarrow$  *nat set*  $\Rightarrow$  *bool* **where**  
*sp\_equiv*  $\sigma \tau I \longleftrightarrow \text{pairwise } \text{sp\_equiv\_pair } ((\lambda n. (\sigma n, \tau n)) \text{ ` } I)$

**lemma** *sp\_equiv\_mono*:  $I \subseteq J \implies \text{sp\_equiv } \sigma \tau J \implies \text{sp\_equiv } \sigma \tau I$   
 ⟨*proof*⟩

**definition** *ad\_agr\_sets* :: *nat set*  $\Rightarrow$  *nat set*  $\Rightarrow$  '*a set*  $\Rightarrow$  ('*a* + '*c*) *val*  $\Rightarrow$   
 ('*a* + '*c*) *val*  $\Rightarrow$  *bool* **where**  
*ad\_agr\_sets* *FV S X*  $\sigma \tau \longleftrightarrow (\forall i \in \text{FV}. \text{ad\_equiv\_pair } X (\sigma i, \tau i)) \wedge \text{sp\_equiv } \sigma \tau S$

**lemma** *ad\_agr\_sets\_comm*: *ad\_agr\_sets* *FV S X*  $\sigma \tau \implies \text{ad\_agr\_sets } \text{FV } S X \tau \sigma$   
 ⟨*proof*⟩

**lemma** *ad\_agr\_sets\_mono*:  $X \subseteq Y \implies \text{ad\_agr\_sets } \text{FV } S Y \sigma \tau \implies \text{ad\_agr\_sets } \text{FV } S X \sigma \tau$   
 ⟨*proof*⟩

**lemma** *ad\_agr\_sets\_mono'*:  $S \subseteq S' \implies \text{ad\_agr\_sets } \text{FV } S' X \sigma \tau \implies \text{ad\_agr\_sets } \text{FV } S X \sigma \tau$   
 ⟨*proof*⟩

**lemma** *ad\_equiv\_list\_comm*: *ad\_equiv\_list* *X xs ys*  $\implies \text{ad\_equiv\_list } X ys xs$   
 ⟨*proof*⟩

**lemma** *ad\_equiv\_list\_mono*:  $X \subseteq Y \implies \text{ad\_equiv\_list } Y xs ys \implies \text{ad\_equiv\_list } X xs ys$   
 ⟨*proof*⟩

**lemma** *ad\_equiv\_list\_trans*:  
**assumes** *ad\_equiv\_list* *X xs ys* *ad\_equiv\_list* *X ys zs*  
**shows** *ad\_equiv\_list* *X xs zs*  
 ⟨*proof*⟩

**lemma** *ad\_equiv\_list\_link*:  $(\forall i \in \text{set } ns. \text{ad\_equiv\_pair } X (\sigma i, \tau i)) \longleftrightarrow$   
*ad\_equiv\_list* *X (map*  $\sigma$  *ns) (map*  $\tau$  *ns)*  
 ⟨*proof*⟩

**lemma** *set\_zip\_comm*:  $(x, y) \in \text{set } (\text{zip } xs ys) \implies (y, x) \in \text{set } (\text{zip } ys xs)$   
 ⟨*proof*⟩

**lemma** *set\_zip\_map*:  $\text{set } (\text{zip } (\text{map } \sigma ns) (\text{map } \tau ns)) = (\lambda n. (\sigma n, \tau n)) \text{ ` set } ns$   
 ⟨*proof*⟩

**lemma** *sp\_equiv\_list\_comm*: *sp\_equiv\_list* *xs ys*  $\implies \text{sp\_equiv\_list } ys xs$   
 ⟨*proof*⟩

**lemma** *sp\_equiv\_list\_trans*:

**assumes** *sp\_equiv\_list xs ys sp\_equiv\_list ys zs*

**shows** *sp\_equiv\_list xs zs*

*<proof>*

**lemma** *sp\_equiv\_list\_link*: *sp\_equiv\_list (map  $\sigma$  ns) (map  $\tau$  ns)  $\longleftrightarrow$  sp\_equiv  $\sigma$   $\tau$  (set ns)*

*<proof>*

**lemma** *ad\_agr\_list\_comm*: *ad\_agr\_list X xs ys  $\implies$  ad\_agr\_list X ys xs*

*<proof>*

**lemma** *ad\_agr\_list\_mono*: *X  $\subseteq$  Y  $\implies$  ad\_agr\_list Y ys xs  $\implies$  ad\_agr\_list X ys xs*

*<proof>*

**lemma** *ad\_agr\_list\_rev\_mono*:

**assumes** *Y  $\subseteq$  X ad\_agr\_list Y ys xs Inl -' set xs  $\subseteq$  Y Inl -' set ys  $\subseteq$  Y*

**shows** *ad\_agr\_list X ys xs*

*<proof>*

**lemma** *ad\_agr\_list\_trans*: *ad\_agr\_list X xs ys  $\implies$  ad\_agr\_list X ys zs  $\implies$  ad\_agr\_list X xs zs*

*<proof>*

**lemma** *ad\_agr\_list\_refl*: *ad\_agr\_list X xs xs*

*<proof>*

**lemma** *ad\_agr\_list\_set*: *ad\_agr\_list X xs ys  $\implies$  y  $\in$  X  $\implies$  Inl y  $\in$  set ys  $\implies$  Inl y  $\in$  set xs*

*<proof>*

**lemma** *ad\_agr\_list\_length*: *ad\_agr\_list X xs ys  $\implies$  length xs = length ys*

*<proof>*

**lemma** *ad\_agr\_list\_eq*: *set ys  $\subseteq$  AD  $\implies$  ad\_agr\_list AD (map Inl xs) (map Inl ys)  $\implies$  xs = ys*

*<proof>*

**lemma** *sp\_equiv\_list\_subset*:

**assumes** *set ms  $\subseteq$  set ns sp\_equiv\_list (map  $\sigma$  ns) (map  $\sigma'$  ns)*

**shows** *sp\_equiv\_list (map  $\sigma$  ms) (map  $\sigma'$  ms)*

*<proof>*

**lemma** *ad\_agr\_list\_subset*: *set ms  $\subseteq$  set ns  $\implies$  ad\_agr\_list X (map  $\sigma$  ns) (map  $\sigma'$  ns)  $\implies$*

*ad\_agr\_list X (map  $\sigma$  ms) (map  $\sigma'$  ms)*

*<proof>*

**lemma** *ad\_agr\_list\_link*: *ad\_agr\_sets (set ns) (set ns) AD  $\sigma$   $\tau$   $\longleftrightarrow$*

*ad\_agr\_list AD (map  $\sigma$  ns) (map  $\tau$  ns)*

*<proof>*

**definition** *ad\_agr* :: *('a, 'b) fo\_fm $\bar{a}$   $\Rightarrow$  'a set  $\Rightarrow$  ('a + 'c) val  $\Rightarrow$  ('a + 'c) val  $\Rightarrow$  bool* **where**

*ad\_agr  $\varphi$  X  $\sigma$   $\tau$   $\longleftrightarrow$  ad\_agr\_sets (fv\_fm $\bar{a}$   $\varphi$ ) (SP  $\varphi$ ) X  $\sigma$   $\tau$*

**lemma** *ad\_agr\_sets\_restrict*:

*ad\_agr\_sets (set (fv\_fm $\bar{a}$  list  $\varphi$ )) (set (fv\_fm $\bar{a}$  list  $\varphi$ )) AD  $\sigma$   $\tau$   $\implies$  ad\_agr  $\varphi$  AD  $\sigma$   $\tau$*

*<proof>*

**lemma** *finite\_Inl*: *finite X  $\implies$  finite (Inl -' X)*

*<proof>*

**lemma** *ex\_out*:

**assumes** *finite X*  
**shows**  $\exists k. k \notin X \wedge k < \text{Suc}(\text{card } X)$   
 <proof>

**lemma** *extend\_τ*:

**assumes**  $\text{ad\_agr\_sets } (FV - \{n\}) (S - \{n\}) X \sigma \tau S \subseteq FV \text{ finite } S \tau ' (FV - \{n\}) \subseteq Z$   
 $\text{Inl}' X \cup \text{Inr}' \{.. < \max 1 (\text{card } (\text{Inr}' \tau ' (S - \{n\})) + (\text{if } n \in S \text{ then } 1 \text{ else } 0))\} \subseteq Z$   
**shows**  $\exists k \in Z. \text{ad\_agr\_sets } FV S X (\sigma(n := x)) (\tau(n := k))$   
 <proof>

**lemma** *esat\_Pred*:

**assumes**  $\text{ad\_agr\_sets } FV S (\bigcup (\text{set}' X)) \sigma \tau \text{fv\_fo\_terms\_set } ts \subseteq FV \sigma \odot e \text{ ts} \in \text{map } \text{Inl}' X$   
 $t \in \text{set } ts$   
**shows**  $\sigma \cdot e \ t = \tau \cdot e \ t$   
 <proof>

**lemma** *sp\_equiv\_list\_fv*:

**assumes**  $(\bigwedge i. i \in \text{fv\_fo\_terms\_set } ts \implies \text{ad\_equiv\_pair } X (\sigma \ i, \tau \ i))$   
 $\bigcup (\text{set\_fo\_term}' \text{ set } ts) \subseteq X \text{ sp\_equiv } \sigma \tau (\text{fv\_fo\_terms\_set } ts)$   
**shows**  $\text{sp\_equiv\_list } (\text{map } ((\cdot e) \ \sigma) \ ts) (\text{map } ((\cdot e) \ \tau) \ ts)$   
 <proof>

**lemma** *esat\_Pred\_inf*:

**assumes**  $\text{fv\_fo\_terms\_set } ts \subseteq FV \text{fv\_fo\_terms\_set } ts \subseteq S$   
 $\text{ad\_agr\_sets } FV S AD \sigma \tau \text{ad\_agr\_list } AD (\sigma \odot e \ ts) \text{ vs}$   
 $\bigcup (\text{set\_fo\_term}' \text{ set } ts) \subseteq AD$   
**shows**  $\text{ad\_agr\_list } AD (\tau \odot e \ ts) \text{ vs}$   
 <proof>

**type\_synonym** ('a, 'c) *fo\_t* = 'a set × nat × ('a + 'c) table

**fun** *esat* :: ('a, 'b) *fo\_fm* ⇒ ('a table, 'b) *fo\_intp* ⇒ ('a + nat) *val* ⇒ ('a + nat) *set* ⇒ *bool* **where**

*esat* (*Pred r ts*) *I*  $\sigma$  *X*  $\longleftrightarrow \sigma \odot e \ ts \in \text{map } \text{Inl}' I (r, \text{length } ts)$   
 | *esat* (*Bool b*) *I*  $\sigma$  *X*  $\longleftrightarrow b$   
 | *esat* (*Eqa t t'*) *I*  $\sigma$  *X*  $\longleftrightarrow \sigma \cdot e \ t = \sigma \cdot e \ t'$   
 | *esat* (*Neg φ*) *I*  $\sigma$  *X*  $\longleftrightarrow \neg \text{esat } \varphi \ I \ \sigma \ X$   
 | *esat* (*Conj φ ψ*) *I*  $\sigma$  *X*  $\longleftrightarrow \text{esat } \varphi \ I \ \sigma \ X \wedge \text{esat } \psi \ I \ \sigma \ X$   
 | *esat* (*Disj φ ψ*) *I*  $\sigma$  *X*  $\longleftrightarrow \text{esat } \varphi \ I \ \sigma \ X \vee \text{esat } \psi \ I \ \sigma \ X$   
 | *esat* (*Exists n φ*) *I*  $\sigma$  *X*  $\longleftrightarrow (\exists x \in X. \text{esat } \varphi \ I (\sigma(n := x)) \ X)$   
 | *esat* (*Forall n φ*) *I*  $\sigma$  *X*  $\longleftrightarrow (\forall x \in X. \text{esat } \varphi \ I (\sigma(n := x)) \ X)$

**fun** *sz\_fm* :: ('a, 'b) *fo\_fm* ⇒ *nat* **where**

*sz\_fm* (*Neg φ*) = *Suc* (*sz\_fm*  $\varphi$ )  
 | *sz\_fm* (*Conj φ ψ*) = *Suc* (*sz\_fm*  $\varphi$  + *sz\_fm*  $\psi$ )  
 | *sz\_fm* (*Disj φ ψ*) = *Suc* (*sz\_fm*  $\varphi$  + *sz\_fm*  $\psi$ )  
 | *sz\_fm* (*Exists n φ*) = *Suc* (*sz\_fm*  $\varphi$ )  
 | *sz\_fm* (*Forall n φ*) = *Suc* (*Suc* (*Suc* (*Suc* (*sz\_fm*  $\varphi$ ))))  
 | *sz\_fm* \_ = 0

**lemma** *sz\_fm\_induct*[*case\_names Pred Bool Eqa Neg Conj Disj Exists Forall*]:

$(\bigwedge r \ ts. P (\text{Pred } r \ ts)) \implies (\bigwedge b. P (\text{Bool } b)) \implies$   
 $(\bigwedge t \ t'. P (\text{Eqa } t \ t')) \implies (\bigwedge \varphi. P \ \varphi \implies P (\text{Neg } \varphi)) \implies$   
 $(\bigwedge \varphi \ \psi. P \ \varphi \implies P \ \psi \implies P (\text{Conj } \varphi \ \psi)) \implies (\bigwedge \varphi \ \psi. P \ \varphi \implies P \ \psi \implies P (\text{Disj } \varphi \ \psi)) \implies$   
 $(\bigwedge n \ \varphi. P \ \varphi \implies P (\text{Exists } n \ \varphi)) \implies (\bigwedge n \ \varphi. P (\text{Exists } n (\text{Neg } \varphi)) \implies P (\text{Forall } n \ \varphi)) \implies P \ \varphi$   
 <proof>

**lemma** *esat\_fv\_cong*:  $(\bigwedge n. n \in \text{fv\_fo_fm} \ \varphi \implies \sigma \ n = \sigma' \ n) \implies \text{esat } \varphi \ I \ \sigma \ X \longleftrightarrow \text{esat } \varphi \ I \ \sigma' \ X$   
 <proof>

**fun** *ad\_terms* :: ('a fo\_term) list  $\Rightarrow$  'a set **where**  
*ad\_terms* ts =  $\bigcup$ (set (map set\_fo\_term ts))

**fun** *act\_edom* :: ('a, 'b) fo\_fmla  $\Rightarrow$  ('a table, 'b) fo\_intp  $\Rightarrow$  'a set **where**  
*act\_edom* (Pred r ts) I = *ad\_terms* ts  $\cup$   $\bigcup$ (set 'I (r, length ts))  
| *act\_edom* (Bool b) I = {}  
| *act\_edom* (Eqv t t') I = set\_fo\_term t  $\cup$  set\_fo\_term t'  
| *act\_edom* (Neg  $\varphi$ ) I = *act\_edom*  $\varphi$  I  
| *act\_edom* (Conj  $\varphi$   $\psi$ ) I = *act\_edom*  $\varphi$  I  $\cup$  *act\_edom*  $\psi$  I  
| *act\_edom* (Disj  $\varphi$   $\psi$ ) I = *act\_edom*  $\varphi$  I  $\cup$  *act\_edom  $\psi$  I  
| *act\_edom* (Exists n  $\varphi$ ) I = *act\_edom*  $\varphi$  I  
| *act\_edom* (Forall n  $\varphi$ ) I = *act\_edom*  $\varphi$  I*

**lemma** *finite\_act\_edom*: wf\_fo\_intp  $\varphi$  I  $\Longrightarrow$  finite (act\_edom  $\varphi$  I)  
<proof>

**fun** *fo\_adom* :: ('a, 'c) fo\_t  $\Rightarrow$  'a set **where**  
*fo\_adom* (AD, n, X) = AD

**theorem** *main*: *ad\_agr*  $\varphi$  AD  $\sigma$   $\tau$   $\Longrightarrow$  *act\_edom*  $\varphi$  I  $\subseteq$  AD  $\Longrightarrow$   
Inl 'AD  $\cup$  Inr '{.. $d$   $\varphi$ }  $\subseteq$  X  $\Longrightarrow$   $\tau$  'fv\_fo\_fmla  $\varphi$   $\subseteq$  X  $\Longrightarrow$   
esat  $\varphi$  I  $\sigma$  UNIV  $\longleftrightarrow$  esat  $\varphi$  I  $\tau$  X  
<proof>

**lemma** *main\_cor\_inf*:  
**assumes** *ad\_agr*  $\varphi$  AD  $\sigma$   $\tau$  *act\_edom*  $\varphi$  I  $\subseteq$  AD  $d$   $\varphi$   $\leq$  n  
 $\tau$  'fv\_fo\_fmla  $\varphi$   $\subseteq$  Inl 'AD  $\cup$  Inr '{.. $n$ }  
**shows** esat  $\varphi$  I  $\sigma$  UNIV  $\longleftrightarrow$  esat  $\varphi$  I  $\tau$  (Inl 'AD  $\cup$  Inr '{.. $n$ })  
<proof>

**lemma** *esat\_UNIV\_cong*:  
**fixes**  $\sigma$  :: nat  $\Rightarrow$  'a + nat  
**assumes** *ad\_agr*  $\varphi$  AD  $\sigma$   $\tau$  *act\_edom*  $\varphi$  I  $\subseteq$  AD  
**shows** esat  $\varphi$  I  $\sigma$  UNIV  $\longleftrightarrow$  esat  $\varphi$  I  $\tau$  UNIV  
<proof>

**lemma** *esat\_UNIV\_ad\_agr\_list*:  
**fixes**  $\sigma$  :: nat  $\Rightarrow$  'a + nat  
**assumes** *ad\_agr\_list* AD (map  $\sigma$  (fv\_fo\_fmla\_list  $\varphi$ )) (map  $\tau$  (fv\_fo\_fmla\_list  $\varphi$ ))  
*act\_edom*  $\varphi$  I  $\subseteq$  AD  
**shows** esat  $\varphi$  I  $\sigma$  UNIV  $\longleftrightarrow$  esat  $\varphi$  I  $\tau$  UNIV  
<proof>

**fun** *fo\_rep* :: ('a, 'c) fo\_t  $\Rightarrow$  'a table **where**  
*fo\_rep* (AD, n, X) = {ts.  $\exists$  ts'  $\in$  X. *ad\_agr\_list* AD (map Inl ts) ts'}

**lemma** *sat\_esat\_conv*:  
**fixes**  $\varphi$  :: ('a :: infinite, 'b) fo\_fmla  
**assumes** *fin*: wf\_fo\_intp  $\varphi$  I  
**shows** sat  $\varphi$  I  $\sigma$   $\longleftrightarrow$  esat  $\varphi$  I (Inl  $\circ$   $\sigma$  :: nat  $\Rightarrow$  'a + nat) UNIV  
<proof>

**lemma** *sat\_ad\_agr\_list*:  
**fixes**  $\varphi$  :: ('a :: infinite, 'b) fo\_fmla  
**and** J :: (('a, nat) fo\_t, 'b) fo\_intp  
**assumes** wf\_fo\_intp  $\varphi$  I  
*ad\_agr\_list* AD (map (Inl  $\circ$   $\sigma$  :: nat  $\Rightarrow$  'a + nat) (fv\_fo\_fmla\_list  $\varphi$ ))

(map (Inl ∘ τ) (fv\_fo\_fmula\_list φ)) act\_edom φ I ⊆ AD  
**shows** sat φ I σ ↔ sat φ I τ  
 ⟨proof⟩

**definition** nfv :: ('a, 'b) fo\_fmula ⇒ nat **where**  
 nfv φ = length (fv\_fo\_fmula\_list φ)

**lemma** nfv\_card: nfv φ = card (fv\_fo\_fmula φ)  
 ⟨proof⟩

**fun** rremdups :: 'a list ⇒ 'a list **where**  
 rremdups [] = []  
 | rremdups (x # xs) = x # rremdups (filter ((≠) x) xs)

**lemma** filter\_rremdups\_filter: filter P (rremdups (filter Q xs)) =  
 rremdups (filter (λx. P x ∧ Q x) xs)  
 ⟨proof⟩

**lemma** filter\_rremdups: filter P (rremdups xs) = rremdups (filter P xs)  
 ⟨proof⟩

**lemma** filter\_take: ∃j. filter P (take i xs) = take j (filter P xs)  
 ⟨proof⟩

**lemma** rremdups\_take: ∃j. rremdups (take i xs) = take j (rremdups xs)  
 ⟨proof⟩

**lemma** rremdups\_app: rremdups (xs @ [x]) = rremdups xs @ (if x ∈ set xs then [] else [x])  
 ⟨proof⟩

**lemma** rremdups\_set: set (rremdups xs) = set xs  
 ⟨proof⟩

**lemma** distinct\_rremdups: distinct (rremdups xs)  
 ⟨proof⟩

**lemma** length\_rremdups: length (rremdups xs) = card (set xs)  
 ⟨proof⟩

**lemma** set\_map\_filter\_sum: set (List.map\_filter (case\_sum Map.empty Some) xs) = Inr -' set xs  
 ⟨proof⟩

**definition** nats :: nat list ⇒ bool **where**  
 nats ns = (ns = [0..<length ns])

**definition** fo\_nmlzd :: 'a set ⇒ ('a + nat) list ⇒ bool **where**  
 fo\_nmlzd AD xs ↔ Inl -' set xs ⊆ AD ∧  
 (let ns = List.map\_filter (case\_sum Map.empty Some) xs in nats (rremdups ns))

**lemma** fo\_nmlzd\_all\_AD:  
**assumes** set xs ⊆ Inl ' AD  
**shows** fo\_nmlzd AD xs  
 ⟨proof⟩

**lemma** card\_Inr\_vimage\_le\_length: card (Inr -' set xs) ≤ length xs  
 ⟨proof⟩

**lemma** fo\_nmlzd\_set:

**assumes**  $fo\_nmlzd\ AD\ xs$   
**shows**  $set\ xs = set\ xs \cap Inl\ 'AD \cup Inr\ '\{..<min\ (length\ xs)\ (card\ (Inr\ -'\ set\ xs))\}$   
 $\langle proof \rangle$

**lemma**  $map\_filter\_take: \exists j. List.map\_filter\ f\ (take\ i\ xs) = take\ j\ (List.map\_filter\ f\ xs)$   
 $\langle proof \rangle$

**lemma**  $fo\_nmlzd\_take: \text{assumes } fo\_nmlzd\ AD\ xs$   
**shows**  $fo\_nmlzd\ AD\ (take\ i\ xs)$   
 $\langle proof \rangle$

**lemma**  $map\_filter\_app: List.map\_filter\ f\ (xs\ @\ [x]) = List.map\_filter\ f\ xs\ @$   
 $(case\ f\ x\ of\ Some\ y \Rightarrow [y] \mid \_ \Rightarrow [])$   
 $\langle proof \rangle$

**lemma**  $fo\_nmlzd\_app\_Inr: Inr\ n \notin set\ xs \Longrightarrow Inr\ n' \notin set\ xs \Longrightarrow fo\_nmlzd\ AD\ (xs\ @\ [Inr\ n]) \Longrightarrow$   
 $fo\_nmlzd\ AD\ (xs\ @\ [Inr\ n']) \Longrightarrow n = n'$   
 $\langle proof \rangle$

**fun**  $all\_tuples :: 'c\ set \Rightarrow nat \Rightarrow 'c\ table\ \text{where}$   
 $all\_tuples\ xs\ 0 = \{\}\}$   
 $| all\_tuples\ xs\ (Suc\ n) = \bigcup ((\lambda as. (\lambda x. x \# as) '\ xs) '\ (all\_tuples\ xs\ n))$

**definition**  $nall\_tuples :: 'a\ set \Rightarrow nat \Rightarrow ('a + nat)\ table\ \text{where}$   
 $nall\_tuples\ AD\ n = \{zs \in all\_tuples\ (Inl\ 'AD \cup Inr\ '\{..<n\})\ n. fo\_nmlzd\ AD\ zs\}$

**lemma**  $all\_tuples\_finite: finite\ xs \Longrightarrow finite\ (all\_tuples\ xs\ n)$   
 $\langle proof \rangle$

**lemma**  $nall\_tuples\_finite: finite\ AD \Longrightarrow finite\ (nall\_tuples\ AD\ n)$   
 $\langle proof \rangle$

**lemma**  $all\_tuplesI: length\ vs = n \Longrightarrow set\ vs \subseteq xs \Longrightarrow vs \in all\_tuples\ xs\ n$   
 $\langle proof \rangle$

**lemma**  $nall\_tuplesI: length\ vs = n \Longrightarrow fo\_nmlzd\ AD\ vs \Longrightarrow vs \in nall\_tuples\ AD\ n$   
 $\langle proof \rangle$

**lemma**  $all\_tuplesD: vs \in all\_tuples\ xs\ n \Longrightarrow length\ vs = n \wedge set\ vs \subseteq xs$   
 $\langle proof \rangle$

**lemma**  $all\_tuples\_setD: vs \in all\_tuples\ xs\ n \Longrightarrow set\ vs \subseteq xs$   
 $\langle proof \rangle$

**lemma**  $nall\_tuplesD: vs \in nall\_tuples\ AD\ n \Longrightarrow$   
 $length\ vs = n \wedge set\ vs \subseteq Inl\ 'AD \cup Inr\ '\{..<n\} \wedge fo\_nmlzd\ AD\ vs$   
 $\langle proof \rangle$

**lemma**  $all\_tuples\_set: all\_tuples\ xs\ n = \{ys. length\ ys = n \wedge set\ ys \subseteq xs\}$   
 $\langle proof \rangle$

**lemma**  $nall\_tuples\_set: nall\_tuples\ AD\ n = \{ys. length\ ys = n \wedge fo\_nmlzd\ AD\ ys\}$   
 $\langle proof \rangle$

**fun**  $pos :: 'a \Rightarrow 'a\ list \Rightarrow nat\ option\ \text{where}$   
 $pos\ a\ [] = None$   
 $| pos\ a\ (x \# xs) =$   
 $(if\ a = x\ then\ Some\ 0\ else\ (case\ pos\ a\ xs\ of\ Some\ n \Rightarrow Some\ (Suc\ n) \mid \_ \Rightarrow None))$

**lemma** *pos\_set*:  $\text{pos } a \text{ } xs = \text{Some } i \implies a \in \text{set } xs$   
 ⟨proof⟩

**lemma** *pos\_length*:  $\text{pos } a \text{ } xs = \text{Some } i \implies i < \text{length } xs$   
 ⟨proof⟩

**lemma** *pos\_sound*:  $\text{pos } a \text{ } xs = \text{Some } i \implies i < \text{length } xs \wedge xs ! i = a$   
 ⟨proof⟩

**lemma** *pos\_complete*:  $\text{pos } a \text{ } xs = \text{None} \implies a \notin \text{set } xs$   
 ⟨proof⟩

**fun** *rem\_nth* ::  $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
*rem\_nth* \_ [] = []  
 | *rem\_nth* 0 (x # xs) = xs  
 | *rem\_nth* (Suc n) (x # xs) = x # *rem\_nth* n xs

**lemma** *rem\_nth\_length*:  $i < \text{length } xs \implies \text{length } (\text{rem\_nth } i \text{ } xs) = \text{length } xs - 1$   
 ⟨proof⟩

**lemma** *rem\_nth\_take\_drop*:  $i < \text{length } xs \implies \text{rem\_nth } i \text{ } xs = \text{take } i \text{ } xs @ \text{drop } (\text{Suc } i) \text{ } xs$   
 ⟨proof⟩

**lemma** *rem\_nth\_sound*:  $\text{distinct } xs \implies \text{pos } n \text{ } xs = \text{Some } i \implies$   
*rem\_nth* i (map  $\sigma$  xs) = map  $\sigma$  (filter (( $\neq$ ) n) xs)  
 ⟨proof⟩

**fun** *add\_nth* ::  $\text{nat} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
*add\_nth* 0 a xs = a # xs  
 | *add\_nth* (Suc n) a zs = (case zs of x # xs  $\Rightarrow$  x # *add\_nth* n a xs)

**lemma** *add\_nth\_length*:  $i \leq \text{length } zs \implies \text{length } (\text{add\_nth } i \text{ } z \text{ } zs) = \text{Suc } (\text{length } zs)$   
 ⟨proof⟩

**lemma** *add\_nth\_take\_drop*:  $i \leq \text{length } zs \implies \text{add\_nth } i \text{ } v \text{ } zs = \text{take } i \text{ } zs @ v \# \text{drop } i \text{ } zs$   
 ⟨proof⟩

**lemma** *add\_nth\_rem\_nth\_map*:  $\text{distinct } xs \implies \text{pos } n \text{ } xs = \text{Some } i \implies$   
*add\_nth* i a (*rem\_nth* i (map  $\sigma$  xs)) = map ( $\sigma$ (n := a)) xs  
 ⟨proof⟩

**lemma** *add\_nth\_rem\_nth\_self*:  $i < \text{length } xs \implies \text{add\_nth } i \text{ } (xs ! i) \text{ } (\text{rem\_nth } i \text{ } xs) = xs$   
 ⟨proof⟩

**lemma** *rem\_nth\_add\_nth*:  $i \leq \text{length } zs \implies \text{rem\_nth } i \text{ } (\text{add\_nth } i \text{ } z \text{ } zs) = zs$   
 ⟨proof⟩

**fun** *merge* ::  $(\text{nat} \times 'a) \text{ list} \Rightarrow (\text{nat} \times 'a) \text{ list} \Rightarrow (\text{nat} \times 'a) \text{ list}$  **where**  
*merge* [] mys = mys  
 | *merge* nxs [] = nxs  
 | *merge* ((n, x) # nxs) ((m, y) # mys) =  
   (if  $n \leq m$  then (n, x) # *merge* nxs ((m, y) # mys)  
   else (m, y) # *merge* ((n, x) # nxs) mys)

**lemma** *merge\_Nil2*[simp]: *merge* nxs [] = nxs  
 ⟨proof⟩

**lemma** *merge\_length*:  $\text{length } (\text{merge } nxs \text{ mys}) = \text{length } (\text{map } fst \ nxs \ @ \ \text{map } fst \ \text{mys})$   
 ⟨proof⟩

**lemma** *insort\_aux\_le*:  $\forall x \in \text{set } nxs. n \leq fst \ x \implies \forall x \in \text{set } mys. m \leq fst \ x \implies n \leq m \implies$   
 $\text{insort } n \ (\text{sort } (\text{map } fst \ nxs \ @ \ m \ \# \ \text{map } fst \ \text{mys})) = n \ \# \ \text{sort } (\text{map } fst \ nxs \ @ \ m \ \# \ \text{map } fst \ \text{mys})$   
 ⟨proof⟩

**lemma** *insort\_aux\_gt*:  $\forall x \in \text{set } nxs. n \leq fst \ x \implies \forall x \in \text{set } mys. m \leq fst \ x \implies \neg n \leq m \implies$   
 $\text{insort } n \ (\text{sort } (\text{map } fst \ nxs \ @ \ m \ \# \ \text{map } fst \ \text{mys})) =$   
 $m \ \# \ \text{insort } n \ (\text{sort } (\text{map } fst \ nxs \ @ \ \text{map } fst \ \text{mys}))$   
 ⟨proof⟩

**lemma** *map\_fst\_merge*:  $\text{sorted\_distinct } (\text{map } fst \ nxs) \implies \text{sorted\_distinct } (\text{map } fst \ \text{mys}) \implies$   
 $\text{map } fst \ (\text{merge } nxs \ \text{mys}) = \text{sort } (\text{map } fst \ nxs \ @ \ \text{map } fst \ \text{mys})$   
 ⟨proof⟩

**lemma** *merge\_map'*:  $\text{sorted\_distinct } (\text{map } fst \ nxs) \implies \text{sorted\_distinct } (\text{map } fst \ \text{mys}) \implies$   
 $\text{fst } ' \ \text{set } nxs \cap \text{fst } ' \ \text{set } mys = \{\}$   $\implies$   
 $\text{map } snd \ nxs = \text{map } \sigma \ (\text{map } fst \ nxs) \implies \text{map } snd \ \text{mys} = \text{map } \sigma \ (\text{map } fst \ \text{mys}) \implies$   
 $\text{map } snd \ (\text{merge } nxs \ \text{mys}) = \text{map } \sigma \ (\text{sort } (\text{map } fst \ nxs \ @ \ \text{map } fst \ \text{mys}))$   
 ⟨proof⟩

**lemma** *merge\_map*:  $\text{sorted\_distinct } ns \implies \text{sorted\_distinct } ms \implies \text{set } ns \cap \text{set } ms = \{\} \implies$   
 $\text{map } snd \ (\text{merge } (\text{zip } ns \ (\text{map } \sigma \ ns)) \ (\text{zip } ms \ (\text{map } \sigma \ ms))) = \text{map } \sigma \ (\text{sort } (ns \ @ \ ms))$   
 ⟨proof⟩

**fun** *fo\_nmlz\_rec* ::  $\text{nat} \Rightarrow ('a + \text{nat} \rightarrow \text{nat}) \Rightarrow 'a \ \text{set} \Rightarrow$   
 $('a + \text{nat}) \ \text{list} \Rightarrow ('a + \text{nat}) \ \text{list}$  **where**  
 $\text{fo\_nmlz\_rec } i \ m \ AD \ [] = []$   
 $| \text{fo\_nmlz\_rec } i \ m \ AD \ (\text{Inl } x \ \# \ xs) = (\text{if } x \in AD \ \text{then } \text{Inl } x \ \# \ \text{fo\_nmlz\_rec } i \ m \ AD \ xs \ \text{else}$   
 $\text{case } m \ (\text{Inl } x) \ \text{of } \text{None} \Rightarrow \text{Inr } i \ \# \ \text{fo\_nmlz\_rec } (\text{Suc } i) \ (m(\text{Inl } x \mapsto i)) \ AD \ xs$   
 $| \text{Some } j \Rightarrow \text{Inr } j \ \# \ \text{fo\_nmlz\_rec } i \ m \ AD \ xs)$   
 $| \text{fo\_nmlz\_rec } i \ m \ AD \ (\text{Inr } n \ \# \ xs) = (\text{case } m \ (\text{Inr } n) \ \text{of } \text{None} \Rightarrow$   
 $\text{Inr } i \ \# \ \text{fo\_nmlz\_rec } (\text{Suc } i) \ (m(\text{Inr } n \mapsto i)) \ AD \ xs$   
 $| \text{Some } j \Rightarrow \text{Inr } j \ \# \ \text{fo\_nmlz\_rec } i \ m \ AD \ xs)$

**lemma** *fo\_nmlz\_rec\_sound*:  $\text{ran } m \subseteq \{..<i\} \implies \text{filter } ((\leq) \ i) \ (\text{rremdups}$   
 $(\text{List.map\_filter } (\text{case\_sum } \text{Map.empty } \text{Some}) \ (\text{fo\_nmlz\_rec } i \ m \ AD \ xs))) = ns \implies$   
 $ns = [i..<i + \text{length } ns]$   
 ⟨proof⟩

**definition** *id\_map* ::  $\text{nat} \Rightarrow ('a + \text{nat} \rightarrow \text{nat})$  **where**  
 $\text{id\_map } n = (\lambda x. \text{case } x \ \text{of } \text{Inl } x \Rightarrow \text{None} \ | \ \text{Inr } x \Rightarrow \text{if } x < n \ \text{then } \text{Some } x \ \text{else } \text{None})$

**lemma** *fo\_nmlz\_rec\_idem*:  $\text{Inl } - ' \ \text{set } ys \subseteq AD \implies$   
 $\text{rremdups } (\text{List.map\_filter } (\text{case\_sum } \text{Map.empty } \text{Some}) \ ys) = ns \implies$   
 $\text{set } (\text{filter } (\lambda n. n < i) \ ns) \subseteq \{..<i\} \implies \text{filter } ((\leq) \ i) \ ns = [i..<i + k] \implies$   
 $\text{fo\_nmlz\_rec } i \ (\text{id\_map } i) \ AD \ ys = ys$   
 ⟨proof⟩

**lemma** *fo\_nmlz\_rec\_length*:  $\text{length } (\text{fo\_nmlz\_rec } i \ m \ AD \ xs) = \text{length } xs$   
 ⟨proof⟩

**lemma** *insert\_Inr*:  $\bigwedge X. \text{insert } (\text{Inr } i) \ (X \cup \text{Inr } ' \ \{..<i\}) = X \cup \text{Inr } ' \ \{..<\text{Suc } i\}$   
 ⟨proof⟩

**lemma** *fo\_nmlz\_rec\_set*:  $\text{ran } m \subseteq \{..<i\} \implies \text{set } (\text{fo\_nmlz\_rec } i \ m \ AD \ xs) \cup \text{Inr } ' \ \{..<i\} =$   
 $\text{set } xs \cap \text{Inl } ' \ AD \cup \text{Inr } ' \ \{..<i + \text{card } (\text{set } xs - \text{Inl } ' \ AD - \text{dom } m)\}$

*<proof>*

**lemma** *fo\_nmlz\_rec\_set\_rev*:  $set (fo\_nmlz\_rec\ i\ m\ AD\ xs) \subseteq Inl\ 'AD \implies set\ xs \subseteq Inl\ 'AD$   
*<proof>*

**lemma** *fo\_nmlz\_rec\_map*:  $inj\_on\ m\ (dom\ m) \implies ran\ m \subseteq \{..<i\} \implies \exists m'. inj\_on\ m'\ (dom\ m') \wedge$   
 $(\forall n. m\ n \neq None \longrightarrow m'\ n = m\ n) \wedge (\forall (x, y) \in set\ (zip\ xs\ (fo\_nmlz\_rec\ i\ m\ AD\ xs)).$   
 $(case\ x\ of\ Inl\ x' \Rightarrow if\ x' \in AD\ then\ x = y\ else\ \exists j. m'\ (Inl\ x') = Some\ j \wedge y = Inr\ j$   
 $| Inr\ n \Rightarrow \exists j. m'\ (Inr\ n) = Some\ j \wedge y = Inr\ j))$   
*<proof>*

**lemma** *ad\_agr\_map*:

**assumes**  $length\ xs = length\ ys\ inj\_on\ m\ (dom\ m)$   
 $\bigwedge x\ y. (x, y) \in set\ (zip\ xs\ ys) \implies (case\ x\ of\ Inl\ x' \Rightarrow$   
 $if\ x' \in AD\ then\ x = y\ else\ m\ x = Some\ y \wedge (case\ y\ of\ Inl\ z \Rightarrow z \notin AD\ | Inr\ _ \Rightarrow True)$   
 $| Inr\ n \Rightarrow m\ x = Some\ y \wedge (case\ y\ of\ Inl\ z \Rightarrow z \notin AD\ | Inr\ _ \Rightarrow True))$   
**shows**  $ad\_agr\_list\ AD\ xs\ ys$   
*<proof>*

**lemma** *fo\_nmlz\_rec\_take*:  $take\ n\ (fo\_nmlz\_rec\ i\ m\ AD\ xs) = fo\_nmlz\_rec\ i\ m\ AD\ (take\ n\ xs)$   
*<proof>*

**definition** *fo\_nmlz* ::  $'a\ set \Rightarrow ('a + nat)\ list \Rightarrow ('a + nat)\ list$  **where**  
 $fo\_nmlz = fo\_nmlz\_rec\ 0\ Map.empty$

**lemma** *fo\_nmlz\_Nil[simp]*:  $fo\_nmlz\ AD\ [] = []$   
*<proof>*

**lemma** *fo\_nmlz\_Cons*:  $fo\_nmlz\ AD\ [x] =$   
 $(case\ x\ of\ Inl\ x \Rightarrow if\ x \in AD\ then\ [Inl\ x]\ else\ [Inr\ 0] | _ \Rightarrow [Inr\ 0])$   
*<proof>*

**lemma** *fo\_nmlz\_Cons\_Cons*:  $fo\_nmlz\ AD\ [x, x] =$   
 $(case\ x\ of\ Inl\ x \Rightarrow if\ x \in AD\ then\ [Inl\ x, Inl\ x]\ else\ [Inr\ 0, Inr\ 0] | _ \Rightarrow [Inr\ 0, Inr\ 0])$   
*<proof>*

**lemma** *fo\_nmlz\_sound*:  $fo\_nmlzd\ AD\ (fo\_nmlz\ AD\ xs)$   
*<proof>*

**lemma** *fo\_nmlz\_length*:  $length\ (fo\_nmlz\ AD\ xs) = length\ xs$   
*<proof>*

**lemma** *fo\_nmlz\_map*:  $\exists \tau. fo\_nmlz\ AD\ (map\ \sigma\ ns) = map\ \tau\ ns$   
*<proof>*

**lemma** *card\_set\_minus*:  $card\ (set\ xs - X) \leq length\ xs$   
*<proof>*

**lemma** *fo\_nmlz\_set*:  $set\ (fo\_nmlz\ AD\ xs) =$   
 $set\ xs \cap Inl\ 'AD \cup Inr\ '\{..<\min\ (length\ xs)\ (card\ (set\ xs - Inl\ 'AD))\}$   
*<proof>*

**lemma** *fo\_nmlz\_set\_rev*:  $set\ (fo\_nmlz\ AD\ xs) \subseteq Inl\ 'AD \implies set\ xs \subseteq Inl\ 'AD$   
*<proof>*

**lemma** *inj\_on\_empty*:  $inj\_on\ Map.empty\ (dom\ Map.empty)$  **and** *ran\_empty\_upto*:  $ran\ Map.empty \subseteq$   
 $\{..<0\}$   
*<proof>*

**lemma** *fo\_nmlz\_ad\_agr*: *ad\_agr\_list AD xs (fo\_nmlz AD xs)*  
 ⟨*proof*⟩

**lemma** *fo\_nmlzd\_mono*: *Inl - ' set xs ⊆ AD ⇒ fo\_nmlzd AD' xs ⇒ fo\_nmlzd AD xs*  
 ⟨*proof*⟩

**lemma** *fo\_nmlz\_idem*: *fo\_nmlzd AD ys ⇒ fo\_nmlz AD ys = ys*  
 ⟨*proof*⟩

**lemma** *fo\_nmlz\_take*: *take n (fo\_nmlz AD xs) = fo\_nmlz AD (take n xs)*  
 ⟨*proof*⟩

**fun** *nall\_tuples\_rec* :: *'a set ⇒ nat ⇒ nat ⇒ ('a + nat) table where*  
*nall\_tuples\_rec AD i 0 = {[]}*  
 | *nall\_tuples\_rec AD i (Suc n) = ⋃((λas. (λx. x # as) ' (Inl ' AD ∪ Inr ' {..*i*))) ' nall\_tuples\_rec AD i n) ∪ (λas. Inr i # as) ' nall\_tuples\_rec AD (Suc i) n*

**lemma** *nall\_tuples\_rec\_Inl*: *vs ∈ nall\_tuples\_rec AD i n ⇒ Inl - ' set vs ⊆ AD*  
 ⟨*proof*⟩

**lemma** *nall\_tuples\_rec\_length*: *xs ∈ nall\_tuples\_rec AD i n ⇒ length xs = n*  
 ⟨*proof*⟩

**lemma** *fun\_upd\_id\_map*: *(id\_map i)(Inr i ↦ i) = id\_map (Suc i)*  
 ⟨*proof*⟩

**lemma** *id\_mapD*: *id\_map j (Inr i) = None ⇒ j ≤ i id\_map j (Inr i) = Some x ⇒ i < j ∧ i = x*  
 ⟨*proof*⟩

**lemma** *nall\_tuples\_rec\_fo\_nmlz\_rec\_sound*: *i ≤ j ⇒ xs ∈ nall\_tuples\_rec AD i n ⇒ fo\_nmlz\_rec j (id\_map j) AD xs = xs*  
 ⟨*proof*⟩

**lemma** *nall\_tuples\_rec\_fo\_nmlz\_rec\_complete*:  
**assumes** *fo\_nmlz\_rec j (id\_map j) AD xs = xs*  
**shows** *xs ∈ nall\_tuples\_rec AD j (length xs)*  
 ⟨*proof*⟩

**lemma** *nall\_tuples\_rec\_fo\_nmlz*: *xs ∈ nall\_tuples\_rec AD 0 (length xs) ⇔ fo\_nmlz AD xs = xs*  
 ⟨*proof*⟩

**lemma** *fo\_nmlzd\_code[code]*: *fo\_nmlzd AD xs ⇔ fo\_nmlz AD xs = xs*  
 ⟨*proof*⟩

**lemma** *nall\_tuples\_code[code]*: *nall\_tuples AD n = nall\_tuples\_rec AD 0 n*  
 ⟨*proof*⟩

**lemma** *exists\_map*: *length xs = length ys ⇒ distinct xs ⇒ ∃ f. ys = map f xs*  
 ⟨*proof*⟩

**lemma** *exists\_fo\_nmlzd*:  
**assumes** *length xs = length ys distinct xs fo\_nmlzd AD ys*  
**shows** *∃ f. ys = fo\_nmlz AD (map f xs)*  
 ⟨*proof*⟩

**lemma** *list\_induct2\_rev[consumes 1]*: *length xs = length ys ⇒ (P [] []) ⇒ (∧ x y xs ys. P xs ys ⇒ P (xs @ [x]) (ys @ [y])) ⇒ P xs ys*

*<proof>*

**lemma** *ad\_agr\_list\_fo\_nmlzd*:

**assumes** *ad\_agr\_list AD vs vs' fo\_nmlzd AD vs fo\_nmlzd AD vs'*

**shows** *vs = vs'*

*<proof>*

**lemma** *fo\_nmlz\_eqI*:

**assumes** *ad\_agr\_list AD vs vs'*

**shows** *fo\_nmlz AD vs = fo\_nmlz AD vs'*

*<proof>*

**lemma** *fo\_nmlz\_eqD*:

**assumes** *fo\_nmlz AD vs = fo\_nmlz AD vs'*

**shows** *ad\_agr\_list AD vs vs'*

*<proof>*

**lemma** *fo\_nmlz\_eq*: *fo\_nmlz AD vs = fo\_nmlz AD vs'  $\longleftrightarrow$  ad\_agr\_list AD vs vs'*

*<proof>*

**lemma** *fo\_nmlz\_mono*:

**assumes** *AD  $\subseteq$  AD' Inl -' set xs  $\subseteq$  AD*

**shows** *fo\_nmlz AD' xs = fo\_nmlz AD xs*

*<proof>*

**definition** *proj\_vals* :: *'c val set  $\Rightarrow$  nat list  $\Rightarrow$  'c table* **where**

*proj\_vals R ns = ( $\lambda\tau$ . map  $\tau$  ns) ' R*

**definition** *proj\_fmula* :: *('a, 'b) fo\_fmula  $\Rightarrow$  'c val set  $\Rightarrow$  'c table* **where**

*proj\_fmula  $\varphi$  R = proj\_vals R (fv\_fo\_fmula\_list  $\varphi$ )*

**lemmas** *proj\_fmula\_map = proj\_fmula\_def[unfolded proj\_vals\_def]*

**definition** *extends\_subst*  $\sigma \tau = (\forall x. \sigma x \neq \text{None} \longrightarrow \sigma x = \tau x)$

**definition** *ext\_tuple* :: *'a set  $\Rightarrow$  nat list  $\Rightarrow$  nat list  $\Rightarrow$*

*('a + nat) list  $\Rightarrow$  ('a + nat) list set* **where**

*ext\_tuple AD fv\_sub fv\_sub\_comp as = (if fv\_sub\_comp = [] then {as}*

*else ( $\lambda$ fs. map snd (merge (zip fv\_sub as) (zip fv\_sub\_comp fs))) ')*

*(nall\_tuples\_rec AD (card (Inr -' set as)) (length fv\_sub\_comp)))*

**lemma** *ext\_tuple\_eq*: *length fv\_sub = length as  $\Longrightarrow$*

*ext\_tuple AD fv\_sub fv\_sub\_comp as =*

*( $\lambda$ fs. map snd (merge (zip fv\_sub as) (zip fv\_sub\_comp fs))) ')*

*(nall\_tuples\_rec AD (card (Inr -' set as)) (length fv\_sub\_comp))*

*<proof>*

**lemma** *map\_map\_of*: *length xs = length ys  $\Longrightarrow$  distinct xs  $\Longrightarrow$*

*ys = map (the  $\circ$  (map\_of (zip xs ys))) xs*

*<proof>*

**lemma** *id\_map\_empty*: *id\_map 0 = Map.empty*

*<proof>*

**lemma** *fo\_nmlz\_rec\_shift*:

**fixes** *xs* :: *('a + nat) list*

**shows** *fo\_nmlz\_rec i (id\_map i) AD xs = xs  $\Longrightarrow$*

*i' = card (Inr -' (Inr -' {..*i*}  $\cup$  set (take *n* xs)))  $\Longrightarrow$  *n  $\leq$  length xs  $\Longrightarrow$**

$fo\_nmlz\_rec\ i'\ (id\_map\ i')\ AD\ (drop\ n\ xs) = drop\ n\ xs$   
 ⟨proof⟩

**fun** *proj\_tuple* :: *nat list*  $\Rightarrow$  (*nat*  $\times$  (*'a* + *nat*)) *list*  $\Rightarrow$  (*'a* + *nat*) *list* **where**  
*proj\_tuple* [] *mys* = []  
 | *proj\_tuple* *ns* [] = []  
 | *proj\_tuple* (*n* # *ns*) ((*m*, *y*) # *mys*) =  
   (*if* *m* < *n* *then* *proj\_tuple* (*n* # *ns*) *mys* *else*  
   *if* *m* = *n* *then* *y* # *proj\_tuple* *ns* *mys*  
   else *proj\_tuple* *ns* ((*m*, *y*) # *mys*))

**lemma** *proj\_tuple\_idle*: *proj\_tuple* (*map* *fst* *nxs*) *nxs* = *map* *snd* *nxs*  
 ⟨proof⟩

**lemma** *proj\_tuple\_merge*: *sorted\_distinct* (*map* *fst* *nxs*)  $\Longrightarrow$  *sorted\_distinct* (*map* *fst* *mys*)  $\Longrightarrow$   
*set* (*map* *fst* *nxs*)  $\cap$  *set* (*map* *fst* *mys*) = {}  $\Longrightarrow$   
*proj\_tuple* (*map* *fst* *nxs*) (*merge* *nxs* *mys*) = *map* *snd* *nxs*  
 ⟨proof⟩

**lemma** *proj\_tuple\_map*:  
**assumes** *sorted\_distinct* *ns* *sorted\_distinct* *ms* *set* *ns*  $\subseteq$  *set* *ms*  
**shows** *proj\_tuple* *ns* (*zip* *ms* (*map*  $\sigma$  *ms*)) = *map*  $\sigma$  *ns*  
 ⟨proof⟩

**lemma** *proj\_tuple\_length*:  
**assumes** *sorted\_distinct* *ns* *sorted\_distinct* *ms* *set* *ns*  $\subseteq$  *set* *ms* *length* *ms* = *length* *xs*  
**shows** *length* (*proj\_tuple* *ns* (*zip* *ms* *xs*)) = *length* *ns*  
 ⟨proof⟩

**lemma** *ext\_tuple\_sound*:  
**assumes** *sorted\_distinct* *fv\_sub* *sorted\_distinct* *fv\_sub\_comp* *sorted\_distinct* *fv\_all*  
*set* *fv\_sub*  $\cap$  *set* *fv\_sub\_comp* = {} *set* *fv\_sub*  $\cup$  *set* *fv\_sub\_comp* = *set* *fv\_all*  
*ass* = *fo\_nmlz* *AD* ' *proj\_vals* *R* *fv\_sub*  
 $\bigwedge \sigma \tau. ad\_agr\_sets\ (set\ fv\_sub)\ (set\ fv\_sub)\ AD\ \sigma\ \tau \Longrightarrow \sigma \in R \longleftrightarrow \tau \in R$   
*xs*  $\in$  *fo\_nmlz* *AD* '  $\bigcup$  (*ext\_tuple* *AD* *fv\_sub* *fv\_sub\_comp* ' *ass*)  
**shows** *fo\_nmlz* *AD* (*proj\_tuple* *fv\_sub* (*zip* *fv\_all* *xs*))  $\in$  *ass*  
*xs*  $\in$  *fo\_nmlz* *AD* ' *proj\_vals* *R* *fv\_all*  
 ⟨proof⟩

**lemma** *ext\_tuple\_complete*:  
**assumes** *sorted\_distinct* *fv\_sub* *sorted\_distinct* *fv\_sub\_comp* *sorted\_distinct* *fv\_all*  
*set* *fv\_sub*  $\cap$  *set* *fv\_sub\_comp* = {} *set* *fv\_sub*  $\cup$  *set* *fv\_sub\_comp* = *set* *fv\_all*  
*ass* = *fo\_nmlz* *AD* ' *proj\_vals* *R* *fv\_sub*  
 $\bigwedge \sigma \tau. ad\_agr\_sets\ (set\ fv\_sub)\ (set\ fv\_sub)\ AD\ \sigma\ \tau \Longrightarrow \sigma \in R \longleftrightarrow \tau \in R$   
*xs* = *fo\_nmlz* *AD* (*map*  $\sigma$  *fv\_all*)  $\sigma \in R$   
**shows** *xs*  $\in$  *fo\_nmlz* *AD* '  $\bigcup$  (*ext\_tuple* *AD* *fv\_sub* *fv\_sub\_comp* ' *ass*)  
 ⟨proof⟩

**definition** *ext\_tuple\_set* *AD* *ns* *ns'* *X* = (*if* *ns'* = [] *then* *X* *else* *fo\_nmlz* *AD* '  $\bigcup$  (*ext\_tuple* *AD* *ns* *ns'* ' *X*))

**lemma** *ext\_tuple\_set\_eq*: *Ball* *X* (*fo\_nmlzd* *AD*)  $\Longrightarrow$  *ext\_tuple\_set* *AD* *ns* *ns'* *X* = *fo\_nmlz* *AD* '  $\bigcup$  (*ext\_tuple* *AD* *ns* *ns'* ' *X*)  
 ⟨proof⟩

**lemma** *ext\_tuple\_set\_mono*: *A*  $\subseteq$  *B*  $\Longrightarrow$  *ext\_tuple\_set* *AD* *ns* *ns'* *A*  $\subseteq$  *ext\_tuple\_set* *AD* *ns* *ns'* *B*  
 ⟨proof⟩

**lemma** *ext\_tuple\_correct*:

**assumes** *sorted\_distinct fv\_sub sorted\_distinct fv\_sub\_comp sorted\_distinct fv\_all*  
*set fv\_sub ∩ set fv\_sub\_comp = {} set fv\_sub ∪ set fv\_sub\_comp = set fv\_all*  
*ass = fo\_nmlz AD ' proj\_vals R fv\_sub*  
 $\bigwedge \sigma \tau. \text{ad\_agr\_sets } (\text{set } fv\_sub) (\text{set } fv\_sub) \text{ AD } \sigma \tau \implies \sigma \in R \longleftrightarrow \tau \in R$   
**shows** *ext\_tuple\_set AD fv\_sub fv\_sub\_comp ass = fo\_nmlz AD ' proj\_vals R fv\_all*  
*<proof>*

**lemma** *proj\_tuple\_sound*:

**assumes** *sorted\_distinct fv\_sub sorted\_distinct fv\_sub\_comp sorted\_distinct fv\_all*  
*set fv\_sub ∩ set fv\_sub\_comp = {} set fv\_sub ∪ set fv\_sub\_comp = set fv\_all*  
*ass = fo\_nmlz AD ' proj\_vals R fv\_sub*  
 $\bigwedge \sigma \tau. \text{ad\_agr\_sets } (\text{set } fv\_sub) (\text{set } fv\_sub) \text{ AD } \sigma \tau \implies \sigma \in R \longleftrightarrow \tau \in R$   
*fo\_nmlz AD xs = xs length xs = length fv\_all*  
*fo\_nmlz AD (proj\_tuple fv\_sub (zip fv\_all xs)) ∈ ass*  
**shows** *xs ∈ fo\_nmlz AD ' ∪ (ext\_tuple AD fv\_sub fv\_sub\_comp ' ass)*  
*<proof>*

**lemma** *proj\_tuple\_correct*:

**assumes** *sorted\_distinct fv\_sub sorted\_distinct fv\_sub\_comp sorted\_distinct fv\_all*  
*set fv\_sub ∩ set fv\_sub\_comp = {} set fv\_sub ∪ set fv\_sub\_comp = set fv\_all*  
*ass = fo\_nmlz AD ' proj\_vals R fv\_sub*  
 $\bigwedge \sigma \tau. \text{ad\_agr\_sets } (\text{set } fv\_sub) (\text{set } fv\_sub) \text{ AD } \sigma \tau \implies \sigma \in R \longleftrightarrow \tau \in R$   
*fo\_nmlz AD xs = xs length xs = length fv\_all*  
**shows** *xs ∈ fo\_nmlz AD ' ∪ (ext\_tuple AD fv\_sub fv\_sub\_comp ' ass) ⟷*  
*fo\_nmlz AD (proj\_tuple fv\_sub (zip fv\_all xs)) ∈ ass*  
*<proof>*

**fun** *unify\_vals\_terms* :: ('a + 'c) list ⇒ ('a fo\_term) list ⇒ (nat → ('a + 'c)) ⇒  
 (nat → ('a + 'c)) option **where**  
*unify\_vals\_terms [] [] σ = Some σ*  
 | *unify\_vals\_terms (v # vs) ((Const c') # ts) σ =*  
   *(if v = Inl c' then unify\_vals\_terms vs ts σ else None)*  
 | *unify\_vals\_terms (v # vs) ((Var n) # ts) σ =*  
   *(case σ n of Some x ⇒ (if v = x then unify\_vals\_terms vs ts σ else None)*  
   | *None ⇒ unify\_vals\_terms vs ts (σ(n := Some v)))*  
 | *unify\_vals\_terms \_ \_ \_ = None*

**lemma** *unify\_vals\_terms\_extends*: *unify\_vals\_terms vs ts σ = Some σ' ⟹ extends\_subst σ σ'*  
*<proof>*

**lemma** *unify\_vals\_terms\_sound*: *unify\_vals\_terms vs ts σ = Some σ' ⟹ (the ∘ σ') ∘ e ts = vs*  
*<proof>*

**lemma** *unify\_vals\_terms\_complete*:  $\sigma'' \circ e \text{ ts} = \text{vs} \implies (\bigwedge n. \sigma n \neq \text{None} \implies \sigma n = \text{Some } (\sigma'' n)) \implies$   
 $\exists \sigma'. \text{unify\_vals\_terms vs ts } \sigma = \text{Some } \sigma'$   
*<proof>*

**definition** *eval\_table* :: 'a fo\_term list ⇒ ('a + 'c) table ⇒ ('a + 'c) table **where**

*eval\_table ts X = (let fvs = fv\_fo\_terms\_list ts in*  
 $\bigcup ((\lambda vs. \text{case } \text{unify\_vals\_terms vs ts } \text{Map.empty of Some } \sigma \Rightarrow$   
 $\{\text{map } (\text{the } \circ \sigma) \text{ fvs}\} \mid \_ \Rightarrow \{\}) ' X))$

**lemma** *eval\_table*:

**fixes** *X* :: ('a + 'c) table  
**shows** *eval\_table ts X = proj\_vals {σ. σ ∘ e ts ∈ X} (fv\_fo\_terms\_list ts)*  
*<proof>*

**fun** *ad\_agr\_close\_rec* :: *nat*  $\Rightarrow$  (*nat*  $\rightarrow$  'a + *nat*)  $\Rightarrow$  'a *set*  $\Rightarrow$   
('a + *nat*) *list*  $\Rightarrow$  ('a + *nat*) *list set* **where**  
*ad\_agr\_close\_rec* *i m AD* [] = {}  
| *ad\_agr\_close\_rec* *i m AD* (*Inl* *x # xs*) = ( $\lambda$ *xs. Inl* *x # xs*) ' *ad\_agr\_close\_rec* *i m AD* *xs*  
| *ad\_agr\_close\_rec* *i m AD* (*Inr* *n # xs*) = (*case* *m n* of *None*  $\Rightarrow$   $\bigcup$  ( $\lambda$ *x. Inl* *x # xs*) ' *ad\_agr\_close\_rec* *i* (*m*(*n* := *Some* (*Inl* *x*))) (*AD* - {*x*} *xs*) ' *AD*)  $\cup$   
( $\lambda$ *xs. Inr* *i # xs*) ' *ad\_agr\_close\_rec* (*Suc* *i*) (*m*(*n* := *Some* (*Inr* *i*))) *AD* *xs*  
| *Some* *v*  $\Rightarrow$  ( $\lambda$ *xs. v # xs*) ' *ad\_agr\_close\_rec* *i m AD* *xs*)

**lemma** *ad\_agr\_close\_rec\_length*: *ys*  $\in$  *ad\_agr\_close\_rec* *i m AD* *xs*  $\Longrightarrow$  *length* *xs* = *length* *ys*  
<proof>

**lemma** *ad\_agr\_close\_rec\_sound*: *ys*  $\in$  *ad\_agr\_close\_rec* *i m AD* *xs*  $\Longrightarrow$   
*fo\_nmlz\_rec* *j* (*id\_map* *j*) *X* *xs* = *xs*  $\Longrightarrow$  *X*  $\cap$  *AD* = {}  $\Longrightarrow$  *X*  $\cap$  *Y* = {}  $\Longrightarrow$  *Y*  $\cap$  *AD* = {}  $\Longrightarrow$   
*inj\_on* *m* (*dom* *m*)  $\Longrightarrow$  *dom* *m* = {..*j*}  $\Longrightarrow$  *ran* *m*  $\subseteq$  *Inl* ' *Y*  $\cup$  *Inr* ' {..*i*}  $\Longrightarrow$  *i*  $\leq$  *j*  $\Longrightarrow$   
*fo\_nmlz\_rec* *i* (*id\_map* *i*) (*X*  $\cup$  *Y*  $\cup$  *AD*) *ys* = *ys*  $\wedge$   
( $\exists$  *m'*. *inj\_on* *m'* (*dom* *m'*)  $\wedge$  ( $\forall$  *n v. m* *n* = *Some* *v*  $\rightarrow$  *m'* (*Inr* *n*) = *Some* *v*)  $\wedge$   
( $\forall$  (*x, y*)  $\in$  *set* (*zip* *xs* *ys*). *case* *x* of *Inl* *x'*  $\Rightarrow$   
if *x'*  $\in$  *X* then *x* = *y* else *m'* *x* = *Some* *y*  $\wedge$  (*case* *y* of *Inl* *z*  $\Rightarrow$  *z*  $\notin$  *X* | *Inr* *x*  $\Rightarrow$  *True*)  
| *Inr* *n*  $\Rightarrow$  *m'* *x* = *Some* *y*  $\wedge$  (*case* *y* of *Inl* *z*  $\Rightarrow$  *z*  $\notin$  *X* | *Inr* *x*  $\Rightarrow$  *True*)))  
<proof>

**lemma** *ad\_agr\_close\_rec\_complete*:

**fixes** *xs* :: ('a + *nat*) *list*  
**shows** *fo\_nmlz\_rec* *j* (*id\_map* *j*) *X* *xs* = *xs*  $\Longrightarrow$   
*X*  $\cap$  *AD* = {}  $\Longrightarrow$  *X*  $\cap$  *Y* = {}  $\Longrightarrow$  *Y*  $\cap$  *AD* = {}  $\Longrightarrow$   
*inj\_on* *m* (*dom* *m*)  $\Longrightarrow$  *dom* *m* = {..*j*}  $\Longrightarrow$  *ran* *m* = *Inl* ' *Y*  $\cup$  *Inr* ' {..*i*}  $\Longrightarrow$  *i*  $\leq$  *j*  $\Longrightarrow$   
( $\bigwedge$  *n b. Inr* *n, b*)  $\in$  *set* (*zip* *xs* *ys*)  $\Longrightarrow$  *case* *m n* of *Some* *v*  $\Rightarrow$  *v* = *b* | *None*  $\Rightarrow$  *b*  $\notin$  *ran* *m*)  $\Longrightarrow$   
*fo\_nmlz\_rec* *i* (*id\_map* *i*) (*X*  $\cup$  *Y*  $\cup$  *AD*) *ys* = *ys*  $\Longrightarrow$  *ad\_agr\_list* *X* *xs* *ys*  $\Longrightarrow$   
*ys*  $\in$  *ad\_agr\_close\_rec* *i m AD* *xs*  
<proof>

**definition** *ad\_agr\_close* :: 'a *set*  $\Rightarrow$  ('a + *nat*) *list*  $\Rightarrow$  ('a + *nat*) *list set* **where**  
*ad\_agr\_close* *AD* *xs* = *ad\_agr\_close\_rec* 0 *Map.empty* *AD* *xs*

**lemma** *ad\_agr\_close\_sound*:

**assumes** *ys*  $\in$  *ad\_agr\_close* *Y* *xs* *fo\_nmlzd* *X* *xs* *X*  $\cap$  *Y* = {}  
**shows** *fo\_nmlzd* (*X*  $\cup$  *Y*) *ys*  $\wedge$  *ad\_agr\_list* *X* *xs* *ys*  
<proof>

**lemma** *ad\_agr\_close\_complete*:

**assumes** *X*  $\cap$  *Y* = {} *fo\_nmlzd* *X* *xs* *fo\_nmlzd* (*X*  $\cup$  *Y*) *ys* *ad\_agr\_list* *X* *xs* *ys*  
**shows** *ys*  $\in$  *ad\_agr\_close* *Y* *xs*  
<proof>

**lemma** *ad\_agr\_close\_empty*: *fo\_nmlzd* *X* *xs*  $\Longrightarrow$  *ad\_agr\_close* {} *xs* = {*xs*}  
<proof>

**lemma** *ad\_agr\_close\_set\_correct*:

**assumes** *AD'*  $\subseteq$  *AD* *sorted\_distinct* *ns*  
 $\bigwedge$   $\sigma$   $\tau. ad\_agr\_sets$  (*set* *ns*) (*set* *ns*) *AD'*  $\sigma$   $\tau$   $\Longrightarrow$   $\sigma \in R \iff \tau \in R$   
**shows**  $\bigcup$  (*ad\_agr\_close* (*AD* - *AD'*) ' *fo\_nmlz* *AD'* ' *proj\_vals* *R* *ns*) = *fo\_nmlz* *AD* ' *proj\_vals* *R* *ns*  
<proof>

**lemma** *ad\_agr\_close\_correct*:

**assumes** *AD'*  $\subseteq$  *AD*  
 $\bigwedge$   $\sigma$   $\tau. ad\_agr\_sets$  (*set* (*fv\_fo\_fmla\_list*  $\varphi$ )) (*set* (*fv\_fo\_fmla\_list*  $\varphi$ )) *AD'*  $\sigma$   $\tau$   $\Longrightarrow$   
 $\sigma \in R \iff \tau \in R$

**shows**  $\bigcup (ad\_agr\_close (AD - AD') \text{ ' } fo\_nmlz AD' \text{ ' } proj\_fmla \varphi R) = fo\_nmlz AD \text{ ' } proj\_fmla \varphi R$   
 <proof>

**definition**  $ad\_agr\_close\_set AD X = (if Set.is\_empty AD then X else \bigcup (ad\_agr\_close AD \text{ ' } X))$

**lemma**  $ad\_agr\_close\_set\_eq: Ball X (fo\_nmlzd AD') \implies ad\_agr\_close\_set AD X = \bigcup (ad\_agr\_close AD \text{ ' } X)$   
 <proof>

**lemma**  $Ball\_fo\_nmlzd: Ball (fo\_nmlz AD \text{ ' } X) (fo\_nmlzd AD)$   
 <proof>

**lemmas**  $ad\_agr\_close\_set\_nmlz\_eq = ad\_agr\_close\_set\_eq[OF Ball\_fo\_nmlzd]$

**definition**  $eval\_pred :: ('a fo\_term) list \Rightarrow 'a table \Rightarrow ('a, 'c) fo\_t \text{ where}$   
 $eval\_pred ts X = (let AD = \bigcup (set (map set\_fo\_term ts)) \cup \bigcup (set \text{ ' } X) \text{ in}$   
 $(AD, length (fv\_fo\_terms\_list ts), eval\_table ts (map Inl \text{ ' } X)))$

**definition**  $eval\_bool :: bool \Rightarrow ('a, 'c) fo\_t \text{ where}$   
 $eval\_bool b = (if b then (\{\}, 0, \{\}) else (\{\}, 0, \{\}))$

**definition**  $eval\_eq :: 'a fo\_term \Rightarrow 'a fo\_term \Rightarrow ('a, nat) fo\_t \text{ where}$   
 $eval\_eq t t' = (case t \text{ of } Var n \Rightarrow$   
 $(case t' \text{ of } Var n' \Rightarrow$   
 $if n = n' then (\{\}, 1, \{[Inr 0]\})$   
 $else (\{\}, 2, \{[Inr 0, Inr 0]\})$   
 $| Const c' \Rightarrow (\{c'\}, 1, \{[Inl c']\}))$   
 $| Const c \Rightarrow$   
 $(case t' \text{ of } Var n' \Rightarrow (\{c\}, 1, \{[Inl c]\})$   
 $| Const c' \Rightarrow if c = c' then (\{c\}, 0, \{\}) else (\{c, c'\}, 0, \{\})))$

**fun**  $eval\_neg :: nat list \Rightarrow ('a, nat) fo\_t \Rightarrow ('a, nat) fo\_t \text{ where}$   
 $eval\_neg ns (AD, \_, X) = (AD, length ns, nall\_tuples AD (length ns) - X)$

**definition**  $eval\_conj\_tuple AD ns\varphi ns\psi xs ys =$   
 $(let cxs = filter (\lambda(n, x). n \notin set ns\psi \wedge isl x) (zip ns\varphi xs);$   
 $nxs = map fst (filter (\lambda(n, x). n \notin set ns\psi \wedge \neg isl x) (zip ns\varphi xs));$   
 $cys = filter (\lambda(n, y). n \notin set ns\varphi \wedge isl y) (zip ns\psi ys);$   
 $nys = map fst (filter (\lambda(n, y). n \notin set ns\varphi \wedge \neg isl y) (zip ns\psi ys)) \text{ in}$   
 $fo\_nmlz AD \text{ ' } ext\_tuple \{\} (sort (ns\varphi @ map fst cys)) nys (map snd (merge (zip ns\varphi xs) cys))) \cap$   
 $fo\_nmlz AD \text{ ' } ext\_tuple \{\} (sort (ns\psi @ map fst cxs)) nxs (map snd (merge (zip ns\psi ys) cxs)))$

**definition**  $eval\_conj\_set AD ns\varphi X\varphi ns\psi X\psi = \bigcup ((\lambda xs. \bigcup (eval\_conj\_tuple AD ns\varphi ns\psi xs \text{ ' } X\psi)) \text{ ' } X\varphi)$

**definition**  $idx\_join AD ns ns\varphi X\varphi ns\psi X\psi =$   
 $(let idx\varphi' = cluster (Some \circ (\lambda xs. fo\_nmlz AD (proj\_tuple ns (zip ns\varphi xs)))) X\varphi;$   
 $idx\psi' = cluster (Some \circ (\lambda ys. fo\_nmlz AD (proj\_tuple ns (zip ns\psi ys)))) X\psi \text{ in}$   
 $set\_of\_idx (mapping\_join (\lambda X\varphi'' X\psi''. eval\_conj\_set AD ns\varphi X\varphi'' ns\psi X\psi'') idx\varphi' idx\psi'))$

**fun**  $eval\_conj :: nat list \Rightarrow ('a, nat) fo\_t \Rightarrow nat list \Rightarrow ('a, nat) fo\_t \Rightarrow$   
 $('a, nat) fo\_t \text{ where}$   
 $eval\_conj ns\varphi (AD\varphi, \_, X\varphi) ns\psi (AD\psi, \_, X\psi) = (let AD = AD\varphi \cup AD\psi; AD\Delta\varphi = AD - AD\varphi;$   
 $AD\Delta\psi = AD - AD\psi; ns = filter (\lambda n. n \in set ns\psi) ns\varphi \text{ in}$   
 $(AD, card (set ns\varphi \cup set ns\psi), idx\_join AD ns ns\varphi (ad\_agr\_close\_set AD\Delta\varphi X\varphi) ns\psi (ad\_agr\_close\_set AD\Delta\psi X\psi)))$

**fun**  $eval\_ajoin :: nat list \Rightarrow ('a, nat) fo\_t \Rightarrow nat list \Rightarrow ('a, nat) fo\_t \Rightarrow$

('a, nat) fo\_t **where**  
 eval\_ajoin nsφ (ADφ, \_, Xφ) nsψ (ADψ, \_, Xψ) = (let AD = ADφ ∪ ADψ; ADΔφ = AD - ADφ;  
 ADΔψ = AD - ADψ;  
 ns = filter (λn. n ∈ set nsψ) nsφ; nsφ' = filter (λn. n ∉ set nsφ) nsψ;  
 idxφ = cluster (Some ∘ (λxs. fo\_nmlz ADψ (proj\_tuple ns (zip nsφ xs)))) (ad\_agr\_close\_set ADΔφ  
 Xφ);  
 idxψ = cluster (Some ∘ (λys. fo\_nmlz ADψ (proj\_tuple ns (zip nsψ ys)))) Xψ in  
 (AD, card (set nsφ ∪ set nsψ), set\_of\_idx (Mapping.map\_values (λxs X. case Mapping.lookup idxψ  
 xs of Some Y ⇒  
 idx\_join AD ns nsφ X nsψ (ad\_agr\_close\_set ADΔψ (ext\_tuple\_set ADψ ns nsφ' {xs} - Y)) | \_  
 ⇒ ext\_tuple\_set AD nsφ nsφ' X) idxφ)))

**fun** eval\_disj :: nat list ⇒ ('a, nat) fo\_t ⇒ nat list ⇒ ('a, nat) fo\_t ⇒  
 ('a, nat) fo\_t **where**  
 eval\_disj nsφ (ADφ, \_, Xφ) nsψ (ADψ, \_, Xψ) = (let AD = ADφ ∪ ADψ;  
 nsφ' = filter (λn. n ∉ set nsφ) nsψ;  
 nsψ' = filter (λn. n ∉ set nsψ) nsφ;  
 ADΔφ = AD - ADφ; ADΔψ = AD - ADψ in  
 (AD, card (set nsφ ∪ set nsψ),  
 ext\_tuple\_set AD nsφ nsφ' (ad\_agr\_close\_set ADΔφ Xφ) ∪  
 ext\_tuple\_set AD nsψ nsψ' (ad\_agr\_close\_set ADΔψ Xψ)))

**fun** eval\_exists :: nat ⇒ nat list ⇒ ('a, nat) fo\_t ⇒ ('a, nat) fo\_t **where**  
 eval\_exists i ns (AD, \_, X) = (case pos i ns of Some j ⇒  
 (AD, length ns - 1, fo\_nmlz AD 'rem\_nth j ' X)  
 | None ⇒ (AD, length ns, X))

**fun** eval\_forall :: nat ⇒ nat list ⇒ ('a, nat) fo\_t ⇒ ('a, nat) fo\_t **where**  
 eval\_forall i ns (AD, \_, X) = (case pos i ns of Some j ⇒  
 let n = card AD in  
 (AD, length ns - 1, Mapping.keys (Mapping.filter (λt Z. n + card (Inr - ' set t) + 1 ≤ card Z)  
 (cluster (Some ∘ (λts. fo\_nmlz AD (rem\_nth j ts))) X)))  
 | None ⇒ (AD, length ns, X))

**lemma** combine\_map2: **assumes** length ys = length xs length ys' = length xs'  
 distinct xs distinct xs' set xs ∩ set xs' = {}  
**shows** ∃ f. ys = map f xs ∧ ys' = map f xs'  
 <proof>

**lemma** combine\_map3: **assumes** length ys = length xs length ys' = length xs' length ys'' = length xs''  
 distinct xs distinct xs' distinct xs'' set xs ∩ set xs' = {} set xs ∩ set xs'' = {} set xs' ∩ set xs'' = {}  
**shows** ∃ f. ys = map f xs ∧ ys' = map f xs' ∧ ys'' = map f xs''  
 <proof>

**lemma** distinct\_set\_zip: length nsx = length xs ⇒ distinct nsx ⇒  
 (a, b) ∈ set (zip nsx xs) ⇒ (a, ba) ∈ set (zip nsx xs) ⇒ b = ba  
 <proof>

**lemma** fo\_nmlz\_idem\_isl:  
**assumes** ∧ x. x ∈ set xs ⇒ (case x of Inl z ⇒ z ∈ X | \_ ⇒ False)  
**shows** fo\_nmlz X xs = xs  
 <proof>

**lemma** set\_zip\_mapI: x ∈ set xs ⇒ (f x, g x) ∈ set (zip (map f xs) (map g xs))  
 <proof>

**lemma** ad\_agr\_list\_fo\_nmlzd\_isl:  
**assumes** ad\_agr\_list X (map f xs) (map g xs) fo\_nmlzd X (map f xs) x ∈ set xs isl (f x)

**shows**  $f x = g x$   
(proof)

**lemma** *eval\_conj\_tuple\_close\_empty2*:

**assumes**  $fo\_nmlzd\ X\ xs\ fo\_nmlzd\ Y\ ys$   
 $length\ nsx = length\ xs\ length\ nsy = length\ ys$   
 $sorted\_distinct\ nsx\ sorted\_distinct\ nsy$   
 $sorted\_distinct\ ns\ set\ ns \subseteq set\ nsx \cap set\ nsy$   
 $fo\_nmlz\ (X \cap Y)\ (proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq fo\_nmlz\ (X \cap Y)\ (proj\_tuple\ ns\ (zip\ nsy\ ys)) \vee$   
 $(proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq proj\_tuple\ ns\ (zip\ nsy\ ys) \wedge$   
 $(\forall x \in set\ (proj\_tuple\ ns\ (zip\ nsx\ xs)).\ isl\ x) \wedge (\forall y \in set\ (proj\_tuple\ ns\ (zip\ nsy\ ys)).\ isl\ y)$   
 $xs' \in ad\_agr\_close\ ((X \cup Y) - X)\ xs\ ys' \in ad\_agr\_close\ ((X \cup Y) - Y)\ ys$   
**shows**  $eval\_conj\_tuple\ (X \cup Y)\ nsx\ nsy\ xs'\ ys' = \{\}$

(proof)

**lemma** *eval\_conj\_tuple\_close\_empty*:

**assumes**  $fo\_nmlzd\ X\ xs\ fo\_nmlzd\ Y\ ys$   
 $length\ nsx = length\ xs\ length\ nsy = length\ ys$   
 $sorted\_distinct\ nsx\ sorted\_distinct\ nsy$   
 $ns = filter\ (\lambda n.\ n \in set\ nsy)\ nsx$   
 $fo\_nmlz\ (X \cap Y)\ (proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq fo\_nmlz\ (X \cap Y)\ (proj\_tuple\ ns\ (zip\ nsy\ ys))$   
 $xs' \in ad\_agr\_close\ ((X \cup Y) - X)\ xs\ ys' \in ad\_agr\_close\ ((X \cup Y) - Y)\ ys$   
**shows**  $eval\_conj\_tuple\ (X \cup Y)\ nsx\ nsy\ xs'\ ys' = \{\}$

(proof)

**lemma** *eval\_conj\_tuple\_empty2*:

**assumes**  $fo\_nmlzd\ Z\ xs\ fo\_nmlzd\ Z\ ys$   
 $length\ nsx = length\ xs\ length\ nsy = length\ ys$   
 $sorted\_distinct\ nsx\ sorted\_distinct\ nsy$   
 $sorted\_distinct\ ns\ set\ ns \subseteq set\ nsx \cap set\ nsy$   
 $fo\_nmlz\ Z\ (proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq fo\_nmlz\ Z\ (proj\_tuple\ ns\ (zip\ nsy\ ys)) \vee$   
 $(proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq proj\_tuple\ ns\ (zip\ nsy\ ys) \wedge$   
 $(\forall x \in set\ (proj\_tuple\ ns\ (zip\ nsx\ xs)).\ isl\ x) \wedge (\forall y \in set\ (proj\_tuple\ ns\ (zip\ nsy\ ys)).\ isl\ y)$   
**shows**  $eval\_conj\_tuple\ Z\ nsx\ nsy\ xs\ ys = \{\}$

(proof)

**lemma** *eval\_conj\_tuple\_empty*:

**assumes**  $fo\_nmlzd\ Z\ xs\ fo\_nmlzd\ Z\ ys$   
 $length\ nsx = length\ xs\ length\ nsy = length\ ys$   
 $sorted\_distinct\ nsx\ sorted\_distinct\ nsy$   
 $ns = filter\ (\lambda n.\ n \in set\ nsy)\ nsx$   
 $fo\_nmlz\ Z\ (proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq fo\_nmlz\ Z\ (proj\_tuple\ ns\ (zip\ nsy\ ys))$   
**shows**  $eval\_conj\_tuple\ Z\ nsx\ nsy\ xs\ ys = \{\}$

(proof)

**lemma** *nall\_tuples\_rec\_filter*:

**assumes**  $xs \in nall\_tuples\_rec\ AD\ n\ (length\ xs)\ ys = filter\ (\lambda x.\ \neg isl\ x)\ xs$   
**shows**  $ys \in nall\_tuples\_rec\ \{\}\ n\ (length\ ys)$

(proof)

**lemma** *nall\_tuples\_rec\_filter\_rev*:

**assumes**  $ys \in nall\_tuples\_rec\ \{\}\ n\ (length\ ys)\ ys = filter\ (\lambda x.\ \neg isl\ x)\ xs$   
 $Inl - 'set\ xs \subseteq AD$   
**shows**  $xs \in nall\_tuples\_rec\ AD\ n\ (length\ xs)$

(proof)

**lemma** *eval\_conj\_set\_aux*:

**fixes**  $AD :: 'a\ set$

**assumes**  $ns\varphi\_def: ns\varphi' = filter (\lambda n. n \notin set\ ns\varphi)\ ns\psi$   
**and**  $ns\psi\_def: ns\psi' = filter (\lambda n. n \notin set\ ns\psi)\ ns\varphi$   
**and**  $X\varphi\_def: X\varphi = fo\_nmlz\ AD\ 'proj\_vals\ R\varphi\ ns\varphi$   
**and**  $X\psi\_def: X\psi = fo\_nmlz\ AD\ 'proj\_vals\ R\psi\ ns\psi$   
**and**  $distinct: sorted\_distinct\ ns\varphi\ sorted\_distinct\ ns\psi$   
**and**  $cxs\_def: cxs = filter (\lambda(n, x). n \notin set\ ns\psi \wedge isl\ x)\ (zip\ ns\varphi\ xs)$   
**and**  $nxs\_def: nxs = map\ fst\ (filter (\lambda(n, x). n \notin set\ ns\psi \wedge \neg isl\ x)\ (zip\ ns\varphi\ xs))$   
**and**  $cys\_def: cys = filter (\lambda(n, y). n \notin set\ ns\varphi \wedge isl\ y)\ (zip\ ns\psi\ ys)$   
**and**  $nys\_def: nys = map\ fst\ (filter (\lambda(n, y). n \notin set\ ns\varphi \wedge \neg isl\ y)\ (zip\ ns\psi\ ys))$   
**and**  $xs\_ys\_def: xs \in X\varphi\ ys \in X\psi$   
**and**  $\sigma xs\_def: xs = map\ \sigma xs\ ns\varphi\ fs\varphi = map\ \sigma xs\ ns\varphi'$   
**and**  $\sigma ys\_def: ys = map\ \sigma ys\ ns\psi\ fs\psi = map\ \sigma ys\ ns\psi'$   
**and**  $fs\varphi\_def: fs\varphi \in nall\_tuples\_rec\ AD\ (card\ (Inr\ -\ 'set\ xs))\ (length\ ns\varphi')$   
**and**  $fs\psi\_def: fs\psi \in nall\_tuples\_rec\ AD\ (card\ (Inr\ -\ 'set\ ys))\ (length\ ns\psi')$   
**and**  $ad\_agr: ad\_agr\_list\ AD\ (map\ \sigma ys\ (sort\ (ns\psi\ @\ ns\psi')))\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ ns\varphi')))$   
**shows**  
 $map\ snd\ (merge\ (zip\ ns\varphi\ xs)\ (zip\ ns\varphi'\ fs\varphi)) =$   
 $map\ snd\ (merge\ (zip\ (sort\ (ns\varphi\ @\ map\ fst\ cys))\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))))$   
 $(zip\ nys\ (map\ \sigma xs\ nys))$  **and**  
 $map\ snd\ (merge\ (zip\ ns\varphi\ xs)\ cys) = map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))$  **and**  
 $map\ \sigma xs\ nys \in$   
 $nall\_tuples\_rec\ \{\}\ (card\ (Inr\ -\ 'set\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))))\ (length\ nys)$   
*(proof)*

**lemma** *eval\_conj\_set\_aux'*:

**fixes**  $AD :: 'a\ set$   
**assumes**  $ns\varphi\_def: ns\varphi' = filter (\lambda n. n \notin set\ ns\varphi)\ ns\psi$   
**and**  $ns\psi\_def: ns\psi' = filter (\lambda n. n \notin set\ ns\psi)\ ns\varphi$   
**and**  $X\varphi\_def: X\varphi = fo\_nmlz\ AD\ 'proj\_vals\ R\varphi\ ns\varphi$   
**and**  $X\psi\_def: X\psi = fo\_nmlz\ AD\ 'proj\_vals\ R\psi\ ns\psi$   
**and**  $distinct: sorted\_distinct\ ns\varphi\ sorted\_distinct\ ns\psi$   
**and**  $cxs\_def: cxs = filter (\lambda(n, x). n \notin set\ ns\psi \wedge isl\ x)\ (zip\ ns\varphi\ xs)$   
**and**  $nxs\_def: nxs = map\ fst\ (filter (\lambda(n, x). n \notin set\ ns\psi \wedge \neg isl\ x)\ (zip\ ns\varphi\ xs))$   
**and**  $cys\_def: cys = filter (\lambda(n, y). n \notin set\ ns\varphi \wedge isl\ y)\ (zip\ ns\psi\ ys)$   
**and**  $nys\_def: nys = map\ fst\ (filter (\lambda(n, y). n \notin set\ ns\varphi \wedge \neg isl\ y)\ (zip\ ns\psi\ ys))$   
**and**  $xs\_ys\_def: xs \in X\varphi\ ys \in X\psi$   
**and**  $\sigma xs\_def: xs = map\ \sigma xs\ ns\varphi\ map\ snd\ cys = map\ \sigma xs\ (map\ fst\ cys)$   
 $ys\psi = map\ \sigma xs\ nys$   
**and**  $\sigma ys\_def: ys = map\ \sigma ys\ ns\psi\ map\ snd\ cxs = map\ \sigma ys\ (map\ fst\ cxs)$   
 $xs\varphi = map\ \sigma ys\ nxs$   
**and**  $fs\varphi\_def: fs\varphi = map\ \sigma xs\ ns\varphi'$   
**and**  $fs\psi\_def: fs\psi = map\ \sigma ys\ ns\psi'$   
**and**  $ys\psi\_def: map\ \sigma xs\ nys \in nall\_tuples\_rec\ \{\}$   
 $(card\ (Inr\ -\ 'set\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))))\ (length\ nys)$   
**and**  $Inl\_set\_AD: Inl\ -\ '(set\ (map\ snd\ cxs) \cup set\ xs\varphi) \subseteq AD$   
 $Inl\ -\ '(set\ (map\ snd\ cys) \cup set\ ys\psi) \subseteq AD$   
**and**  $ad\_agr: ad\_agr\_list\ AD\ (map\ \sigma ys\ (sort\ (ns\psi\ @\ ns\psi')))\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ ns\varphi')))$   
**shows**  
 $map\ snd\ (merge\ (zip\ ns\varphi\ xs)\ (zip\ ns\varphi'\ fs\varphi)) =$   
 $map\ snd\ (merge\ (zip\ (sort\ (ns\varphi\ @\ map\ fst\ cys))\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))))$   
 $(zip\ nys\ (map\ \sigma xs\ nys))$  **and**  
 $map\ snd\ (merge\ (zip\ ns\varphi\ xs)\ cys) = map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))$   
 $fs\varphi \in nall\_tuples\_rec\ AD\ (card\ (Inr\ -\ 'set\ xs))\ (length\ ns\varphi')$   
*(proof)*

**lemma** *eval\_conj\_set\_correct*:

**assumes**  $ns\varphi\_def: ns\varphi' = filter (\lambda n. n \notin set\ ns\varphi)\ ns\psi$   
**and**  $ns\psi\_def: ns\psi' = filter (\lambda n. n \notin set\ ns\psi)\ ns\varphi$

**and**  $X\varphi\_def$ :  $X\varphi = fo\_nmlz\ AD\ 'proj\_vals\ R\varphi\ ns\varphi$   
**and**  $X\psi\_def$ :  $X\psi = fo\_nmlz\ AD\ 'proj\_vals\ R\psi\ ns\psi$   
**and**  $distinct$ :  $sorted\_distinct\ ns\varphi\ sorted\_distinct\ ns\psi$   
**shows**  $eval\_conj\_set\ AD\ ns\varphi\ X\varphi\ ns\psi\ X\psi = ext\_tuple\_set\ AD\ ns\varphi\ ns\psi\ X\varphi \cap ext\_tuple\_set\ AD\ ns\psi\ X\psi$   
 $\langle proof \rangle$

**lemma**  $esat\_exists\_not\_fv$ :  $n \notin fv\_fo\_fmla\ \varphi \implies X \neq \{\} \implies$   
 $esat\ (Exists\ n\ \varphi)\ I\ \sigma\ X \longleftrightarrow esat\ \varphi\ I\ \sigma\ X$   
 $\langle proof \rangle$

**lemma**  $esat\_forall\_not\_fv$ :  $n \notin fv\_fo\_fmla\ \varphi \implies X \neq \{\} \implies$   
 $esat\ (Forall\ n\ \varphi)\ I\ \sigma\ X \longleftrightarrow esat\ \varphi\ I\ \sigma\ X$   
 $\langle proof \rangle$

**lemma**  $proj\_sat\_vals$ :  $proj\_sat\ \varphi\ I =$   
 $proj\_vals\ \{\sigma.\ sat\ \varphi\ I\ \sigma\}\ (fv\_fo\_fmla\_list\ \varphi)$   
 $\langle proof \rangle$

**lemma**  $fv\_fo\_fmla\_list\_Pred$ :  $remdups\_adj\ (sort\ (fv\_fo\_terms\_list\ ts)) = fv\_fo\_terms\_list\ ts$   
 $\langle proof \rangle$

**lemma**  $ad\_agr\_list\_fv\_list'$ :  $\bigcup (set\ (map\ set\_fo\_term\ ts)) \subseteq X \implies$   
 $ad\_agr\_list\ X\ (map\ \sigma\ (fv\_fo\_terms\_list\ ts))\ (map\ \tau\ (fv\_fo\_terms\_list\ ts)) \implies$   
 $ad\_agr\_list\ X\ (\sigma \odot e\ ts)\ (\tau \odot e\ ts)$   
 $\langle proof \rangle$

**lemma**  $ext\_tuple\_ad\_agr\_close$ :  
**assumes**  $S\varphi\_def$ :  $S\varphi \equiv \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$   
**and**  $AD\_sub$ :  $act\_edom\ \varphi\ I \subseteq AD\varphi\ AD\varphi \subseteq AD$   
**and**  $X\varphi\_def$ :  $X\varphi = fo\_nmlz\ AD\varphi\ 'proj\_vals\ S\varphi\ (fv\_fo\_fmla\_list\ \varphi)$   
**and**  $ns\varphi'\_def$ :  $ns\varphi' = filter\ (\lambda n.\ n \notin fv\_fo\_fmla\ \varphi)\ ns\psi$   
**and**  $sd\_ns\psi$ :  $sorted\_distinct\ ns\psi$   
**and**  $fv\_Un$ :  $fv\_fo\_fmla\ \psi = fv\_fo\_fmla\ \varphi \cup set\ ns\psi$   
**shows**  $ext\_tuple\_set\ AD\ (fv\_fo\_fmla\_list\ \varphi)\ ns\varphi'\ (ad\_agr\_close\_set\ (AD - AD\varphi)\ X\varphi) =$   
 $fo\_nmlz\ AD\ 'proj\_vals\ S\varphi\ (fv\_fo\_fmla\_list\ \psi)$   
 $ad\_agr\_close\_set\ (AD - AD\varphi)\ X\varphi = fo\_nmlz\ AD\ 'proj\_vals\ S\varphi\ (fv\_fo\_fmla\_list\ \varphi)$   
 $\langle proof \rangle$

**lemma**  $proj\_ext\_tuple$ :  
**assumes**  $S\varphi\_def$ :  $S\varphi \equiv \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$   
**and**  $AD\_sub$ :  $act\_edom\ \varphi\ I \subseteq AD$   
**and**  $X\varphi\_def$ :  $X\varphi = fo\_nmlz\ AD\ 'proj\_vals\ S\varphi\ (fv\_fo\_fmla\_list\ \varphi)$   
**and**  $ns\varphi'\_def$ :  $ns\varphi' = filter\ (\lambda n.\ n \notin fv\_fo\_fmla\ \varphi)\ ns\psi$   
**and**  $sd\_ns\psi$ :  $sorted\_distinct\ ns\psi$   
**and**  $fv\_Un$ :  $fv\_fo\_fmla\ \psi = fv\_fo\_fmla\ \varphi \cup set\ ns\psi$   
**and**  $Z\_props$ :  $\bigwedge xs.\ xs \in Z \implies fo\_nmlz\ AD\ xs = xs \wedge length\ xs = length\ (fv\_fo\_fmla\_list\ \psi)$   
**shows**  $Z \cap ext\_tuple\_set\ AD\ (fv\_fo\_fmla\_list\ \varphi)\ ns\varphi'\ X\varphi =$   
 $\{xs \in Z.\ fo\_nmlz\ AD\ (proj\_tuple\ (fv\_fo\_fmla\_list\ \varphi)\ (zip\ (fv\_fo\_fmla\_list\ \psi)\ xs)) \in X\varphi\}$   
 $Z - ext\_tuple\_set\ AD\ (fv\_fo\_fmla\_list\ \varphi)\ ns\varphi'\ X\varphi =$   
 $\{xs \in Z.\ fo\_nmlz\ AD\ (proj\_tuple\ (fv\_fo\_fmla\_list\ \varphi)\ (zip\ (fv\_fo\_fmla\_list\ \psi)\ xs)) \notin X\varphi\}$   
 $\langle proof \rangle$

**lemma**  $fo\_nmlz\_proj\_sub$ :  $fo\_nmlz\ AD\ 'proj\_fmla\ \varphi\ R \subseteq nall\_tuples\ AD\ (nfv\ \varphi)$   
 $\langle proof \rangle$

**lemma**  $fin\_ad\_agr\_list\_iff$ :  
**fixes**  $AD :: ('a :: infinite)\ set$

**assumes**  $finite\ AD \wedge vs. vs \in Z \implies length\ vs = n$   
 $Z = \{ts. \exists ts' \in X. ad\_agr\_list\ AD\ (map\ Inl\ ts)\ ts'\}$   
**shows**  $finite\ Z \iff \bigcup (set\ 'Z) \subseteq AD$   
 $\langle proof \rangle$

**lemma** *proj\_out\_list*:  
**fixes**  $AD :: ('a :: infinite)\ set$   
**and**  $\sigma :: nat \Rightarrow 'a + nat$   
**and**  $ns :: nat\ list$   
**assumes**  $finite\ AD$   
**shows**  $\exists \tau. ad\_agr\_list\ AD\ (map\ \sigma\ ns)\ (map\ (Inl \circ \tau)\ ns) \wedge$   
 $(\forall j\ x. j \in set\ ns \longrightarrow \sigma\ j = Inl\ x \longrightarrow \tau\ j = x)$   
 $\langle proof \rangle$

**lemma** *proj\_out*:  
**fixes**  $\varphi :: ('a :: infinite, 'b)\ fo\_fmla$   
**and**  $J :: (('a, nat)\ fo\_t, 'b)\ fo\_intp$   
**assumes**  $wf\_fo\_intp\ \varphi\ I\ esat\ \varphi\ I\ \sigma\ UNIV$   
**shows**  $\exists \tau. esat\ \varphi\ I\ (Inl \circ \tau)\ UNIV \wedge (\forall i\ x. i \in fv\_fo\_fmla\ \varphi \wedge \sigma\ i = Inl\ x \longrightarrow \tau\ i = x) \wedge$   
 $ad\_agr\_list\ (act\_edom\ \varphi\ I)\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \varphi))\ (map\ (Inl \circ \tau)\ (fv\_fo\_fmla\_list\ \varphi))$   
 $\langle proof \rangle$

**lemma** *proj\_fmla\_esat\_sat*:  
**fixes**  $\varphi :: ('a :: infinite, 'b)\ fo\_fmla$   
**and**  $J :: (('a, nat)\ fo\_t, 'b)\ fo\_intp$   
**assumes**  $wf: wf\_fo\_intp\ \varphi\ I$   
**shows**  $proj\_fmla\ \varphi\ \{\sigma. esat\ \varphi\ I\ \sigma\ UNIV\} \cap map\ Inl\ 'UNIV =$   
 $map\ Inl\ 'proj\_fmla\ \varphi\ \{\sigma. sat\ \varphi\ I\ \sigma\}$   
 $\langle proof \rangle$

**lemma** *norm\_proj\_fmla\_esat\_sat*:  
**fixes**  $\varphi :: ('a :: infinite, 'b)\ fo\_fmla$   
**assumes**  $wf\_fo\_intp\ \varphi\ I$   
**shows**  $fo\_nmlz\ (act\_edom\ \varphi\ I)\ 'proj\_fmla\ \varphi\ \{\sigma. esat\ \varphi\ I\ \sigma\ UNIV\} =$   
 $fo\_nmlz\ (act\_edom\ \varphi\ I)\ 'map\ Inl\ 'proj\_fmla\ \varphi\ \{\sigma. sat\ \varphi\ I\ \sigma\}$   
 $\langle proof \rangle$

**lemma** *proj\_sat\_fmla*:  $proj\_sat\ \varphi\ I = proj\_fmla\ \varphi\ \{\sigma. sat\ \varphi\ I\ \sigma\}$   
 $\langle proof \rangle$

**fun** *fo\_wf* ::  $('a, 'b)\ fo\_fmla \Rightarrow ('b \times nat \Rightarrow 'a\ list\ set) \Rightarrow ('a, nat)\ fo\_t \Rightarrow bool$  **where**  
 $fo\_wf\ \varphi\ I\ (AD, n, X) \iff finite\ AD \wedge finite\ X \wedge n = nfv\ \varphi \wedge$   
 $wf\_fo\_intp\ \varphi\ I \wedge AD = act\_edom\ \varphi\ I \wedge fo\_rep\ (AD, n, X) = proj\_sat\ \varphi\ I \wedge$   
 $Inl\ -' \bigcup (set\ 'X) \subseteq AD \wedge (\forall vs \in X. fo\_nmlzd\ AD\ vs \wedge length\ vs = n)$

**fun** *fo\_fin* ::  $('a, nat)\ fo\_t \Rightarrow bool$  **where**  
 $fo\_fin\ (AD, n, X) \iff (\forall x \in \bigcup (set\ 'X). isl\ x)$

**lemma** *fo\_rep\_fin*:  
**assumes**  $fo\_wf\ \varphi\ I\ (AD, n, X)\ fo\_fin\ (AD, n, X)$   
**shows**  $fo\_rep\ (AD, n, X) = map\ projl\ 'X$   
 $\langle proof \rangle$

**definition** *eval\_abs* ::  $('a, 'b)\ fo\_fmla \Rightarrow ('a\ table, 'b)\ fo\_intp \Rightarrow ('a, nat)\ fo\_t$  **where**  
 $eval\_abs\ \varphi\ I = (act\_edom\ \varphi\ I, nfv\ \varphi, fo\_nmlz\ (act\_edom\ \varphi\ I)\ 'proj\_fmla\ \varphi\ \{\sigma. esat\ \varphi\ I\ \sigma\ UNIV\})$

**lemma** *map\_projl\_Inl*:  $map\ projl\ (map\ Inl\ xs) = xs$   
 $\langle proof \rangle$

**lemma fo\_rep\_eval\_abs:**  
**fixes**  $\varphi :: ('a :: \text{infinite}, 'b) \text{ fo\_fmula}$   
**assumes**  $\text{wf\_fo\_intp } \varphi I$   
**shows**  $\text{fo\_rep } (\text{eval\_abs } \varphi I) = \text{proj\_sat } \varphi I$   
 $\langle \text{proof} \rangle$

**lemma fo\_wf\_eval\_abs:**  
**fixes**  $\varphi :: ('a :: \text{infinite}, 'b) \text{ fo\_fmula}$   
**assumes**  $\text{wf\_fo\_intp } \varphi I$   
**shows**  $\text{fo\_wf } \varphi I (\text{eval\_abs } \varphi I)$   
 $\langle \text{proof} \rangle$

**lemma fo\_fin:**  
**fixes**  $t :: ('a :: \text{infinite}, \text{nat}) \text{ fo\_t}$   
**assumes**  $\text{fo\_wf } \varphi I t$   
**shows**  $\text{fo\_fin } t = \text{finite } (\text{fo\_rep } t)$   
 $\langle \text{proof} \rangle$

**lemma eval\_pred:**  
**fixes**  $I :: 'b \times \text{nat} \Rightarrow 'a :: \text{infinite list set}$   
**assumes**  $\text{finite } (I (r, \text{length } ts))$   
**shows**  $\text{fo\_wf } (\text{Pred } r ts) I (\text{eval\_pred } ts (I (r, \text{length } ts)))$   
 $\langle \text{proof} \rangle$

**lemma ad\_agr\_list\_eval:**  $\bigcup (\text{set } (\text{map } \text{set\_fo\_term } ts)) \subseteq AD \Longrightarrow \text{ad\_agr\_list } AD (\sigma \odot e ts) zs \Longrightarrow$   
 $\exists \tau. zs = \tau \odot e ts$   
 $\langle \text{proof} \rangle$

**lemma sp\_equiv\_list\_fv\_list:**  
**assumes**  $\text{sp\_equiv\_list } (\sigma \odot e ts) (\tau \odot e ts)$   
**shows**  $\text{sp\_equiv\_list } (\text{map } \sigma (\text{fv\_fo\_terms\_list } ts)) (\text{map } \tau (\text{fv\_fo\_terms\_list } ts))$   
 $\langle \text{proof} \rangle$

**lemma ad\_agr\_list\_fv\_list:**  $\text{ad\_agr\_list } X (\sigma \odot e ts) (\tau \odot e ts) \Longrightarrow$   
 $\text{ad\_agr\_list } X (\text{map } \sigma (\text{fv\_fo\_terms\_list } ts)) (\text{map } \tau (\text{fv\_fo\_terms\_list } ts))$   
 $\langle \text{proof} \rangle$

**lemma eval\_bool:**  $\text{fo\_wf } (\text{Bool } b) I (\text{eval\_bool } b)$   
 $\langle \text{proof} \rangle$

**lemma eval\_eq:** **fixes**  $I :: 'b \times \text{nat} \Rightarrow 'a :: \text{infinite list set}$   
**shows**  $\text{fo\_wf } (\text{Eq } t t') I (\text{eval\_eq } t t')$   
 $\langle \text{proof} \rangle$

**lemma fv\_fo\_terms\_list\_Var:**  $\text{fv\_fo\_terms\_list\_rec } (\text{map } \text{Var } ns) = ns$   
 $\langle \text{proof} \rangle$

**lemma eval\_eterms\_map\_Var:**  $\sigma \odot e \text{ map } \text{Var } ns = \text{map } \sigma ns$   
 $\langle \text{proof} \rangle$

**lemma fo\_wf\_eval\_table:**  
**fixes**  $AD :: 'a \text{ set}$   
**assumes**  $\text{fo\_wf } \varphi I (AD, n, X)$   
**shows**  $X = \text{fo\_nmlz } AD \text{ 'eval\_table } (\text{map } \text{Var } [0..<n]) X$   
 $\langle \text{proof} \rangle$

**lemma fo\_rep\_norm:**

**fixes**  $AD :: ('a :: infinite) set$   
**assumes**  $fo\_wf \ \varphi \ I \ (AD, n, X)$   
**shows**  $X = fo\_nmlz \ AD \ ' \ map \ Inl \ ' \ fo\_rep \ (AD, n, X)$   
 $\langle proof \rangle$

**lemma**  $fo\_wf\_X$ :  
**fixes**  $\varphi :: ('a :: infinite, 'b) fo\_fmla$   
**assumes**  $wf: fo\_wf \ \varphi \ I \ (AD, n, X)$   
**shows**  $X = fo\_nmlz \ AD \ ' \ proj\_fmla \ \varphi \ \{\sigma. \ esat \ \varphi \ I \ \sigma \ UNIV\}$   
 $\langle proof \rangle$

**lemma**  $eval\_neg$ :  
**fixes**  $\varphi :: ('a :: infinite, 'b) fo\_fmla$   
**assumes**  $wf: fo\_wf \ \varphi \ I \ t$   
**shows**  $fo\_wf \ (Neg \ \varphi) \ I \ (eval\_neg \ (fv\_fo\_fmla\_list \ \varphi) \ t)$   
 $\langle proof \rangle$

**definition**  $cross\_with \ f \ t \ t' = \bigcup((\lambda xs. \bigcup(f \ xs \ ' \ t')) \ ' \ t)$

**lemma**  $mapping\_join\_cross\_with$ :  
**assumes**  $\bigwedge x \ x'. \ x \in t \implies x' \in t' \implies h \ x \neq h' \ x' \implies f \ x \ x' = \{\}$   
**shows**  $set\_of\_idx \ (mapping\_join \ (cross\_with \ f) \ (cluster \ (Some \circ \ h) \ t) \ (cluster \ (Some \circ \ h') \ t')) = cross\_with \ f \ t \ t'$   
 $\langle proof \rangle$

**lemma**  $fo\_nmlzd\_mono\_sub: X \subseteq X' \implies fo\_nmlzd \ X \ xs \implies fo\_nmlzd \ X' \ xs$   
 $\langle proof \rangle$

**lemma**  $idx\_join$ :  
**assumes**  $X\varphi\_props: \bigwedge vs. \ vs \in X\varphi \implies fo\_nmlzd \ AD \ vs \wedge length \ vs = length \ ns\varphi$   
**assumes**  $X\psi\_props: \bigwedge vs. \ vs \in X\psi \implies fo\_nmlzd \ AD \ vs \wedge length \ vs = length \ ns\psi$   
**assumes**  $sd\_ns: sorted\_distinct \ ns\varphi \ sorted\_distinct \ ns\psi$   
**assumes**  $ns\_def: ns = filter \ (\lambda n. \ n \in set \ ns\psi) \ ns\varphi$   
**shows**  $idx\_join \ AD \ ns \ ns\varphi \ X\varphi \ ns\psi \ X\psi = eval\_conj\_set \ AD \ ns\varphi \ X\varphi \ ns\psi \ X\psi$   
 $\langle proof \rangle$

**lemma**  $proj\_fmla\_conj\_sub$ :  
**assumes**  $AD\_sub: act\_edom \ \psi \ I \subseteq AD$   
**shows**  $fo\_nmlz \ AD \ ' \ proj\_fmla \ (Conj \ \varphi \ \psi) \ \{\sigma. \ esat \ \varphi \ I \ \sigma \ UNIV\} \cap fo\_nmlz \ AD \ ' \ proj\_fmla \ (Conj \ \varphi \ \psi) \ \{\sigma. \ esat \ \psi \ I \ \sigma \ UNIV\} \subseteq fo\_nmlz \ AD \ ' \ proj\_fmla \ (Conj \ \varphi \ \psi) \ \{\sigma. \ esat \ (Conj \ \varphi \ \psi) \ I \ \sigma \ UNIV\}$   
 $\langle proof \rangle$

**lemma**  $eval\_conj$ :  
**fixes**  $\varphi :: ('a :: infinite, 'b) fo\_fmla$   
**assumes**  $wf: fo\_wf \ \varphi \ I \ t\varphi \ fo\_wf \ \psi \ I \ t\psi$   
**shows**  $fo\_wf \ (Conj \ \varphi \ \psi) \ I \ (eval\_conj \ (fv\_fo\_fmla\_list \ \varphi) \ t\varphi \ (fv\_fo\_fmla\_list \ \psi) \ t\psi)$   
 $\langle proof \rangle$

**lemma**  $map\_values\_cluster: (\bigwedge w \ z \ Z. \ Z \subseteq X \implies z \in Z \implies w \in f \ (h \ z) \ \{z\} \implies w \in f \ (h \ z) \ Z) \implies (\bigwedge w \ z \ Z. \ Z \subseteq X \implies z \in Z \implies w \in f \ (h \ z) \ Z \implies (\exists z' \in Z. \ w \in f \ (h \ z) \ \{z'\})) \implies set\_of\_idx \ (Mapping.map\_values \ f \ (cluster \ (Some \circ \ h) \ X)) = \bigcup((\lambda x. \ f \ (h \ x) \ \{x\}) \ ' \ X)$   
 $\langle proof \rangle$

**lemma**  $fo\_nmlz\_twice$ :  
**assumes**  $sorted\_distinct \ ns \ sorted\_distinct \ ns' \ set \ ns \subseteq set \ ns'$   
**shows**  $fo\_nmlz \ AD \ (proj\_tuple \ ns \ (zip \ ns' \ (fo\_nmlz \ AD \ (map \ \sigma \ ns')))) = fo\_nmlz \ AD \ (map \ \sigma \ ns)$   
 $\langle proof \rangle$

**lemma** *map\_values\_cong*:

**assumes**  $\bigwedge x y. \text{Mapping.lookup } t x = \text{Some } y \implies f x y = f' x y$

**shows**  $\text{Mapping.map\_values } f t = \text{Mapping.map\_values } f' t$

*<proof>*

**lemma** *ad\_agr\_close\_set\_length*:  $z \in \text{ad\_agr\_close\_set } AD X \implies (\bigwedge x. x \in X \implies \text{length } x = n) \implies \text{length } z = n$

*<proof>*

**lemma** *ad\_agr\_close\_set\_sound*:  $z \in \text{ad\_agr\_close\_set } (AD - AD') X \implies (\bigwedge x. x \in X \implies \text{fo\_nmlzd } AD' x) \implies AD' \subseteq AD \implies \text{fo\_nmlzd } AD z$

*<proof>*

**lemma** *ext\_tuple\_set\_length*:  $z \in \text{ext\_tuple\_set } AD ns ns' X \implies (\bigwedge x. x \in X \implies \text{length } x = \text{length } ns) \implies \text{length } z = \text{length } ns + \text{length } ns'$

*<proof>*

**lemma** *eval\_ajoin*:

**fixes**  $\varphi :: ('a :: \text{infinite}, 'b) \text{fo\_fmla}$

**assumes**  $\text{wf}: \text{fo\_wf } \varphi I t\varphi \text{fo\_wf } \psi I t\psi$

**shows**  $\text{fo\_wf } (\text{Conj } \varphi (\text{Neg } \psi)) I$

$(\text{eval\_ajoin } (\text{fv\_fo\_fmla\_list } \varphi) t\varphi (\text{fv\_fo\_fmla\_list } \psi) t\psi)$

*<proof>*

**lemma** *eval\_disj*:

**fixes**  $\varphi :: ('a :: \text{infinite}, 'b) \text{fo\_fmla}$

**assumes**  $\text{wf}: \text{fo\_wf } \varphi I t\varphi \text{fo\_wf } \psi I t\psi$

**shows**  $\text{fo\_wf } (\text{Disj } \varphi \psi) I$

$(\text{eval\_disj } (\text{fv\_fo\_fmla\_list } \varphi) t\varphi (\text{fv\_fo\_fmla\_list } \psi) t\psi)$

*<proof>*

**lemma** *fv\_ex\_all*:

**assumes**  $\text{pos } i (\text{fv\_fo\_fmla\_list } \varphi) = \text{None}$

**shows**  $\text{fv\_fo\_fmla\_list } (\text{Exists } i \varphi) = \text{fv\_fo\_fmla\_list } \varphi$

$\text{fv\_fo\_fmla\_list } (\text{Forall } i \varphi) = \text{fv\_fo\_fmla\_list } \varphi$

*<proof>*

**lemma** *nfv\_ex\_all*:

**assumes**  $\text{Some } i (\text{fv\_fo\_fmla\_list } \varphi) = \text{Some } j$

**shows**  $\text{nfv } \varphi = \text{Suc } (\text{nfv } (\text{Exists } i \varphi)) \text{nfv } \varphi = \text{Suc } (\text{nfv } (\text{Forall } i \varphi))$

*<proof>*

**lemma** *fv\_fo\_fmla\_list\_exists*:  $\text{fv\_fo\_fmla\_list } (\text{Exists } n \varphi) = \text{filter } ((\neq) n) (\text{fv\_fo\_fmla\_list } \varphi)$

*<proof>*

**lemma** *eval\_exists*:

**fixes**  $\varphi :: ('a :: \text{infinite}, 'b) \text{fo\_fmla}$

**assumes**  $\text{wf}: \text{fo\_wf } \varphi I t$

**shows**  $\text{fo\_wf } (\text{Exists } i \varphi) I (\text{eval\_exists } i (\text{fv\_fo\_fmla\_list } \varphi) t)$

*<proof>*

**lemma** *fv\_fo\_fmla\_list\_forall*:  $\text{fv\_fo\_fmla\_list } (\text{Forall } n \varphi) = \text{filter } ((\neq) n) (\text{fv\_fo\_fmla\_list } \varphi)$

*<proof>*

**lemma** *pairwise\_take\_drop*:

**assumes**  $\text{pairwise } P (\text{set } (\text{zip } xs ys)) \text{length } xs = \text{length } ys$

**shows**  $\text{pairwise } P (\text{set } (\text{zip } (\text{take } i xs @ \text{drop } (\text{Suc } i) xs) (\text{take } i ys @ \text{drop } (\text{Suc } i) ys)))$

*<proof>*

**lemma** *fo\_nmlz\_set\_card*:

*fo\_nmlz AD xs = xs  $\implies$  set xs = set xs  $\cap$  Inl ' AD  $\cup$  Inr ' {..*card* (Inr -' set xs)}*

*<proof>*

**lemma** *ad\_agr\_list\_take\_drop*: *ad\_agr\_list AD xs ys  $\implies$*

*ad\_agr\_list AD (take i xs @ drop (Suc i) xs) (take i ys @ drop (Suc i) ys)*

*<proof>*

**lemma** *fo\_nmlz\_rem\_nth\_add\_nth*:

**assumes** *fo\_nmlz AD zs = zs i  $\leq$  length zs*

**shows** *fo\_nmlz AD (rem\_nth i (fo\_nmlz AD (add\_nth i z zs))) = zs*

*<proof>*

**lemma** *ad\_agr\_list\_add*:

**assumes** *ad\_agr\_list AD xs ys i  $\leq$  length xs*

**shows**  $\exists z' \in \text{Inl ' AD} \cup \text{Inr ' \{..*Suc* (*card* (Inr -' set ys))\}} \cup \text{set ys.}$

*ad\_agr\_list AD (take i xs @ z # drop i xs) (take i ys @ z' # drop i ys)*

*<proof>*

**lemma** *add\_nth\_restrict*:

**assumes** *fo\_nmlz AD zs = zs i  $\leq$  length zs*

**shows**  $\exists z' \in \text{Inl ' AD} \cup \text{Inr ' \{..*Suc* (*card* (Inr -' set zs))\}}.$

*fo\_nmlz AD (add\_nth i z zs) = fo\_nmlz AD (add\_nth i z' zs)*

*<proof>*

**lemma** *fo\_nmlz\_add\_rem*:

**assumes** *i  $\leq$  length zs*

**shows**  $\exists z'. \text{fo\_nmlz AD (add\_nth i z zs) = fo\_nmlz AD (add\_nth i z' (fo\_nmlz AD zs))}$

*<proof>*

**lemma** *fo\_nmlz\_add\_rem'*:

**assumes** *i  $\leq$  length zs*

**shows**  $\exists z'. \text{fo\_nmlz AD (add\_nth i z (fo\_nmlz AD zs)) = fo\_nmlz AD (add\_nth i z' zs)}$

*<proof>*

**lemma** *fo\_nmlz\_add\_nth\_rem\_nth*:

**assumes** *fo\_nmlz AD xs = xs i < length xs*

**shows**  $\exists z. \text{fo\_nmlz AD (add\_nth i z (fo\_nmlz AD (rem\_nth i xs))) = xs}$

*<proof>*

**lemma** *sp\_equiv\_list\_almost\_same*: *sp\_equiv\_list (xs @ v # ys) (xs @ w # ys)  $\implies$*

*v  $\in$  set xs  $\cup$  set ys  $\vee$  w  $\in$  set xs  $\cup$  set ys  $\implies$  v = w*

*<proof>*

**lemma** *ad\_agr\_list\_add\_nth*:

**assumes** *i  $\leq$  length zs ad\_agr\_list AD (add\_nth i v zs) (add\_nth i w zs) v  $\neq$  w*

**shows**  $\{v, w\} \cap (\text{Inl ' AD} \cup \text{set zs}) = \{\}$

*<proof>*

**lemma** *Inr\_in\_tuple*:

**assumes** *fo\_nmlz AD zs = zs n < card (Inr -' set zs)*

**shows** *Inr n  $\in$  set zs*

*<proof>*

**lemma** *card\_wit\_sub*:

**assumes** *finite Z card Z  $\leq$  card {ts  $\in$  X.  $\exists z \in Z. ts = f z$ }*

**shows**  $f \text{ ' } Z \subseteq X$   
 ⟨proof⟩

**lemma** *add\_nth\_iff\_card*:

**assumes**  $(\bigwedge xs. xs \in X \implies fo\_nmlz\ AD\ xs = xs)$   $(\bigwedge xs. xs \in X \implies i < length\ xs)$   
 $fo\_nmlz\ AD\ zs = zs\ i \leq length\ zs\ finite\ AD\ finite\ X$

**shows**  $(\forall z. fo\_nmlz\ AD\ (add\_nth\ i\ z\ zs) \in X) \longleftrightarrow$

$Suc\ (card\ AD + card\ (Inr\ \text{' } set\ zs)) \leq card\ \{ts \in X. \exists z. ts = fo\_nmlz\ AD\ (add\_nth\ i\ z\ zs)\}$   
 ⟨proof⟩

**lemma** *set\_fo\_nmlz\_add\_nth\_rem\_nth*:

**assumes**  $j < length\ xs\ \bigwedge x. x \in X \implies fo\_nmlz\ AD\ x = x$

$\bigwedge x. x \in X \implies j < length\ x$

**shows**  $\{ts \in X. \exists z. ts = fo\_nmlz\ AD\ (add\_nth\ j\ z\ (fo\_nmlz\ AD\ (rem\_nth\ j\ xs)))\} =$   
 $\{y \in X. fo\_nmlz\ AD\ (rem\_nth\ j\ y) = fo\_nmlz\ AD\ (rem\_nth\ j\ xs)\}$

⟨proof⟩

**lemma** *eval\_forall*:

**fixes**  $\varphi :: ('a :: infinite, 'b)\ fo\_fmla$

**assumes**  $wf: fo\_wf\ \varphi\ I\ t$

**shows**  $fo\_wf\ (Forall\ i\ \varphi)\ I\ (eval\_forall\ i\ (fv\_fo\_fmla\_list\ \varphi)\ t)$

⟨proof⟩

**fun** *fo\_res* ::  $('a, nat)\ fo\_t \Rightarrow 'a\ eval\_res$  **where**

$fo\_res\ (AD, n, X) = (if\ fo\_fin\ (AD, n, X)\ then\ Fin\ (map\ projl\ \text{' } X)\ else\ Infn)$

**lemma** *fo\_res\_fin*:

**fixes**  $t :: ('a :: infinite, nat)\ fo\_t$

**assumes**  $fo\_wf\ \varphi\ I\ t\ finite\ (fo\_rep\ t)$

**shows**  $fo\_res\ t = Fin\ (fo\_rep\ t)$

⟨proof⟩

**lemma** *fo\_res\_infn*:

**fixes**  $t :: ('a :: infinite, nat)\ fo\_t$

**assumes**  $fo\_wf\ \varphi\ I\ t\ \neg finite\ (fo\_rep\ t)$

**shows**  $fo\_res\ t = Infn$

⟨proof⟩

**lemma** *fo\_rep*:  $fo\_wf\ \varphi\ I\ t \implies fo\_rep\ t = proj\_sat\ \varphi\ I$

⟨proof⟩

**global\\_interpretation** *Ailamazyan*:  $eval\_fo\ fo\_wf\ eval\_pred\ fo\_rep\ fo\_res$

$eval\_bool\ eval\_eq\ eval\_neg\ eval\_conj\ eval\_ajoin\ eval\_disj$

$eval\_exists\ eval\_forall$

**defines**  $eval\_fmla = Ailamazyan.eval\_fmla$

**and**  $eval = Ailamazyan.eval$

⟨proof⟩

**definition** *esat\_UNIV* ::  $('a :: infinite, 'b)\ fo\_fmla \Rightarrow ('a\ table, 'b)\ fo\_intp \Rightarrow ('a + nat)\ val \Rightarrow bool$

**where**

$esat\_UNIV\ \varphi\ I\ \sigma = esat\ \varphi\ I\ \sigma\ UNIV$

**lemma** *esat\_UNIV\_code*[code]:  $esat\_UNIV\ \varphi\ I\ \sigma \longleftrightarrow (if\ wf\_fo\_intp\ \varphi\ I\ then$

$(case\ eval\_fmla\ \varphi\ I\ of\ (AD, n, X) \Rightarrow$

$fo\_nmlz\ (act\_edom\ \varphi\ I)\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \varphi)) \in X)$

$else\ esat\_UNIV\ \varphi\ I\ \sigma)$

⟨proof⟩

**lemma** *sat\_code*[code]:

**fixes**  $\varphi :: ('a :: \text{infinite}, 'b) \text{fo\_fmla}$   
**shows**  $\text{sat } \varphi I \sigma \longleftrightarrow (\text{if } \text{wf\_fo\_intp } \varphi I \text{ then}$   
 $(\text{case } \text{eval\_fmla } \varphi I \text{ of } (AD, n, X) \Rightarrow$   
 $\text{fo\_nmlz } (\text{act\_edom } \varphi I) (\text{map } (\text{Inl} \circ \sigma) (\text{fv\_fo\_fmla\_list } \varphi)) \in X)$   
 $\text{else } \text{sat } \varphi I \sigma)$   
*<proof>*

**end**

**theory** *Ailamazyan\_Code*

**imports** *HOL-Library.Code\_Target\_Nat Containers.Containers Ailamazyan*

**begin**

**definition** *insert\_db* ::  $'a \Rightarrow 'b \Rightarrow ('a, 'b \text{ set}) \text{ mapping} \Rightarrow ('a, 'b \text{ set}) \text{ mapping}$  **where**

$\text{insert\_db } k v m = (\text{case } \text{Mapping.lookup } m k \text{ of } \text{None} \Rightarrow$   
 $\text{Mapping.update } k (\{v\}) m$   
 $| \text{Some } vs \Rightarrow \text{Mapping.update } k (\{v\} \cup vs) m)$

**fun** *convert\_db\_rec* ::  $('a \times 'c \text{ list}) \text{ list} \Rightarrow (('a \times \text{nat}), 'c \text{ list set}) \text{ mapping} \Rightarrow$

$(('a \times \text{nat}), 'c \text{ list set}) \text{ mapping}$  **where**  
 $\text{convert\_db\_rec } [] m = m$

$| \text{convert\_db\_rec } ((r, ts) \# ktss) m = \text{convert\_db\_rec } ktss (\text{insert\_db } (r, \text{length } ts) ts m)$

**lemma** *convert\_db\_rec\_mono*:  $\text{Mapping.lookup } m (r, n) = \text{Some } tss \Longrightarrow$

$\exists tss'. \text{Mapping.lookup } (\text{convert\_db\_rec } ktss m) (r, n) = \text{Some } tss' \wedge tss \subseteq tss'$   
*<proof>*

**lemma** *convert\_db\_rec\_sound*:  $(r, ts) \in \text{set } ktss \Longrightarrow$

$\exists tss. \text{Mapping.lookup } (\text{convert\_db\_rec } ktss m) (r, \text{length } ts) = \text{Some } tss \wedge ts \in tss$   
*<proof>*

**lemma** *convert\_db\_rec\_complete*:  $\text{Mapping.lookup } (\text{convert\_db\_rec } ktss m) (r, n) = \text{Some } tss' \Longrightarrow$

$ts \in tss' \Longrightarrow$   
 $(\text{length } ts = n \wedge (r, ts) \in \text{set } ktss) \vee (\exists tss. \text{Mapping.lookup } m (r, n) = \text{Some } tss \wedge ts \in tss)$   
*<proof>*

**definition** *convert\_db* ::  $('a \times 'c \text{ list}) \text{ list} \Rightarrow ('c \text{ table}, 'a) \text{fo\_intp}$  **where**

$\text{convert\_db } ktss = (\text{let } m = \text{convert\_db\_rec } ktss \text{ Mapping.empty in}$   
 $(\lambda x. \text{case } \text{Mapping.lookup } m x \text{ of } \text{None} \Rightarrow \{\} | \text{Some } v \Rightarrow v))$

**lemma** *convert\_db\_correct*:  $(ts \in \text{convert\_db } ktss (r, n) \longrightarrow n = \text{length } ts) \wedge$

$((r, ts) \in \text{set } ktss \longleftrightarrow ts \in \text{convert\_db } ktss (r, \text{length } ts))$   
*<proof>*

**lemma** *Inl\_vimage\_set\_code*[code\_unfold]:  $\text{Inl} - ' \text{ set } as = \text{set } (\text{List.map\_filter } (\text{case\_sum } \text{Some } \text{Map.empty})$   
 $as)$

*<proof>*

**lemma** *Inr\_vimage\_set\_code*[code\_unfold]:  $\text{Inr} - ' \text{ set } as = \text{set } (\text{List.map\_filter } (\text{case\_sum } \text{Map.empty}$   
 $\text{Some}) as)$

*<proof>*

**lemma** *Inl\_vimage\_code*:  $\text{Inl} - ' as = \text{projl} - ' \{x \in as. \text{isl } x\}$

*<proof>*



(*String.implode* "Q", [20, 42]),  
(*String.implode* "Q", [30, 43]))

end

## References

- [1] A. K. Ailamazyan, M. M. Gilula, A. P. Stolboushkin, and G. F. Schwartz. Reduction of a relational model with infinite domains to the case of finite domains. *Dokl. Akad. Nauk SSSR*, 286:308–311, 1986.
- [2] A. Avron and Y. Hirshfeld. On first order database query languages. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 226–231. IEEE Computer Society, 1991.
- [3] M. Y. Vardi. The complexity of relational query languages (extended abstract). In H. R. Lewis, B. B. Simons, W. A. Burkhard, and L. H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982.