

# First-Order Query Evaluation

Martin Raszyk

March 17, 2025

## 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  $\Rightarrow$  nat set where
  fv_fo_term_set (Var n) = {n}
| fv_fo_term_set _ = {}

definition fv_fo_terms_set :: ('a fo_term) list  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  nat list where
  fv_fo_terms_list ts = remdups_adj (sort (fv_fo_terms_list_rec ts))

fun eval_term :: 'a val  $\Rightarrow$  'a fo_term  $\Rightarrow$  'a (infix <·> 60) where
  eval_term  $\sigma$  (Const c) = c
| eval_term  $\sigma$  (Var n) =  $\sigma$  n

definition eval_terms :: 'a val  $\Rightarrow$  ('a fo_term) list  $\Rightarrow$  'a list (infix <⊙> 60) where
  eval_terms  $\sigma$  ts = map (eval_term  $\sigma$ ) ts

lemma finite_set_fo_term: finite (set_fo_term t)
by (cases t) auto

lemma list_fo_term_set: set (list_fo_term t) = set_fo_term t
by (cases t) auto

lemma finite_fv_fo_term_set: finite (fv_fo_term_set t)
by (cases t) auto

lemma fv_fo_term_setD:  $n \in$  fv_fo_term_set t  $\Longrightarrow$  t = Var n
by (cases t) auto

lemma fv_fo_term_set_list: set (fv_fo_term_list t) = fv_fo_term_set t
by (cases t) auto

lemma sorted_distinct_fv_fo_term_list: sorted_distinct (fv_fo_term_list t)
by (cases t) auto

lemma fv_fo_term_set_cong: fv_fo_term_set t = fv_fo_term_set (map_fo_term f t)
by (cases t) auto

lemma fv_fo_terms_setI: Var m  $\in$  set ts  $\Longrightarrow$  m  $\in$  fv_fo_terms_set ts
by (induction ts) (auto simp: fv_fo_terms_set_def)

lemma fv_fo_terms_setD: m  $\in$  fv_fo_terms_set ts  $\Longrightarrow$  Var m  $\in$  set ts
by (induction ts) (auto simp: fv_fo_terms_set_def dest: fv_fo_term_setD)

lemma finite_fv_fo_terms_set: finite (fv_fo_terms_set ts)
by (auto simp: fv_fo_terms_set_def finite_fv_fo_term_set)

lemma fv_fo_terms_set_list: set (fv_fo_terms_list ts) = fv_fo_terms_set ts
using fv_fo_term_set_list
unfolding fv_fo_terms_list_def
by (induction ts rule: fv_fo_terms_list_rec.induct)
  (auto simp: fv_fo_terms_set_def set_insort_key)

lemma distinct_remdups_adj_sort: sorted xs  $\Longrightarrow$  distinct (remdups_adj xs)

```

by (induction xs rule: induct\_list012) auto

**lemma** sorted\_distinct\_fv\_fo\_terms\_list: sorted\_distinct (fv\_fo\_terms\_list ts)  
**unfolding** fv\_fo\_terms\_list\_def  
**by** (induction ts rule: fv\_fo\_terms\_list\_rec.induct)  
(auto simp add: sorted\_insort intro: distinct\_remdups\_adj\_sort)

**lemma** fv\_fo\_terms\_set\_cong: fv\_fo\_terms\_set ts = fv\_fo\_terms\_set (map (map\_fo\_term f) ts)  
**using** fv\_fo\_term\_set\_cong  
**by** (induction ts) (fastforce simp: fv\_fo\_terms\_set\_def)+

**lemma** eval\_term\_cong: ( $\bigwedge n. n \in \text{fv\_fo\_term\_set } t \implies \sigma n = \sigma' n$ )  $\implies$   
eval\_term  $\sigma$  t = eval\_term  $\sigma'$  t  
**by** (cases t) auto

**lemma** eval\_terms\_fv\_fo\_terms\_set:  $\sigma \odot ts = \sigma' \odot ts \implies n \in \text{fv\_fo\_terms\_set } ts \implies \sigma n = \sigma' n$

**proof** (induction ts)  
**case** (Cons t ts)  
**then show** ?case  
**by** (cases t) (auto simp: eval\_terms\_def fv\_fo\_terms\_set\_def)  
**qed** (auto simp: eval\_terms\_def fv\_fo\_terms\_set\_def)

**lemma** eval\_terms\_cong: ( $\bigwedge n. n \in \text{fv\_fo\_terms\_set } ts \implies \sigma n = \sigma' n$ )  $\implies$   
eval\_terms  $\sigma$  ts = eval\_terms  $\sigma'$  ts  
**by** (auto simp: eval\_terms\_def fv\_fo\_terms\_set\_def intro: eval\_term\_cong)

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

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

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

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

```

lemma finite_fv_fo_fmla: finite (fv_fo_fmla  $\varphi$ )
  by (induction  $\varphi$  rule: fv_fo_fmla.induct)
    (auto simp: finite_fv_fo_term_set finite_fv_fo_terms_set)

lemma fv_fo_fmla_list_set: set (fv_fo_fmla_list  $\varphi$ ) = fv_fo_fmla  $\varphi$ 
  unfolding fv_fo_fmla_list_def
  by (induction  $\varphi$  rule: fv_fo_fmla.induct) (auto simp: fv_fo_terms_set_list fv_fo_term_set_list)

lemma sorted_distinct_fv_list: sorted_distinct (fv_fo_fmla_list  $\varphi$ )
  by (auto simp: fv_fo_fmla_list_def intro: distinct_remdups_adj_sort)

lemma length_fv_fo_fmla_list: length (fv_fo_fmla_list  $\varphi$ ) = card (fv_fo_fmla  $\varphi$ )
  using fv_fo_fmla_list_set[of  $\varphi$ ] sorted_distinct_fv_list[of  $\varphi$ ]
    distinct_card[of fv_fo_fmla_list  $\varphi$ ]
  by auto

lemma fv_fo_fmla_list_eq: fv_fo_fmla  $\varphi$  = fv_fo_fmla  $\psi$   $\implies$  fv_fo_fmla_list  $\varphi$  = fv_fo_fmla_list
 $\psi$ 
  using fv_fo_fmla_list_set sorted_distinct_fv_list
  by (metis sorted_distinct_set_unique)

lemma fv_fo_fmla_list_Conj: fv_fo_fmla_list (Conj  $\varphi$   $\psi$ ) = fv_fo_fmla_list (Conj  $\psi$   $\varphi$ )
  using fv_fo_fmla_list_eq[of Conj  $\varphi$   $\psi$  Conj  $\psi$   $\varphi$ ]
  by auto

type_synonym 'a table = ('a list) set

type_synonym ('t, 'b) fo_intp = 'b  $\times$  nat  $\implies$  't

fun wf_fo_intp :: ('a, 'b) fo_fmla  $\implies$  ('a table, 'b) fo_intp  $\implies$  bool where
  wf_fo_intp (Pred r ts) I  $\longleftrightarrow$  finite (I (r, length ts))
| wf_fo_intp (Bool b) I  $\longleftrightarrow$  True
| wf_fo_intp (Eqa t t') I  $\longleftrightarrow$  True
| wf_fo_intp (Neg  $\varphi$ ) I  $\longleftrightarrow$  wf_fo_intp  $\varphi$  I
| wf_fo_intp (Conj  $\varphi$   $\psi$ ) I  $\longleftrightarrow$  wf_fo_intp  $\varphi$  I  $\wedge$  wf_fo_intp  $\psi$  I
| wf_fo_intp (Disj  $\varphi$   $\psi$ ) I  $\longleftrightarrow$  wf_fo_intp  $\varphi$  I  $\wedge$  wf_fo_intp  $\psi$  I
| wf_fo_intp (Exists n  $\varphi$ ) I  $\longleftrightarrow$  wf_fo_intp  $\varphi$  I
| wf_fo_intp (Forall n  $\varphi$ ) I  $\longleftrightarrow$  wf_fo_intp  $\varphi$  I

fun sat :: ('a, 'b) fo_fmla  $\implies$  ('a table, 'b) fo_intp  $\implies$  'a val  $\implies$  bool where
  sat (Pred r ts) I  $\sigma$   $\longleftrightarrow$   $\sigma \odot ts \in I$  (r, length ts)
| sat (Bool b) I  $\sigma$   $\longleftrightarrow$  b
| sat (Eqa t t') I  $\sigma$   $\longleftrightarrow$   $\sigma \cdot t = \sigma \cdot t'$ 
| sat (Neg  $\varphi$ ) I  $\sigma$   $\longleftrightarrow$   $\neg$ sat  $\varphi$  I  $\sigma$ 
| sat (Conj  $\varphi$   $\psi$ ) I  $\sigma$   $\longleftrightarrow$  sat  $\varphi$  I  $\sigma$   $\wedge$  sat  $\psi$  I  $\sigma$ 
| sat (Disj  $\varphi$   $\psi$ ) I  $\sigma$   $\longleftrightarrow$  sat  $\varphi$  I  $\sigma$   $\vee$  sat  $\psi$  I  $\sigma$ 
| sat (Exists n  $\varphi$ ) I  $\sigma$   $\longleftrightarrow$  ( $\exists x$ . sat  $\varphi$  I ( $\sigma(n := x)$ ))
| sat (Forall n  $\varphi$ ) I  $\sigma$   $\longleftrightarrow$  ( $\forall x$ . sat  $\varphi$  I ( $\sigma(n := x)$ ))

lemma sat_fv_cong: ( $\bigwedge n$ .  $n \in$  fv_fo_fmla  $\varphi \implies \sigma n = \sigma' n$ )  $\implies$ 
  sat  $\varphi$  I  $\sigma$   $\longleftrightarrow$  sat  $\varphi$  I  $\sigma'$ 
proof (induction  $\varphi$  arbitrary:  $\sigma$   $\sigma'$ )
  case (Neg  $\varphi$ )
  show ?case
    using Neg(1)[of  $\sigma$   $\sigma'$ ] Neg(2)
    by auto
next

```

```

case (Conj  $\varphi$   $\psi$ )
show ?case
  using Conj(1,2)[of  $\sigma$   $\sigma'$ ] Conj(3)
  by auto
next
case (Disj  $\varphi$   $\psi$ )
show ?case
  using Disj(1,2)[of  $\sigma$   $\sigma'$ ] Disj(3)
  by auto
next
case (Exists  $n$   $\varphi$ )
have  $\bigwedge x. \text{sat } \varphi I (\sigma(n := x)) = \text{sat } \varphi I (\sigma'(n := x))$ 
  using Exists(2)
  by (auto intro!: Exists(1))
then show ?case
  by simp
next
case (Forall  $n$   $\varphi$ )
have  $\bigwedge x. \text{sat } \varphi I (\sigma(n := x)) = \text{sat } \varphi I (\sigma'(n := x))$ 
  using Forall(2)
  by (auto intro!: Forall(1))
then show ?case
  by simp
qed (auto cong: eval_terms_cong eval_term_cong)

definition proj_sat :: ('a, 'b) fo_fmla  $\Rightarrow$  ('a table, 'b) fo_intp  $\Rightarrow$  'a table where
  proj_sat  $\varphi I = (\lambda\sigma. \text{map } \sigma (\text{fv\_fo\_fm\_la\_list } \varphi)) \text{ ' } \{\sigma. \text{sat } \varphi I \sigma\}$ 

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_fmla  $\Rightarrow$  ('b  $\times$  nat  $\Rightarrow$  'a list set)  $\Rightarrow$  't  $\Rightarrow$  bool
  and abs :: ('a fo_term) list  $\Rightarrow$  'a table  $\Rightarrow$  't
  and rep :: 't  $\Rightarrow$  'a table
  and res :: 't  $\Rightarrow$  'a eval_res
  and eval_bool :: bool  $\Rightarrow$  't
  and eval_eq :: 'a fo_term  $\Rightarrow$  'a fo_term  $\Rightarrow$  't
  and eval_neg :: nat list  $\Rightarrow$  't  $\Rightarrow$  't
  and eval_conj :: nat list  $\Rightarrow$  't  $\Rightarrow$  nat list  $\Rightarrow$  't  $\Rightarrow$  't
  and eval_ajoin :: nat list  $\Rightarrow$  't  $\Rightarrow$  nat list  $\Rightarrow$  't  $\Rightarrow$  't
  and eval_disj :: nat list  $\Rightarrow$  't  $\Rightarrow$  nat list  $\Rightarrow$  't  $\Rightarrow$  't
  and eval_exists :: nat  $\Rightarrow$  nat list  $\Rightarrow$  't  $\Rightarrow$  't
  and eval_forall :: nat  $\Rightarrow$  nat list  $\Rightarrow$  't  $\Rightarrow$  't
  assumes fo_rep: wf  $\varphi I t \Longrightarrow \text{rep } t = \text{proj\_sat } \varphi I$ 
  and fo_res_fin: wf  $\varphi I t \Longrightarrow \text{finite } (\text{rep } t) \Longrightarrow \text{res } t = \text{Fin } (\text{rep } t)$ 
  and fo_res_infin: wf  $\varphi I t \Longrightarrow \neg \text{finite } (\text{rep } t) \Longrightarrow \text{res } t = \text{Infin}$ 
  and fo_abs: finite (I (r, length ts))  $\Longrightarrow \text{wf } (\text{Pred } r \text{ } ts) I (\text{abs } ts (I (\text{r}, \text{length } ts)))$ 
  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  $\varphi I t \Longrightarrow \text{wf } (\text{Neg } \varphi) I (\text{eval\_neg } (\text{fv\_fo\_fm\_la\_list } \varphi) t)$ 
  and fo_conj: wf  $\varphi I t\varphi \Longrightarrow \text{wf } \psi I t\psi \Longrightarrow (\text{case } \psi \text{ of Neg } \psi' \Rightarrow \text{False} \mid \_ \Rightarrow \text{True}) \Longrightarrow$ 
    wf (Conj  $\varphi$   $\psi$ ) I (eval_conj (fv\_fo\_fm\_la\_list  $\varphi$ ) t $\varphi$  (fv\_fo\_fm\_la\_list  $\psi$ ) t $\psi$ )
  and fo_ajoin: wf  $\varphi I t\varphi \Longrightarrow \text{wf } \psi' I t\psi' \Longrightarrow$ 

```

```

    wf (Conj  $\varphi$  (Neg  $\psi'$ )) I (eval_ajoin (fv_fo_fmula_list  $\varphi$ ) t $\varphi$  (fv_fo_fmula_list  $\psi'$ ) t $\psi'$ )
and fo_disj: wf  $\varphi$  I t $\varphi$   $\implies$  wf  $\psi$  I t $\psi$   $\implies$ 
    wf (Disj  $\varphi$   $\psi$ ) I (eval_disj (fv_fo_fmula_list  $\varphi$ ) t $\varphi$  (fv_fo_fmula_list  $\psi$ ) t $\psi$ )
and fo_exists: wf  $\varphi$  I t  $\implies$  wf (Exists i  $\varphi$ ) I (eval_exists i (fv_fo_fmula_list  $\varphi$ ) t)
and fo_forall: wf  $\varphi$  I t  $\implies$  wf (Forall i  $\varphi$ ) I (eval_forall i (fv_fo_fmula_list  $\varphi$ ) t)
begin

fun eval_fmula :: ('a, 'b) fo_fmula  $\Rightarrow$  ('a table, 'b) fo_intp  $\Rightarrow$  '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  $\varphi$ ) I = eval_neg (fv_fo_fmula_list  $\varphi$ ) (eval_fmula  $\varphi$  I)
| eval_fmula (Conj  $\varphi$   $\psi$ ) I = (let ns $\varphi$  = fv_fo_fmula_list  $\varphi$ ; ns $\psi$  = fv_fo_fmula_list  $\psi$ ;
  X $\varphi$  = eval_fmula  $\varphi$  I in
  case  $\psi$  of Neg  $\psi'$   $\Rightarrow$  let X $\psi'$  = eval_fmula  $\psi'$  I in
    eval_ajoin ns $\varphi$  X $\varphi$  (fv_fo_fmula_list  $\psi'$ ) X $\psi'$ 
  | _  $\Rightarrow$  eval_conj ns $\varphi$  X $\varphi$  ns $\psi$  (eval_fmula  $\psi$  I))
| eval_fmula (Disj  $\varphi$   $\psi$ ) I = eval_disj (fv_fo_fmula_list  $\varphi$ ) (eval_fmula  $\varphi$  I)
  (fv_fo_fmula_list  $\psi$ ) (eval_fmula  $\psi$  I)
| eval_fmula (Exists i  $\varphi$ ) I = eval_exists i (fv_fo_fmula_list  $\varphi$ ) (eval_fmula  $\varphi$  I)
| eval_fmula (Forall i  $\varphi$ ) I = eval_forall i (fv_fo_fmula_list  $\varphi$ ) (eval_fmula  $\varphi$  I)

lemma eval_fmula_correct:
  fixes  $\varphi$  :: ('a :: infinite, 'b) fo_fmula
  assumes wf_fo_intp  $\varphi$  I
  shows wf  $\varphi$  I (eval_fmula  $\varphi$  I)
  using assms
proof (induction  $\varphi$  I rule: eval_fmula.induct)
  case (1 r ts I)
  then show ?case
    using fo_abs
    by auto
next
  case (2 b I)
  then show ?case
    using fo_bool
    by auto
next
  case (3 t t' I)
  then show ?case
    using fo_eq
    by auto
next
  case (4  $\varphi$  I)
  then show ?case
    using fo_neg
    by auto
next
  case (5  $\varphi$   $\psi$  I)
  have fins: wf_fo_intp  $\varphi$  I wf_fo_intp  $\psi$  I
    using 5(10)
    by auto
  have eval $\varphi$ : wf  $\varphi$  I (eval_fmula  $\varphi$  I)
    using 5(1)[OF _ _ fins(1)]
    by auto
  show ?case
proof (cases  $\exists \psi'. \psi = \text{Neg } \psi'$ )
  case True

```

```

then obtain  $\psi'$  where  $\psi\_def: \psi = Neg \psi'$ 
  by auto
have  $fin: wf\_fo\_intp \psi' I$ 
  using  $fins(2)$ 
  by (auto simp:  $\psi\_def$ )
have  $eval\psi': wf \psi' I (eval\_fmla \psi' I)$ 
  using  $5(5)[OF \_ \_ \_ \psi\_def fin]$ 
  by auto
show ?thesis
  unfolding  $\psi\_def$ 
  using  $fo\_ajoin[OF eval\var eval\psi']$ 
  by auto
next
case False
then have  $eval\psi: wf \psi I (eval\_fmla \psi I)$ 
  using  $5 fins(2)$ 
  by (cases  $\psi$ ) auto
have  $eval: eval\_fmla (Conj \var \psi) I = eval\_conj (fv\_fo\_fmla\_list \var) (eval\_fmla \var I)$ 
  ( $fv\_fo\_fmla\_list \psi$ ) ( $eval\_fmla \psi I$ )
  using False
  by (auto simp: Let\_def split: fo\_fmla.splits)
show  $wf (Conj \var \psi) I (eval\_fmla (Conj \var \psi) I)$ 
  using  $fo\_conj[OF eval\var eval\psi, folded eval] False$ 
  by (auto split: fo\_fmla.splits)
qed
next
case ( $6 \var \psi I$ )
then show ?case
  using  $fo\_disj$ 
  by auto
next
case ( $7 i \var I$ )
then show ?case
  using  $fo\_exists$ 
  by auto
next
case ( $8 i \var I$ )
then show ?case
  using  $fo\_forall$ 
  by auto
qed

```

**definition**  $eval :: ('a, 'b) fo\_fmla \Rightarrow ('a \text{ table}, 'b) fo\_intp \Rightarrow 'a \text{ eval\_res}$  **where**  
 $eval \var I = (if wf\_fo\_intp \var I then res (eval\_fmla \var I) else Wf\_error)$

**lemma**  $eval\_fmla\_proj\_sat$ :  
**fixes**  $\var :: ('a :: infinite, 'b) fo\_fmla$   
**assumes**  $wf\_fo\_intp \var I$   
**shows**  $rep (eval\_fmla \var I) = proj\_sat \var I$   
**using**  $eval\_fmla\_correct[OF assms]$   
**by** (*auto simp: fo\\_rep*)

**lemma**  $eval\_sound$ :  
**fixes**  $\var :: ('a :: infinite, 'b) fo\_fmla$   
**assumes**  $eval \var I = Fin Z$   
**shows**  $Z = proj\_sat \var I$

**proof** –  
**have**  $wf \var I (eval\_fmla \var I)$

```

    using eval_fmula_correct assms
  by (auto simp: eval_def split: if_splits)
then show ?thesis
  using assms fo_res_fin fo_res_infin
  by (fastforce simp: eval_def fo_rep split: if_splits)
qed

```

```

lemma eval_complete:
  fixes  $\varphi :: ('a :: infinite, 'b) fo_fmula$ 
  assumes eval  $\varphi I = \text{Infin}$ 
  shows infinite (proj_sat  $\varphi I$ )
proof -
  have wf  $\varphi I$  (eval_fmula  $\varphi I$ )
  using eval_fmula_correct assms
  by (auto simp: eval_def split: if_splits)
  then show ?thesis
  using assms fo_res_fin
  by (auto simp: eval_def fo_rep split: if_splits)
qed

```

end

end

theory Mapping\_Code

imports Containers.Mapping\_Impl

begin

```

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

```

```

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")
    ( $\lambda \_.$  set_of_idx (RBT_Mapping t))
    | Some  $\_ \Rightarrow \bigcup (\text{snd } \text{' set (RBT_Mapping2.entries t)})$ )
  unfolding RBT_Mapping_def
  by transfer (auto simp: ran_def rbt_comp_lookup[OF ID_ccompare] ord.is_rbt_def linorder.rbt_lookup_in_tree[OF comparator.linorder[OF ID_ccompare]] split: option.splits)+

```

```

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")
    ( $\lambda \_.$  Mapping.combine f (RBT_Mapping t) (RBT_Mapping u))
    | Some  $\_ \Rightarrow$  RBT_Mapping (RBT_Mapping2.join ( $\lambda \_.$  f) t u))
  by (auto simp add: Mapping.combine.abs_eq Mapping_inject lookup_join split: option.split)

```

```

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

```

```

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")
    ( $\lambda \_.$  mapping_join f (RBT_Mapping t) (RBT_Mapping u))

```



```

| Some _ => RBT_Mapping (RBT_Mapping2.meet (λ_. f) t u))
by (auto simp add: mapping_join.abs_eq Mapping_inject lookup_meet split: option.split)

context fixes dummy :: 'a :: ccompare begin

lift_definition diff ::
  ('a, 'b) mapping_rbt => ('a, 'b) mapping_rbt => ('a, 'b) mapping_rbt is rbt_comp_minus ccomp
by (auto 4 3 intro: linorder.rbt_minus_is_rbt ID_ccompare ord.is_rbt_rbt_sorted simp: rbt_comp_minus[OF
ID_ccompare'])

end

context assumes ID_ccompare_neq_None: ID CCOMPARE('a :: ccompare) ≠ None
begin

lemma lookup_diff:
  RBT_Mapping2.lookup (diff (t1 :: ('a, 'b) mapping_rbt) t2) =
  (λk. case RBT_Mapping2.lookup t1 k of None => None | Some v1 => (case RBT_Mapping2.lookup t2
k of None => Some v1 | Some v2 => None))
by transfer (auto simp add: fun_eq_iff linorder.rbt_lookup_rbt_minus[OF mapping_linorder] ID_ccompare_neq_None
restrict_map_def split: option.splits)

end

lift_definition mapping_antijoin :: ('a, 'b) mapping => ('a, 'b) mapping => ('a, 'b) mapping is
λm m' x. case m x of None => None | Some y => (case m' x of None => Some y | Some y' => None) .

lemma mapping_antijoin_code[code]:
  fixes t :: ('a :: ccompare, 'b) mapping_rbt
  shows mapping_antijoin (RBT_Mapping t) (RBT_Mapping u) =
  (case ID CCOMPARE('a) of None => Code.abort (STR "mapping_antijoin RBT_Mapping: ccompare
= None") (λ_. mapping_antijoin (RBT_Mapping t) (RBT_Mapping u))
| Some _ => RBT_Mapping (diff t u))
by (auto simp add: mapping_antijoin.abs_eq Mapping_inject lookup_diff split: option.split)

end

theory Cluster
  imports Mapping_Code
begin

lemma these_Un[simp]: Option.these (A ∪ B) = Option.these A ∪ Option.these B
by (auto simp: Option.these_def)

lemma these_insert[simp]: Option.these (insert x A) = (case x of Some a => insert a | None => id)
(Option.these A)
by (auto simp: Option.these_def split: option.splits) force

lemma these_image_Un[simp]: Option.these (f ` (A ∪ B)) = Option.these (f ` A) ∪ Option.these (f ` B)
by (auto simp: Option.these_def)

lemma these_imageI: f x = Some y ==> x ∈ X ==> y ∈ Option.these (f ` X)
by (force simp: Option.these_def)

lift_definition cluster :: ('b => 'a option) => 'b set => ('a, 'b set) mapping is
λf Y x. if Some x ∈ f ` Y then Some {y ∈ Y. f y = Some x} else None .

lemma set_of_idx_cluster: set_of_idx (cluster (Some ∘ f) X) = X
by transfer (auto simp: ran_def)

```

**lemma** *lookup\_cluster'*: *Mapping.lookup (cluster (Some o h) X) y = (if y ∈ h ' X then None else Some {x ∈ X. h x = y})*  
**by** *transfer auto*

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

**definition** *add\_to\_rbt* :: *'a × 'b ⇒ ('a, 'b set) rbt ⇒ ('a, 'b set) rbt* **where**  
*add\_to\_rbt = (λ(a, b) t. case rbt\_lookup t a of Some X ⇒ rbt\_insert a (insert b X) t | None ⇒ rbt\_insert a {b} t)*

**abbreviation** *add\_option\_to\_rbt* *f* ≡ *(λb \_ t. case f b of Some a ⇒ add\_to\_rbt (a, b) t | None ⇒ t)*

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

**end**

**context** *linorder*  
**begin**

**lemma** *is\_rbt\_add\_to\_rbt*: *is\_rbt t ⇒ is\_rbt (add\_to\_rbt ab t)*  
**by** *(auto simp: add\_to\_rbt\_def split: prod.splits option.splits)*

**lemma** *is\_rbt\_fold\_add\_to\_rbt*: *is\_rbt t' ⇒ is\_rbt (RBT\_Impl.fold (add\_option\_to\_rbt f) t t')*  
**by** *(induction t arbitrary: t') (auto 0 0 simp: is\_rbt\_add\_to\_rbt split: option.splits)*

**lemma** *is\_rbt\_cluster\_rbt*: *is\_rbt (cluster\_rbt f t)*  
**using** *is\_rbt\_fold\_add\_to\_rbt Empty\_is\_rbt*  
**by** *(fastforce simp: cluster\_rbt\_def)*

**lemma** *rbt\_insert\_entries\_None*: *is\_rbt t ⇒ rbt\_lookup t k = None ⇒ set (RBT\_Impl.entries (rbt\_insert k v t)) = insert (k, v) (set (RBT\_Impl.entries t))*  
**by** *(auto simp: rbt\_lookup\_in\_tree[symmetric] rbt\_lookup\_rbt\_insert split: if\_splits)*

**lemma** *rbt\_insert\_entries\_Some*: *is\_rbt t ⇒ rbt\_lookup t k = Some v' ⇒ set (RBT\_Impl.entries (rbt\_insert k v t)) = insert (k, v) (set (RBT\_Impl.entries t) - {(k, v')})*  
**by** *(auto simp: rbt\_lookup\_in\_tree[symmetric] rbt\_lookup\_rbt\_insert split: if\_splits)*

**lemma** *keys\_add\_to\_rbt*: *is\_rbt t ⇒ set (RBT\_Impl.keys (add\_to\_rbt (a, b) t)) = insert a (set (RBT\_Impl.keys t))*  
**by** *(auto simp: add\_to\_rbt\_def RBT\_Impl.keys\_def rbt\_insert\_entries\_None rbt\_insert\_entries\_Some split: option.splits)*

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

*Option.these (f ' set (RBT\_Impl.keys t)) ∪ set (RBT\_Impl.keys t')*

**proof** *(induction t arbitrary: t')*

**case** *(Branch col t1 k v t2)*

**have** *valid: is\_rbt (RBT\_Impl.fold (add\_option\_to\_rbt f) t1 t')*

**using** *Branch(3)*

**by** *(auto intro: is\_rbt\_fold\_add\_to\_rbt)*

**show** *?case*

**proof** *(cases f k)*

**case** *None*

**show** *?thesis*

```

    by (auto simp: None Branch(2)[OF valid] Branch(1)[OF Branch(3)])
next
case (Some a)
have valid': is_rbt (add_to_rbt (a, k) (RBT_Impl.fold (add_option_to_rbt f) t1 t'))
  by (auto intro: is_rbt_add_to_rbt[OF valid])
show ?thesis
  by (auto simp: Some Branch(2)[OF valid'] keys_add_to_rbt[OF valid] Branch(1)[OF Branch(3)])
qed
qed auto

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  $\implies$  {b} | Some Y  $\implies$  insert b Y) else rbt_lookup t x)
  by (auto simp: add_to_rbt_def rbt_lookup_rbt_insert split: option.splits)

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  $\implies$  {} | Some Y  $\implies$  Y)) else None)
proof (induction t arbitrary: t')
case Empty
then show ?case
  using rbt_lookup_iff_keys(2,3)[OF is_rbt_rbt_sorted]
  by (fastforce split: option.splits)
next
case (Branch col t1 k v t2)
have valid: is_rbt (RBT_Impl.fold (add_option_to_rbt f) t1 t')
  using Branch(3)
  by (auto intro: is_rbt_fold_add_to_rbt)
show ?case
proof (cases f k)
case None
have fold_set: x  $\in$  Option.these (f ' set (RBT_Impl.keys t2))  $\cup$  ((Option.these (f ' set (RBT_Impl.keys
t1))  $\cup$  set (RBT_Impl.keys t'))  $\longleftrightarrow$ 
  x  $\in$  Option.these (f ' set (RBT_Impl.keys (Branch col t1 k v t2)))  $\cup$  set (RBT_Impl.keys t')
  by (auto simp: None)
show ?thesis
  unfolding fold_simps comp_def None option.case(1) Branch(2)[OF valid] keys_add_to_rbt[OF
valid] keys_fold_add_to_rbt[OF Branch(3)]
  rbt_lookup_add_to_rbt[OF valid] Branch(1)[OF Branch(3)] fold_set
  using rbt_lookup_iff_keys(2,3)[OF is_rbt_rbt_sorted[OF Branch(3)]]
  by (auto simp: None split: option.splits) (auto dest: these_imageI)
next
case (Some a)
have valid': is_rbt (add_to_rbt (a, k) (RBT_Impl.fold (add_option_to_rbt f) t1 t'))
  by (auto intro: is_rbt_add_to_rbt[OF valid])
have fold_set: x  $\in$  Option.these (f ' set (RBT_Impl.keys t2))  $\cup$  (insert a (Option.these (f ' set
(RBT_Impl.keys t1))  $\cup$  set (RBT_Impl.keys t')))  $\longleftrightarrow$ 
  x  $\in$  Option.these (f ' set (RBT_Impl.keys (Branch col t1 k v t2)))  $\cup$  set (RBT_Impl.keys t')
  by (auto simp: Some)
have F1: (case if P then Some X else None of None  $\implies$  {k} | Some Y  $\implies$  insert k Y) =
  (if P then (insert k X) else {k}) for P X
  by auto
have F2: (case if a = x then Some X else if P then Some Y else None of None  $\implies$  {} | Some Y  $\implies$ 
Y) =
  (if a = x then X else if P then Y else {})
  for P X and Y :: 'b set
  by auto

```

```

show ?thesis
  unfolding fold_simps comp_def Some option.case(2) Branch(2)[OF valid] keys_add_to_rbt[OF
valid] keys_fold_add_to_rbt[OF Branch(3)]
  rbt_lookup_add_to_rbt[OF valid] Branch(1)[OF Branch(3)] fold_set F1 F2
  using rbt_lookup_iff_keys(2,3)[OF is_rbt_rbt_sorted[OF Branch(3)]]
  by (auto simp: Some split: option.splits) (auto dest: these_imageI)
qed
qed

end

context
  fixes c :: 'a comparator
begin

definition add_to_rbt_comp :: 'a × 'b ⇒ ('a, 'b set) rbt ⇒ ('a, 'b set) rbt where
  add_to_rbt_comp = (λ(a, b) t. case rbt_comp_lookup c t a of None ⇒ rbt_comp_insert c a {b} t
  | Some X ⇒ rbt_comp_insert c a (insert b X) t)

abbreviation add_option_to_rbt_comp f ≡ (λb _ t. case f b of Some a ⇒ add_to_rbt_comp (a, b) t
  | None ⇒ t)

definition cluster_rbt_comp :: ('b ⇒ 'a option) ⇒ ('b, unit) rbt ⇒ ('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)
  unfolding add_to_rbt_comp_def ord.add_to_rbt_def rbt_comp_lookup[OF c] rbt_comp_insert[OF
c]
  by simp

lemma cluster_rbt_comp: cluster_rbt_comp = ord.cluster_rbt (lt_of_comp c)
  unfolding cluster_rbt_comp_def ord.cluster_rbt_def add_to_rbt_comp
  by simp

end

end

lift_definition mapping_of_cluster :: ('b ⇒ 'a :: ccompare option) ⇒ ('b, unit) rbt ⇒ ('a, 'b set)
mapping_rbt is
  cluster_rbt_comp ccomp
  using linorder.is_rbt_fold_add_to_rbt[OF comparator.linorder[OF ID_ccompare] ord.Empty_is_rbt]
  by (fastforce simp: cluster_rbt_comp[OF ID_ccompare] ord.cluster_rbt_def)

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 -
  {
  fix c c'

```

```

    assume assms: ID ccompare = (Some c :: 'a comparator option) ID ccompare = (Some c' :: 'b
comparator option)
  have c_def: c = ccomp
    using assms(1)
    by auto
  have c'_def: c' = ccomp
    using assms(2)
    by auto
  have c: comparator (ccomp :: 'a comparator)
    using ID_ccountare'[OF assms(1)]
    by (auto simp: c_def)
  have c': comparator (ccomp :: 'b comparator)
    using ID_ccountare'[OF assms(2)]
    by (auto simp: c'_def)
  note c_class = comparator.linorder[OF c]
  note c'_class = comparator.linorder[OF c']
  have rbt_lookup_cluster: ord.rbt_lookup cless (cluster_rbt_comp ccomp f t) =
    (λx. if x ∈ Option.these (f ' (set (RBT_Impl.keys t))) then Some {y ∈ (set (RBT_Impl.keys t)). f
y = Some x} else None)
  if ord.is_rbt cless (t :: ('b, unit) rbt) ∨ ID ccompare = (None :: 'b comparator option) for t
  proof -
    have is_rbt_t: ord.is_rbt cless t
      using assms that
      by auto
    show ?thesis
      unfolding cluster_rbt_comp[OF c] ord.cluster_rbt_def linorder.rbt_lookup_fold_add_to_rbt[OF
c_class ord.Empty_is_rbt]
      by (auto simp: ord.rbt_lookup.simps split: option.splits)
    qed
    have dom_ord_rbt_lookup: ord.is_rbt cless t ⇒ dom (ord.rbt_lookup cless t) = set (RBT_Impl.keys
t) for t :: ('b, unit) rbt
      using linorder.rbt_lookup_keys[OF c'_class] ord.is_rbt_def
      by auto
    have cluster_f (Collect (RBT_Set2.member t)) = Mapping (RBT_Mapping2.lookup (mapping_of_cluster
f (mapping_rbt.impl_of t)))
      using assms(2)[unfolded c'_def]
      by (transfer fixing: f) (auto simp: in_these_eq rbt_comp_lookup[OF c] rbt_comp_lookup[OF c']
rbt_lookup_cluster dom_ord_rbt_lookup)
    }
    then show ?thesis
      unfolding RBT_set_def
      by (auto split: option.splits)
  qed

end
theory Ailamazyan
  imports Eval_FO Cluster Mapping_Code
begin

fun SP :: ('a, 'b) fo_fmula ⇒ nat set where
  SP (Eq a (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*:  $SP \varphi \subseteq fv\_fo\_fmla \varphi$   
**by** (*induction*  $\varphi$  *rule*: *SP.induct*) *auto*

**lemma** *finite\_SP*: *finite* (*SP*  $\varphi$ )  
**using** *SP\_fv finite\_fv\_fo\_fmla finite\_subset* **by** *fastforce*

**fun** *SP\_list\_rec* :: ('a, 'b) *fo\_fmla*  $\Rightarrow$  *nat list* **where**  
*SP\_list\_rec* (*Eqa* (*Var*  $n$ ) (*Var*  $n'$ )) = (*if*  $n \neq n'$  *then* [ $n, n'$ ] *else* [])  
| *SP\_list\_rec* (*Neg*  $\varphi$ ) = *SP\_list\_rec*  $\varphi$   
| *SP\_list\_rec* (*Conj*  $\varphi \psi$ ) = *SP\_list\_rec*  $\varphi$  @ *SP\_list\_rec*  $\psi$   
| *SP\_list\_rec* (*Disj*  $\varphi \psi$ ) = *SP\_list\_rec*  $\varphi$  @ *SP\_list\_rec*  $\psi$   
| *SP\_list\_rec* (*Exists*  $n \varphi$ ) = *filter* ( $\lambda m. n \neq m$ ) (*SP\_list\_rec*  $\varphi$ )  
| *SP\_list\_rec* (*Forall*  $n \varphi$ ) = *filter* ( $\lambda m. n \neq m$ ) (*SP\_list\_rec*  $\varphi$ )  
| *SP\_list\_rec* \_ = []

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

**lemma** *SP\_list\_set*: *set* (*SP\_list*  $\varphi$ ) = *SP*  $\varphi$   
**unfolding** *SP\_list\_def*  
**by** (*induction*  $\varphi$  *rule*: *SP.induct*) (*auto simp*: *fv\_fo\_terms\_set\_list*)

**lemma** *sorted\_distinct\_SP\_list*: *sorted\_distinct* (*SP\_list*  $\varphi$ )  
**unfolding** *SP\_list\_def*  
**by** (*auto intro*: *distinct\_remdups\_adj\_sort*)

**fun** *d* :: ('a, 'b) *fo\_fmla*  $\Rightarrow$  *nat* **where**  
*d* (*Eqa* (*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*:  $1 \leq d \varphi$   
**by** (*induction*  $\varphi$  *rule*: *d.induct*) *auto*

**lemma** *card\_SP\_d*: *card* (*SP*  $\varphi$ )  $\leq d \varphi$   
**using** *dual\_order.trans*  
**by** (*induction*  $\varphi$  *rule*: *SP.induct*) (*fastforce simp*: *card\_Diff1\_le finite\_SP*)+

**fun** *eval\_eterm* :: ('a + 'c) *val*  $\Rightarrow$  'a *fo\_term*  $\Rightarrow$  'a + 'c (**infix**  $\langle \cdot e \rangle$  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**  $\langle \odot e \rangle$  60) **where**  
*eval\_eterms*  $\sigma$  *ts* = *map* (*eval\_eterm*  $\sigma$ ) *ts*

**lemma** *eval\_eterm\_cong*: ( $\bigwedge n. n \in fv\_fo\_term\_set t \implies \sigma n = \sigma' n$ )  $\implies$   
*eval\_eterm*  $\sigma$  *t* = *eval\_eterm*  $\sigma'$  *t*  
**by** (*cases* *t*) *auto*

**lemma** *eval\_eterms\_fv\_fo\_terms\_set*:  $\sigma \odot e$  *ts* =  $\sigma' \odot e$  *ts*  $\implies n \in fv\_fo\_terms\_set$  *ts*  $\implies \sigma n = \sigma' n$   
**proof** (*induction* *ts*)  
**case** (*Cons* *t* *ts*)  
**then show** ?*case*

by (cases t) (auto simp: eval\_eterms\_def fv\_fo\_terms\_set\_def)  
**qed** (auto simp: eval\_eterms\_def fv\_fo\_terms\_set\_def)

**lemma** eval\_eterms\_cong:  $(\bigwedge n. n \in \text{fv\_fo\_terms\_set } ts \implies \sigma n = \sigma' n) \implies$   
 $\text{eval\_eterms } \sigma \text{ } ts = \text{eval\_eterms } \sigma' \text{ } ts$   
 by (auto simp: eval\_eterms\_def fv\_fo\_terms\_set\_def intro: eval\_eterm\_cong)

**lemma** eval\_terms\_eterms:  $\text{map } \text{Inl } (\sigma \odot ts) = (\text{Inl} \circ \sigma) \odot_e ts$

**proof** (induction ts)

case (Cons t ts)

then show ?case

by (cases t) (auto simp: eval\_terms\_def eval\_eterms\_def)

**qed** (auto simp: eval\_terms\_def eval\_eterms\_def)

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

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

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

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

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

**lemma** ad\_equiv\_pair\_refl[simp]:  $\text{ad\_equiv\_pair } X (a, a)$   
 by auto

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

**lemma** ad\_equiv\_pair\_comm:  $\text{ad\_equiv\_pair } X (a, a') \longleftrightarrow \text{ad\_equiv\_pair } X (a', a)$   
 by (auto simp: ad\_equiv\_pair.simps)

**lemma** ad\_equiv\_pair\_mono:  $X \subseteq Y \implies \text{ad\_equiv\_pair } Y (a, a') \implies \text{ad\_equiv\_pair } X (a, a')$   
**unfolding** ad\_equiv\_pair.simps  
 by fastforce

**lemma** sp\_equiv\_pair\_comm:  $\text{sp\_equiv\_pair } x \text{ } y \longleftrightarrow \text{sp\_equiv\_pair } y \text{ } x$   
 by (cases x; cases y) auto

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

**lemma** sp\_equiv\_mono:  $I \subseteq J \implies \text{sp\_equiv } \sigma \text{ } \tau \text{ } J \implies \text{sp\_equiv } \sigma \text{ } \tau \text{ } I$   
 by (auto simp: sp\_equiv\_def pairwise\_def)

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

**lemma** adAgrSets\_comm:  $\text{adAgrSets } FV \text{ } S \text{ } X \text{ } \sigma \text{ } \tau \implies \text{adAgrSets } FV \text{ } S \text{ } X \text{ } \tau \text{ } \sigma$   
**unfolding** adAgrSets\_def sp\_equiv\_def pairwise\_def  
 by (subst ad\_equiv\_pair\_comm) auto

**lemma** *ad\_agr\_sets\_mono*:  $X \subseteq Y \implies \text{ad\_agr\_sets } FV S Y \sigma \tau \implies \text{ad\_agr\_sets } FV S X \sigma \tau$   
**using** *ad\_equiv\_pair\_mono*  
**by** (*fastforce simp: ad\_agr\_sets\_def*)

**lemma** *ad\_agr\_sets\_mono'*:  $S \subseteq S' \implies \text{ad\_agr\_sets } FV S' X \sigma \tau \implies \text{ad\_agr\_sets } FV S X \sigma \tau$   
**by** (*auto simp: ad\_agr\_sets\_def sp\_equiv\_def pairwise\_def*)

**lemma** *ad\_equiv\_list\_comm*:  $\text{ad\_equiv\_list } X xs ys \implies \text{ad\_equiv\_list } X ys xs$   
**by** (*auto simp: ad\_equiv\_list\_def*) (*smt (verit, del\_insts) ad\_equiv\_pair\_comm in\_set\_zip prod.sel(1) prod.sel(2)*)

**lemma** *ad\_equiv\_list\_mono*:  $X \subseteq Y \implies \text{ad\_equiv\_list } Y xs ys \implies \text{ad\_equiv\_list } X xs ys$   
**using** *ad\_equiv\_pair\_mono*  
**by** (*fastforce simp: ad\_equiv\_list\_def*)

**lemma** *ad\_equiv\_list\_trans*:  
**assumes**  $\text{ad\_equiv\_list } X xs ys \text{ ad\_equiv\_list } X ys zs$   
**shows**  $\text{ad\_equiv\_list } X xs zs$   
**proof** –  
**have**  $\text{length } xs = \text{length } ys \text{ length } xs = \text{length } zs \text{ length } ys = \text{length } zs$   
**using** *assms*  
**by** (*auto simp: ad\_equiv\_list\_def*)  
**have**  $\bigwedge x z. (x, z) \in \text{set } (\text{zip } xs zs) \implies \text{ad\_equiv\_pair } X (x, z)$   
**proof** –  
**fix**  $x z$   
**assume**  $(x, z) \in \text{set } (\text{zip } xs zs)$   
**then obtain**  $i$  **where**  $i\_def: i < \text{length } xs \text{ xs } ! i = x \text{ zs } ! i = z$   
**by** (*auto simp: set\_zip*)  
**define**  $y$  **where**  $y = ys ! i$   
**have**  $\text{ad\_equiv\_pair } X (x, y) \text{ ad\_equiv\_pair } X (y, z)$   
**using** *assms lens i\_def*  
**by** (*fastforce simp: set\_zip y\_def ad\_equiv\_list\_def*)  
**then show**  $\text{ad\_equiv\_pair } X (x, z)$   
**unfolding**  $\text{ad\_equiv\_pair.simps}$   
**by** *blast*  
**qed**  
**then show** *?thesis*  
**using** *assms*  
**by** (*auto simp: ad\_equiv\_list\_def*)  
**qed**

**lemma** *ad\_equiv\_list\_link*:  $(\forall i \in \text{set } ns. \text{ad\_equiv\_pair } X (\sigma i, \tau i)) \longleftrightarrow \text{ad\_equiv\_list } X (\text{map } \sigma ns) (\text{map } \tau ns)$   
**by** (*auto simp: ad\_equiv\_list\_def set\_zip*) (*metis in\_set\_conv\_nth nth\_map*)

**lemma** *set\_zip\_comm*:  $(x, y) \in \text{set } (\text{zip } xs ys) \implies (y, x) \in \text{set } (\text{zip } ys xs)$   
**by** (*metis in\_set\_zip prod.sel(1) prod.sel(2)*)

**lemma** *set\_zip\_map*:  $\text{set } (\text{zip } (\text{map } \sigma ns) (\text{map } \tau ns)) = (\lambda n. (\sigma n, \tau n)) \text{ ' set } ns$   
**by** (*induction ns*) *auto*

**lemma** *sp\_equiv\_list\_comm*:  $\text{sp\_equiv\_list } xs ys \implies \text{sp\_equiv\_list } ys xs$   
**unfolding** *sp\_equiv\_list\_def*  
**using** *set\_zip\_comm*  
**by** (*auto simp: pairwise\_def*) *force* $+$

**lemma** *sp\_equiv\_list\_trans*:  
**assumes**  $\text{sp\_equiv\_list } xs ys \text{ sp\_equiv\_list } ys zs$



```

    shows sp_equiv_list xs zs
  proof -
    have lens: length xs = length ys length xs = length zs length ys = length zs
      using assms
      by (auto simp: sp_equiv_list_def)
    have pairwise sp_equiv_pair (set (zip xs zs))
    proof (rule pairwiseI)
      fix xz xz'
      assume xz ∈ set (zip xs zs) xz' ∈ set (zip xs zs)
      then obtain x z i x' z' i' where xz_def: i < length xs xs ! i = x zs ! i = z
        xz = (x, z) i' < length xs xs ! i' = x' zs ! i' = z' xz' = (x', z')
        by (auto simp: set_zip)
      define y where y = ys ! i
      define y' where y' = ys ! i'
      have sp_equiv_pair (x, y) (x', y') sp_equiv_pair (y, z) (y', z')
        using assms lens xz_def
        by (auto simp: sp_equiv_list_def pairwise_def y_def y'_def set_zip) metis+
      then show sp_equiv_pair xz xz'
        by (auto simp: xz_def)
    qed
  then show ?thesis
    using assms
    by (auto simp: sp_equiv_list_def)
qed

lemma sp_equiv_list_link: sp_equiv_list (map σ ns) (map τ ns) ⟷ sp_equiv σ τ (set ns)
  apply (auto simp: sp_equiv_list_def sp_equiv_def pairwise_def set_zip in_set_conv_nth)
  apply (metis nth_map)
  apply (metis nth_map)
  apply fastforce+
  done

lemma ad_agr_list_comm: ad_agr_list X xs ys ⟹ ad_agr_list X ys xs
  using ad_equiv_list_comm sp_equiv_list_comm
  by (fastforce simp: ad_agr_list_def)

lemma ad_agr_list_mono: X ⊆ Y ⟹ ad_agr_list Y ys xs ⟹ ad_agr_list X ys xs
  using ad_equiv_list_mono
  by (force simp: ad_agr_list_def)

lemma ad_agr_list_rev_mono:
  assumes Y ⊆ X ad_agr_list Y ys xs Inl -' set xs ⊆ Y Inl -' set ys ⊆ Y
  shows ad_agr_list X ys xs
  proof -
    have (a, b) ∈ set (zip ys xs) ⟹ ad_equiv_pair Y (a, b) ⟹ ad_equiv_pair X (a, b) for a b
      using assms
      apply (cases a; cases b)
      apply (auto simp: ad_agr_list_def ad_equiv_list_def vimage_def set_zip)
    unfolding ad_equiv_pair.simps
      apply (metis Collect_mem_eq Collect_mono_iff imageI nth_mem)
      apply (metis Collect_mem_eq Collect_mono_iff imageI nth_mem)
      apply (metis Collect_mem_eq Collect_mono_iff imageI nth_mem)
      apply (metis Inl_Inr_False image_iff)
    done
  then show ?thesis
    using assms
    by (fastforce simp: ad_agr_list_def ad_equiv_list_def)
qed

```

**lemma** *ad\_agr\_list\_trans*:  $ad\_agr\_list\ X\ xs\ ys \implies ad\_agr\_list\ X\ ys\ zs \implies ad\_agr\_list\ X\ xs\ zs$   
**using** *ad\_equiv\_list\_trans sp\_equiv\_list\_trans*  
**by** (*force simp: ad\_agr\_list\_def*)

**lemma** *ad\_agr\_list\_refl*:  $ad\_agr\_list\ X\ xs\ xs$   
**by** (*auto simp: ad\_agr\_list\_def ad\_equiv\_list\_def set\_zip ad\_equiv\_pair.simps*  
*sp\_equiv\_list\_def pairwise\_def*)

**lemma** *ad\_agr\_list\_set*:  $ad\_agr\_list\ X\ xs\ ys \implies y \in X \implies Inl\ y \in set\ ys \implies Inl\ y \in set\ xs$   
**by** (*auto simp: ad\_agr\_list\_def ad\_equiv\_list\_def set\_zip in\_set\_conv\_nth*)  
*(metis ad\_equiv\_pair.simps image\_eqI)*

**lemma** *ad\_agr\_list\_length*:  $ad\_agr\_list\ X\ xs\ ys \implies length\ xs = length\ ys$   
**by** (*auto simp: ad\_agr\_list\_def*)

**lemma** *ad\_agr\_list\_eq*:  $set\ ys \subseteq AD \implies ad\_agr\_list\ AD\ (map\ Inl\ xs)\ (map\ Inl\ ys) \implies xs = ys$   
**by** (*fastforce simp: ad\_agr\_list\_def ad\_equiv\_list\_def set\_zip ad\_equiv\_pair.simps*  
*intro!: nth\_equalityI*)

**lemma** *sp\_equiv\_list\_subset*:  
**assumes**  $set\ ms \subseteq set\ ns$  *sp\_equiv\_list*  $(map\ \sigma\ ns)\ (map\ \sigma'\ ns)$   
**shows** *sp\_equiv\_list*  $(map\ \sigma\ ms)\ (map\ \sigma'\ ms)$   
**unfolding** *sp\_equiv\_list\_def length\_map pairwise\_def*  
**proof** (*rule conjI, rule refl, (rule ballI)+, rule impI*)  
**fix**  $x\ y$   
**assume**  $x \in set\ (zip\ (map\ \sigma\ ms)\ (map\ \sigma'\ ms))$   $y \in set\ (zip\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns))$   $x \neq y$   
**then have**  $x \in set\ (zip\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns))$   $y \in set\ (zip\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns))$   $x \neq y$   
**using** *assms(1)*  
**by** (*auto simp: set\_zip*) *(metis in\_set\_conv\_nth nth\_map subset\_iff)+*  
**then show** *sp\_equiv\_pair*  $x\ y$   
**using** *assms(2)*  
**by** (*auto simp: sp\_equiv\_list\_def pairwise\_def*)

**qed**

**lemma** *ad\_agr\_list\_subset*:  $set\ ms \subseteq set\ ns \implies ad\_agr\_list\ X\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns) \implies$   
 $ad\_agr\_list\ X\ (map\ \sigma\ ms)\ (map\ \sigma'\ ms)$   
**by** (*auto simp: ad\_agr\_list\_def ad\_equiv\_list\_def sp\_equiv\_list\_subset set\_zip*)  
*(metis (no\_types, lifting) in\_set\_conv\_nth nth\_map subset\_iff)*

**lemma** *ad\_agr\_list\_link*:  $ad\_agr\_sets\ (set\ ns)\ (set\ ns)\ AD\ \sigma\ \tau \longleftrightarrow$   
 $ad\_agr\_list\ AD\ (map\ \sigma\ ns)\ (map\ \tau\ ns)$   
**unfolding** *ad\_agr\_sets\_def ad\_agr\_list\_def*  
**using** *ad\_equiv\_list\_link sp\_equiv\_list\_link*  
**by** *fastforce*

**definition** *ad\_agr* ::  $('a, 'b)\ fo\_fmla \Rightarrow 'a\ set \Rightarrow ('a + 'c)\ val \Rightarrow ('a + 'c)\ val \Rightarrow bool$  **where**  
 $ad\_agr\ \varphi\ X\ \sigma\ \tau \longleftrightarrow ad\_agr\_sets\ (fv\_fo\_fmla\ \varphi)\ (SP\ \varphi)\ X\ \sigma\ \tau$

**lemma** *ad\_agr\_sets\_restrict*:  
 $ad\_agr\_sets\ (set\ (fv\_fo\_fmla\_list\ \varphi))\ (set\ (fv\_fo\_fmla\_list\ \varphi))\ AD\ \sigma\ \tau \implies ad\_agr\ \varphi\ AD\ \sigma\ \tau$   
**using** *sp\_equiv\_mono SP\_fv*  
**unfolding** *fv\_fo\_fmla\_list\_set*  
**by** (*auto simp: ad\_agr\_sets\_def ad\_agr\_def*) *blast*

**lemma** *finite\_Inl*:  $finite\ X \implies finite\ (Inl\ -\ 'X)$   
**using** *finite\_vimageI* [of  $X\ Inl$ ]  
**by** (*auto simp: vimage\_def*)

```

lemma ex_out:
  assumes finite X
  shows  $\exists k. k \notin X \wedge k < \text{Suc}(\text{card } X)$ 
  using card_mono[OF assms, of  $\{..<\text{Suc}(\text{card } X)\}$ ]
  by auto

lemma extend_τ:
  assumes ad_agr_sets ( $FV - \{n\}$ ) ( $S - \{n\}$ )  $X \sigma \tau S \subseteq FV$  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 (cases  $n \in S$ )
  case True
  note  $n\_in\_S = \text{True}$ 
  show ?thesis
  proof (cases  $x \in \text{Inl}' X$ )
  case True
  show ?thesis
  using assms  $n\_in\_S \text{True}$ 
  apply (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def intro!: beXI[of _ x])
  unfolding ad_equiv_pair.simps
  apply (metis True insert_Diff insert_iff subsetD)
  done
  next
  case False
  note  $\sigma\_n\_not\_Inl = \text{False}$ 
  show ?thesis
  proof (cases  $\exists m \in S - \{n\}. x = \sigma m$ )
  case True
  obtain  $m$  where  $m\_def: m \in S - \{n\} x = \sigma m$ 
  using True
  by auto
  have  $\tau\_m\_in: \tau m \in Z$ 
  using assms  $m\_def$ 
  by auto
  show ?thesis
  using assms  $n\_in\_S \sigma\_n\_not\_Inl \text{True } m\_def$ 
  by (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def intro!: beXI[of _  $\tau m$ ])
  next
  case False
  have  $out: x \notin \sigma '(S - \{n\})$ 
  using False
  by auto
  have  $fin: \text{finite}(\text{Inr}' \tau '(S - \{n\}))$ 
  using assms(3)
  by (simp add: finite_vimageI)
  obtain  $k$  where  $k\_def: \text{Inr } k \notin \tau '(S - \{n\}) k < \text{Suc}(\text{card}(\text{Inr}' \tau '(S - \{n\})))$ 
  using ex_out[OF  $fin$ ] True
  by auto
  show ?thesis
  using assms  $n\_in\_S \sigma\_n\_not\_Inl out k\_def$  assms(5)
  apply (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def intro!: beXI[of _  $\text{Inr } k$ ])
  unfolding ad_equiv_pair.simps
  apply fastforce
  apply (metis image_eqI insertE insert_Diff)
  done
qed
qed

```

```

next
  case False
  show ?thesis
  proof (cases x ∈ Inl ' X)
    case x_in: True
    then show ?thesis
      using assms False
      by (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def intro!: beXI[of _ x])
  next
  case x_out: False
  then show ?thesis
    using assms False
    apply (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def intro!: beXI[of _ Inr 0])
    unfolding ad_equiv_pair.simps
    apply fastforce
    done
  qed
qed

```

**lemma** *esat\_Pred*:

```

assumes ad_agr_sets FV S (⋃(set ' X)) σ τ fv_fo_terms_set ts ⊆ FV σ ⊙ e ts ∈ map Inl ' X
  t ∈ set ts
shows σ · e t = τ · e t
proof (cases t)
  case (Var n)
  obtain vs where vs_def: σ ⊙ e ts = map Inl vs vs ∈ X
    using assms(3)
    by auto
  have σ n ∈ set (σ ⊙ e ts)
    using assms(4)
    by (force simp: eval_eterms_def Var)
  then have σ n ∈ Inl ' ⋃ (set ' X)
    using vs_def(2)
    unfolding vs_def(1)
    by auto
  moreover have n ∈ FV
    using assms(2,4)
    by (fastforce simp: Var fv_fo_terms_set_def)
  ultimately show ?thesis
    using assms(1)
    unfolding ad_equiv_pair.simps ad_agr_sets_def Var
    by fastforce
  qed auto

```

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

```

assumes (⋀i. i ∈ fv_fo_terms_set ts ⇒ ad_equiv_pair X (σ i, τ i))
  ⋃(set_fo_term ' set ts) ⊆ X sp_equiv σ τ (fv_fo_terms_set ts)
shows sp_equiv_list (map ((·e) σ) ts) (map ((·e) τ) ts)
using assms
proof (induction ts)
  case (Cons t ts)
  have ind: sp_equiv_list (map ((·e) σ) ts) (map ((·e) τ) ts)
    using Cons
    by (auto simp: fv_fo_terms_set_def sp_equiv_def pairwise_def)
  show ?case
  proof (cases t)
    case (Const c)
    have c_X: c ∈ X

```

```

using Cons(3)
by (auto simp: Const)
have fv_t: fv_fo_term_set t = {}
by (auto simp: Const)
have t' ∈ set ts ⇒ sp_equiv_pair (σ · e t, τ · e t) (σ · e t', τ · e t') for t'
using c_X Const Cons(2)
apply (cases t')
apply (auto simp: fv_fo_terms_set_def)
unfolding ad_equiv_pair.simps
by (metis Cons(2) ad_equiv_pair.simps fv_fo_terms_setI image_insert insert_iff list.set(2)
mk_disjoint_insert)+
then show sp_equiv_list (map ((·) e) σ) (t # ts) (map ((·) e) τ) (t # ts)
using ind pairwise_insert[of sp_equiv_pair (σ · e t, τ · e t)]
unfolding sp_equiv_list_def set_zip_map
by (auto simp: sp_equiv_pair_comm fv_fo_terms_set_def fv_t)
next
case (Var n)
have ad_n: ad_equiv_pair X (σ n, τ n)
using Cons(2)
by (auto simp: fv_fo_terms_set_def Var)
have sp_equiv_Var: ∧n'. Var n' ∈ set ts ⇒ sp_equiv_pair (σ n, τ n) (σ n', τ n')
using Cons(4)
by (auto simp: sp_equiv_def pairwise_def fv_fo_terms_set_def Var)
have t' ∈ set ts ⇒ sp_equiv_pair (σ · e t, τ · e t) (σ · e t', τ · e t') for t'
using Cons(2,3) sp_equiv_Var
apply (cases t')
apply (auto simp: Var)
apply (metis SUP_le_iff ad_equiv_pair.simps ad_n fo_term.set_intros imageI subset_eq)
apply (metis SUP_le_iff ad_equiv_pair.simps ad_n fo_term.set_intros imageI subset_eq)
done
then show ?thesis
using ind pairwise_insert[of sp_equiv_pair (σ · e t, τ · e t) (λn. (σ · e n, τ · e n))] 'set ts]
unfolding sp_equiv_list_def set_zip_map
by (auto simp: sp_equiv_pair_comm)
qed
qed (auto simp: sp_equiv_def sp_equiv_list_def fv_fo_terms_set_def)

lemma esat_Pred_inf:
assumes fv_fo_terms_set ts ⊆ FV fv_fo_terms_set ts ⊆ S
adAgr_sets FV S AD σ τ adAgr_list AD (σ ⊙ e ts) vs
⋃(set_fo_term 'set ts) ⊆ AD
shows adAgr_list AD (τ ⊙ e ts) vs
proof -
have sp: sp_equiv σ τ (fv_fo_terms_set ts)
using assms(2,3) sp_equiv_mono
unfolding adAgr_sets_def
by auto
have (∧i. i ∈ fv_fo_terms_set ts ⇒ ad_equiv_pair AD (σ i, τ i))
using assms(1,3)
by (auto simp: adAgr_sets_def)
then have sp_equiv_list (map ((·) e) σ) ts (map ((·) e) τ) ts
using sp_equiv_list_fv[OF _ assms(5) sp]
by auto
moreover have t ∈ set ts ⇒ ∀i ∈ fv_fo_terms_set ts. ad_equiv_pair AD (σ i, τ i) ⇒ sp_equiv σ
τ S ⇒ ad_equiv_pair AD (σ · e t, τ · e t) for t
by (cases t) (auto simp: ad_equiv_pair.simps intro!: fv_fo_terms_setI)
ultimately have adAgr_list:
adAgr_list AD (σ ⊙ e ts) (τ ⊙ e ts)

```

```

unfolding eval_eterms_def ad_agr_list_def ad_equiv_list_link[symmetric]
using assms(1,3)
by (auto simp: ad_agr_sets_def)
show ?thesis
by (rule ad_agr_list_comm[OF ad_agr_list_trans[OF ad_agr_list_comm[OF assms(4)] ad_agr_list]])
qed

```

```

type_synonym ('a, 'c) fo_t = 'a set × nat × ('a + 'c) table

```

```

fun esat :: ('a, 'b) fo_fmula ⇒ ('a table, 'b) fo_intp ⇒ ('a + nat) val ⇒ ('a + nat) set ⇒ bool where
  esat (Pred r ts) I σ X ↔ σ ⊙ e ts ∈ map Inl ' I (r, length ts)
| esat (Bool b) I σ X ↔ b
| esat (Eqa t t') I σ X ↔ σ · e t = σ · e t'
| esat (Neg φ) I σ X ↔ ¬esat φ I σ X
| esat (Conj φ ψ) I σ X ↔ esat φ I σ X ∧ esat ψ I σ X
| esat (Disj φ ψ) I σ X ↔ esat φ I σ X ∨ esat ψ I σ X
| esat (Exists n φ) I σ X ↔ (∃ x ∈ X. esat φ I (σ(n := x)) X)
| esat (Forall n φ) I σ X ↔ (∀ x ∈ X. esat φ I (σ(n := x)) X)

```

```

fun sz_fmula :: ('a, 'b) fo_fmula ⇒ nat where
  sz_fmula (Neg φ) = Suc (sz_fmula φ)
| sz_fmula (Conj φ ψ) = Suc (sz_fmula φ + sz_fmula ψ)
| sz_fmula (Disj φ ψ) = Suc (sz_fmula φ + sz_fmula ψ)
| sz_fmula (Exists n φ) = Suc (sz_fmula φ)
| sz_fmula (Forall n φ) = Suc (Suc (Suc (Suc (sz_fmula φ))))
| sz_fmula _ = 0

```

```

lemma sz_fmula_induct[case_names Pred Bool Eqa Neg Conj Disj Exists Forall]:

```

```

  (∧ r ts. P (Pred r ts)) ⇒ (∧ b. P (Bool b)) ⇒
  (∧ t t'. P (Eqa t t')) ⇒ (∧ φ. P φ ⇒ P (Neg φ)) ⇒
  (∧ φ ψ. P φ ⇒ P ψ ⇒ P (Conj φ ψ)) ⇒ (∧ φ ψ. P φ ⇒ P ψ ⇒ P (Disj φ ψ)) ⇒
  (∧ n φ. P φ ⇒ P (Exists n φ)) ⇒ (∧ n φ. P (Exists n (Neg φ)) ⇒ P (Forall n φ)) ⇒ P φ

```

```

proof (induction sz_fmula φ arbitrary: φ rule: nat_less_induct)

```

```

case 1

```

```

  have IH: ∧ψ. sz_fmula ψ < sz_fmula φ ⇒ P ψ

```

```

    using 1

```

```

    by auto

```

```

  then show ?case

```

```

    using 1(2,3,4,5,6,7,8,9)

```

```

    by (cases φ) auto

```

```

qed

```

```

lemma esat_fv_cong: (∧ n. n ∈ fv_fo_fmula φ ⇒ σ n = σ' n) ⇒ esat φ I σ X ↔ esat φ I σ' X

```

```

proof (induction φ arbitrary: σ σ' rule: sz_fmula_induct)

```

```

case (Pred r ts)

```

```

  then show ?case

```

```

    by (auto simp: eval_eterms_def fv_fo_terms_set_def)

```

```

    (smt comp_apply eval_eterm_cong fv_fo_term_set_cong image_insert insertCI map_eq_conv
    mk_disjoint_insert)+

```

```

next

```

```

case (Eqa t t')

```

```

  then show ?case

```

```

    by (cases t; cases t') auto

```

```

next

```

```

case (Neg φ)

```

```

  show ?case

```

```

    using Neg(1)[of σ σ'] Neg(2) by auto

```

```

next

```

```

    case (Conj  $\varphi 1$   $\varphi 2$ )
    show ?case
      using Conj(1,2)[of  $\sigma$   $\sigma'$ ] Conj(3) by auto
next
  case (Disj  $\varphi 1$   $\varphi 2$ )
  show ?case
    using Disj(1,2)[of  $\sigma$   $\sigma'$ ] Disj(3) by auto
next
  case (Exists  $n$   $\varphi$ )
  show ?case
  proof (rule iffI)
    assume esat (Exists  $n$   $\varphi$ )  $I$   $\sigma$   $X$ 
    then obtain  $x$  where  $x\_def: x \in X$  esat  $\varphi$   $I$  ( $\sigma(n := x)$ )  $X$ 
      by auto
    from  $x\_def(2)$  have esat  $\varphi$   $I$  ( $\sigma'(n := x)$ )  $X$ 
      using Exists(1)[of  $\sigma(n := x)$   $\sigma'(n := x)$ ] Exists(2) by fastforce
    with  $x\_def(1)$  show esat (Exists  $n$   $\varphi$ )  $I$   $\sigma' X$ 
      by auto
  next
    assume esat (Exists  $n$   $\varphi$ )  $I$   $\sigma' X$ 
    then obtain  $x$  where  $x\_def: x \in X$  esat  $\varphi$   $I$  ( $\sigma'(n := x)$ )  $X$ 
      by auto
    from  $x\_def(2)$  have esat  $\varphi$   $I$  ( $\sigma(n := x)$ )  $X$ 
      using Exists(1)[of  $\sigma(n := x)$   $\sigma'(n := x)$ ] Exists(2) by fastforce
    with  $x\_def(1)$  show esat (Exists  $n$   $\varphi$ )  $I$   $\sigma X$ 
      by auto
  qed
next
  case (Forall  $n$   $\varphi$ )
  then show ?case
    by auto
qed auto

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_fmula  $\Rightarrow$  ('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 (Eq  $t$   $t'$ )  $I$  = 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  $\varphi$   $I$ )
  using finite_Inl
  by (induction  $\varphi$   $I$  rule: wf_fo_intp.induct)
  (auto simp: finite_set_fo_term vimage_def)

fun fo_adom :: ('a, 'c) fo_t  $\Rightarrow$  'a set where
  fo_adom (AD,  $n$ ,  $X$ ) = AD

theorem main: adAgr  $\varphi$  AD  $\sigma$   $\tau$   $\Longrightarrow$  act_edom  $\varphi$   $I$   $\subseteq$  AD  $\Longrightarrow$ 
  Inl ' AD  $\cup$  Inr ' {.. $d$   $\varphi$ }  $\subseteq$   $X$   $\Longrightarrow$   $\tau$  ' fv_fo_fmula  $\varphi$   $\subseteq$   $X$   $\Longrightarrow$ 
  esat  $\varphi$   $I$   $\sigma$  UNIV  $\longleftrightarrow$  esat  $\varphi$   $I$   $\tau$   $X$ 
proof (induction  $\varphi$  arbitrary:  $\sigma$   $\tau$  rule: sz_fmula_induct)

```

```

case (Pred r ts)
have fv_sub: fv_fo_terms_set ts  $\subseteq$  fv_fo_fmula (Pred r ts)
  by auto
have sub_AD:  $\bigcup$ (set ' I (r, length ts))  $\subseteq$  AD
  using Pred(2)
  by auto
show ?case
  unfolding esat.simps
proof (rule iffI)
  assume assm:  $\sigma \odot e$  ts  $\in$  map Inl ' I (r, length ts)
  have  $\sigma \odot e$  ts =  $\tau \odot e$  ts
    using esat_Pred[OF ad_agr_sets_mono[OF sub_AD Pred(1)[unfolded ad_agr_def]]
      fv_sub assm]
    by (auto simp: eval_eterms_def)
  with assm show  $\tau \odot e$  ts  $\in$  map Inl ' I (r, length ts)
    by auto
next
  assume assm:  $\tau \odot e$  ts  $\in$  map Inl ' I (r, length ts)
  have  $\tau \odot e$  ts =  $\sigma \odot e$  ts
    using esat_Pred[OF ad_agr_sets_comm[OF ad_agr_sets_mono[OF
      sub_AD Pred(1)[unfolded ad_agr_def]]] fv_sub assm]
    by (auto simp: eval_eterms_def)
  with assm show  $\sigma \odot e$  ts  $\in$  map Inl ' I (r, length ts)
    by auto
qed
next
case (Eqa x1 x2)
show ?case
proof (cases x1; cases x2)
  fix c c'
  assume x1 = Const c x2 = Const c'
  with Eqa show ?thesis
    by auto
next
  fix c m'
  assume assms: x1 = Const c x2 = Var m'
  with Eqa(1,2) have  $\sigma$  m' = Inl c  $\longleftrightarrow$   $\tau$  m' = Inl c
    apply (auto simp: ad_agr_def ad_agr_sets_def)
    unfolding ad_equiv_pair.simps
    by fastforce+
  with assms show ?thesis
    by fastforce
next
  fix m c'
  assume assms: x1 = Var m x2 = Const c'
  with Eqa(1,2) have  $\sigma$  m = Inl c'  $\longleftrightarrow$   $\tau$  m = Inl c'
    apply (auto simp: ad_agr_def ad_agr_sets_def)
    unfolding ad_equiv_pair.simps
    by fastforce+
  with assms show ?thesis
    by auto
next
  fix m m'
  assume assms: x1 = Var m x2 = Var m'
  with Eqa(1,2) have  $\sigma$  m =  $\sigma$  m'  $\longleftrightarrow$   $\tau$  m =  $\tau$  m'
    by (auto simp: ad_agr_def ad_agr_sets_def sp_equiv_def pairwise_def split: if_splits)
  with assms show ?thesis
    by auto

```



```

qed
next
case (Neg  $\varphi$ )
from Neg(2) have ad_agr  $\varphi$  AD  $\sigma$   $\tau$ 
  by (auto simp: ad_agr_def)
with Neg show ?case
  by auto
next
case (Conj  $\varphi_1$   $\varphi_2$ )
have aux: ad_agr  $\varphi_1$  AD  $\sigma$   $\tau$  ad_agr  $\varphi_2$  AD  $\sigma$   $\tau$ 
  Inl ' AD  $\cup$  Inr ' {.. $d$   $\varphi_1$ }  $\subseteq$  X Inl ' AD  $\cup$  Inr ' {.. $d$   $\varphi_2$ }  $\subseteq$  X
   $\tau$  ' fv_fo_fmula  $\varphi_1 \subseteq$  X  $\tau$  ' fv_fo_fmula  $\varphi_2 \subseteq$  X
using Conj(3,5,6)
by (auto simp: ad_agr_def ad_agr_sets_def sp_equiv_def pairwise_def)
show ?case
  using Conj(1)[OF aux(1) _ aux(3) aux(5)] Conj(2)[OF aux(2) _ aux(4) aux(6)] Conj(4)
  by auto
next
case (Disj  $\varphi_1$   $\varphi_2$ )
have aux: ad_agr  $\varphi_1$  AD  $\sigma$   $\tau$  ad_agr  $\varphi_2$  AD  $\sigma$   $\tau$ 
  Inl ' AD  $\cup$  Inr ' {.. $d$   $\varphi_1$ }  $\subseteq$  X Inl ' AD  $\cup$  Inr ' {.. $d$   $\varphi_2$ }  $\subseteq$  X
   $\tau$  ' fv_fo_fmula  $\varphi_1 \subseteq$  X  $\tau$  ' fv_fo_fmula  $\varphi_2 \subseteq$  X
using Disj(3,5,6)
by (auto simp: ad_agr_def ad_agr_sets_def sp_equiv_def pairwise_def)
show ?case
  using Disj(1)[OF aux(1) _ aux(3) aux(5)] Disj(2)[OF aux(2) _ aux(4) aux(6)] Disj(4)
  by auto
next
case (Exists  $m$   $\varphi$ )
show ?case
proof (rule iffI)
  assume esat (Exists  $m$   $\varphi$ ) I  $\sigma$  UNIV
  then obtain  $x$  where assm: esat  $\varphi$  I ( $\sigma(m := x)$ ) UNIV
  by auto
  have  $m \in SP \varphi \implies Suc$  (card (Inr -'  $\tau$  ' ( $SP \varphi - \{m\}$ )))  $\leq$  card ( $SP \varphi$ )
  by (metis Diff_insert_absorb card_image card_le_Suc_iff finite_Diff finite_SP
    image_vimage_subset inj_Inr mk_disjoint_insert surj_card_le)
  moreover have card (Inr -'  $\tau$  '  $SP \varphi$ )  $\leq$  card ( $SP \varphi$ )
  by (metis card_image finite_SP image_vimage_subset inj_Inr surj_card_le)
  ultimately have  $max$  1 (card (Inr -'  $\tau$  ' ( $SP \varphi - \{m\}$ ))) + (if  $m \in SP \varphi$  then 1 else 0)  $\leq$   $d$   $\varphi$ 
  using d_pos card_SP_d[of  $\varphi$ ]
  by auto
  then have  $\exists x' \in X$ . ad_agr  $\varphi$  AD ( $\sigma(m := x)$ ) ( $\tau(m := x')$ )
  using extend_ $\tau$ [OF Exists(2)][unfolded ad_agr_def fv_fo_fmula.simps SP.simps
    SP_fv[of  $\varphi$ ] finite_SP Exists(5)[unfolded fv_fo_fmula.simps]]
    Exists(4)
  by (force simp: ad_agr_def)
  then obtain  $x'$  where  $x'_def$ :  $x' \in X$  ad_agr  $\varphi$  AD ( $\sigma(m := x)$ ) ( $\tau(m := x')$ )
  by auto
  from Exists(5) have  $\tau(m := x')$  ' fv_fo_fmula  $\varphi \subseteq$  X
  using  $x'_def$ (1) by fastforce
  then have esat  $\varphi$  I ( $\tau(m := x')$ ) X
  using Exists  $x'_def$ (1,2) assm
  by fastforce
  with  $x'_def$  show esat (Exists  $m$   $\varphi$ ) I  $\tau$  X
  by auto
next
assume esat (Exists  $m$   $\varphi$ ) I  $\tau$  X

```

```

then obtain  $z$  where  $assm: z \in X \text{ esat } \varphi \ I \ (\tau(m := z)) \ X$ 
  by auto
have  $ad\_agr: ad\_agr\_sets \ (fv\_fo\_fmla \ \varphi - \{m\}) \ (SP \ \varphi - \{m\}) \ AD \ \tau \ \sigma$ 
  using  $Exists(2)[unfolded \ ad\_agr\_def \ fv\_fo\_fmla.simps \ SP.simps]$ 
  by  $(rule \ ad\_agr\_sets\_comm)$ 
have  $\exists x. \ ad\_agr \ \varphi \ AD \ (\sigma(m := x)) \ (\tau(m := z))$ 
  using  $extend\_tau[OF \ ad\_agr \ SP\_fv[of \ \varphi] \ finite\_SP \ subset\_UNIV \ subset\_UNIV] \ ad\_agr\_sets\_comm$ 
  unfolding  $ad\_agr\_def$ 
  by fastforce
then obtain  $x$  where  $x\_def: ad\_agr \ \varphi \ AD \ (\sigma(m := x)) \ (\tau(m := z))$ 
  by auto
have  $\tau(m := z) \ ' \ fv\_fo\_fmla \ (Exists \ m \ \varphi) \ \subseteq \ X$ 
  using  $Exists$ 
  by fastforce
with  $x\_def$  have  $esat \ \varphi \ I \ (\sigma(m := x)) \ UNIV$ 
  using  $Exists \ assm$ 
  by fastforce
then show  $esat \ (Exists \ m \ \varphi) \ I \ \sigma \ UNIV$ 
  by auto
qed
next
case  $(Forall \ n \ \varphi)$ 
have  $unfold: act\_edom \ (Forall \ n \ \varphi) \ I = act\_edom \ (Exists \ n \ (Neg \ \varphi)) \ I$ 
   $Inl \ ' \ AD \ \cup \ Inr \ ' \ \{..\lt d \ (Forall \ n \ \varphi)\} = Inl \ ' \ AD \ \cup \ Inr \ ' \ \{..\lt d \ (Exists \ n \ (Neg \ \varphi))\}$ 
   $fv\_fo\_fmla \ (Forall \ n \ \varphi) = fv\_fo\_fmla \ (Exists \ n \ (Neg \ \varphi))$ 
  by auto
have  $pred: ad\_agr \ (Exists \ n \ (Neg \ \varphi)) \ AD \ \sigma \ \tau$ 
  using  $Forall(2)$ 
  by  $(auto \ simp: \ ad\_agr\_def)$ 
show  $?case$ 
  using  $Forall(1)[OF \ pred \ Forall(3,4,5)[unfolded \ unfold]]$ 
  by auto
qed auto

lemma  $main\_cor\_inf:$ 
  assumes  $ad\_agr \ \varphi \ AD \ \sigma \ \tau \ act\_edom \ \varphi \ I \ \subseteq \ AD \ d \ \varphi \ \leq \ n$ 
   $\tau \ ' \ fv\_fo\_fmla \ \varphi \ \subseteq \ Inl \ ' \ AD \ \cup \ Inr \ ' \ \{..\lt n\}$ 
  shows  $esat \ \varphi \ I \ \sigma \ UNIV \ \longleftrightarrow \ esat \ \varphi \ I \ \tau \ (Inl \ ' \ AD \ \cup \ Inr \ ' \ \{..\lt n\})$ 
proof  $-$ 
  show  $?thesis$ 
  using  $main[OF \ assms(1,2) \ _ \ assms(4)] \ assms(3)$ 
  by fastforce
qed

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  $-$ 
  show  $?thesis$ 
  using  $main[OF \ assms(1,2) \ subset\_UNIV \ subset\_UNIV]$ 
  by auto
qed

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
using esat_UNIV_cong[OF iffD2[OF ad_agr_def, OF ad_agr_sets_mono'[OF SP_fv],
  OF iffD2[OF ad_agr_list_link, OF assms(1), unfolded fv_fo_fmula_list_set]] assms(2)] .

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_fmula
  assumes fin: wf_fo_intp  $\varphi$  I
  shows sat  $\varphi$  I  $\sigma$   $\longleftrightarrow$  esat  $\varphi$  I (Inl  $\circ$   $\sigma$  :: nat  $\Rightarrow$  'a + nat) UNIV
  using assms
proof (induction  $\varphi$  arbitrary: I  $\sigma$  rule: sz_fmula_induct)
  case (Pred r ts)
  show ?case
    unfolding sat.simps esat.simps comp_def[symmetric] eval_terms_eterms[symmetric]
    by auto
  next
  case (Eqv t t')
  show ?case
    by (cases t; cases t') auto
  next
  case (Exists n  $\varphi$ )
  show ?case
proof (rule iffI)
  assume sat (Exists n  $\varphi$ ) I  $\sigma$ 
  then obtain x where x_def: esat  $\varphi$  I (Inl  $\circ$   $\sigma$ (n := x)) UNIV
    using Exists
    by fastforce
  have Inl_unfold: Inl  $\circ$   $\sigma$ (n := x) = (Inl  $\circ$   $\sigma$ )(n := Inl x)
    by auto
  show esat (Exists n  $\varphi$ ) I (Inl  $\circ$   $\sigma$ ) UNIV
    using x_def
    unfolding Inl_unfold
    by auto
  next
  assume esat (Exists n  $\varphi$ ) I (Inl  $\circ$   $\sigma$ ) UNIV
  then obtain x where x_def: esat  $\varphi$  I ((Inl  $\circ$   $\sigma$ )(n := x)) UNIV
    by auto
  show sat (Exists n  $\varphi$ ) I  $\sigma$ 
proof (cases x)
  case (Inl a)
  have Inl_unfold: (Inl  $\circ$   $\sigma$ )(n := x) = Inl  $\circ$   $\sigma$ (n := a)
    by (auto simp: Inl)
  show ?thesis
    using x_def[unfolded Inl_unfold] Exists
    by fastforce
  next
  case (Inr b)
  obtain c where c_def: c  $\notin$  act_edom  $\varphi$  I  $\cup$   $\sigma$  'fv_fo_fmula  $\varphi$ 
    using arb_element finite_act_edom[OF Exists(2), simplified] finite_fv_fo_fmula
    by (metis finite_Un finite_imageI)
  have wf_local: wf_fo_intp  $\varphi$  I
    using Exists(2)
    by auto
  have (a, a')  $\in$  set (zip (map ( $\lambda$ x. if x = n then Inr b else (Inl  $\circ$   $\sigma$ ) x) (fv_fo_fmula_list  $\varphi$ ))
    (map ( $\lambda$ a. Inl (if a = n then c else  $\sigma$  a)) (fv_fo_fmula_list  $\varphi$ )))  $\implies$ 
    ad_equiv_pair (act_edom  $\varphi$  I) (a, a') for a a'

```

```

    using c_def
    by (cases a; cases a') (auto simp: set_zip ad_equiv_pair.simps split: if_splits)
  then have sat  $\varphi$  I ( $\sigma(n := c)$ )
    using c_def[folded fv_fo_fmula_list_set]
    by (auto simp: ad_agr_list_def ad_equiv_list_def fun_upd_def sp_equiv_list_def pairwise_def
set_zip split: if_splits
      intro!: Exists(1)[OF wf_local, THEN iffD2, OF esat_UNIV_ad_agr_list[OF _ subset_refl,
THEN iffD1, OF _ x_def[unfolded Inr]]])
    then show ?thesis
      by auto
  qed
qed
next
case (Forall n  $\varphi$ )
show ?case
  using Forall(1)[of I  $\sigma$ ] Forall(2)
  by auto
qed auto

```

```

lemma sat_ad_agr_list:
  fixes  $\varphi :: ('a :: infinite, 'b) fo_fmula$ 
  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_fmula_list  $\varphi$ ))
  (map (Inl  $\circ$   $\tau$ ) (fv_fo_fmula_list  $\varphi$ )) act_edom  $\varphi$  I  $\subseteq$  AD
  shows sat  $\varphi$  I  $\sigma \longleftrightarrow$  sat  $\varphi$  I  $\tau$ 
  using esat_UNIV_ad_agr_list[OF assms(2,3)] sat_esat_conv[OF assms(1)]
  by auto

```

```

definition nfv :: ('a, 'b) fo_fmula  $\Rightarrow$  nat where
  nfv  $\varphi =$  length (fv_fo_fmula_list  $\varphi$ )

```

```

lemma nfv_card: nfv  $\varphi =$  card (fv_fo_fmula_list  $\varphi$ )
proof -
  have distinct (fv_fo_fmula_list  $\varphi$ )
    using sorted_distinct_fv_list
    by auto
  then have length (fv_fo_fmula_list  $\varphi$ ) = card (set (fv_fo_fmula_list  $\varphi$ ))
    using distinct_card by fastforce
  then show ?thesis
    unfolding fv_fo_fmula_list_set by (auto simp: nfv_def)
qed

```

```

fun rremdups :: 'a list  $\Rightarrow$  'a list where
  rremdups [] = []
| rremdups (x # xs) = x # rremdups (filter (( $\neq$ ) x) xs)

```

```

lemma filter_rremdups_filter: filter P (rremdups (filter Q xs)) =
  rremdups (filter ( $\lambda x. P x \wedge Q x$ ) xs)
  apply (induction xs arbitrary: Q)
  apply auto
  by metis

```

```

lemma filter_rremdups: filter P (rremdups xs) = rremdups (filter P xs)
  using filter_rremdups_filter[where Q= $\lambda_. True$ ]
  by auto

```

```

lemma filter_take:  $\exists j. filter P (take i xs) = take j (filter P xs)$ 

```

```

apply (induction xs arbitrary: i)
  apply (auto)
  apply (metis filter.simps(1) filter.simps(2) take_Cons' take_Suc_Cons)
apply (metis filter.simps(2) take0 take_Cons')
done

```

```

lemma rremdups_take:  $\exists j. rremdups (take\ i\ xs) = take\ j\ (rremdups\ xs)$ 
proof (induction xs arbitrary: i)
  case (Cons x xs)
  show ?case
  proof (cases i)
    case (Suc n)
    obtain j where j_def: rremdups (take n xs) = take j (rremdups xs)
      using Cons by auto
    obtain j' where j'_def: filter (( $\neq$ ) x) (take j (rremdups xs)) =
      take j' (filter (( $\neq$ ) x) (rremdups xs))
      using filter_take
      by blast
    show ?thesis
    by (auto simp: Suc filter_rremdups[symmetric] j_def j'_def intro: exI[of _ Suc j'])
  qed (auto simp add: take_Cons')
qed auto

```

```

lemma rremdups_app: rremdups (xs @ [x]) = rremdups xs @ (if x  $\in$  set xs then [] else [x])
apply (induction xs)
apply auto
apply (smt filter.simps(1) filter.simps(2) filter_append filter_rremdups)+
done

```

```

lemma rremdups_set: set (rremdups xs) = set xs
by (induction xs) (auto simp: filter_rremdups[symmetric])

```

```

lemma distinct_rremdups: distinct (rremdups xs)
proof (induction length xs arbitrary: xs rule: nat_less_induct)
  case 1
  then have IH:  $\bigwedge m\ ys. length\ (ys :: 'a\ list) < length\ xs \implies distinct\ (rremdups\ ys)$ 
    by auto
  show ?case
  proof (cases xs)
    case (Cons z zs)
    show ?thesis
    using IH
    by (auto simp: Cons rremdups_set le_imp_less_Suc)
  qed auto
qed

```

```

lemma length_rremdups: length (rremdups xs) = card (set xs)
using distinct_card[OF distinct_rremdups]
by (subst eq_commute) (auto simp: rremdups_set)

```

```

lemma set_map_filter_sum: set (List.map_filter (case_sum Map.empty Some) xs) = Inr -' set xs
by (induction xs) (auto simp: List.map_filter_simps split: sum.splits)

```

```

definition nats :: nat list  $\Rightarrow$  bool where
  nats ns = (ns = [0.. $length\ ns$ ])

```

```

definition fo_nmlzd :: 'a set  $\Rightarrow$  ('a + nat) list  $\Rightarrow$  bool where
  fo_nmlzd AD xs  $\longleftrightarrow$  Inl -' set xs  $\subseteq$  AD  $\wedge$ 

```

```

    (let ns = List.map_filter (case_sum Map.empty Some) xs in nats (rremdups ns))

lemma fo_nmlzd_all_AD:
  assumes set xs  $\subseteq$  Inl ' AD
  shows fo_nmlzd AD xs
proof -
  have List.map_filter (case_sum Map.empty Some) xs = []
    using assms
  by (induction xs) (auto simp: List.map_filter_simps)
then show ?thesis
  using assms
  by (auto simp: fo_nmlzd_def nats_def Let_def)
qed

lemma card_Inr_vimage_le_length: card (Inr -' set xs)  $\leq$  length xs
proof -
  have card (Inr -' set xs)  $\leq$  card (set xs)
    by (meson List.finite_set card_inj_on_le vimage_vimage_subset inj_Inr)
  moreover have ...  $\leq$  length xs
    by (rule card_length)
  finally show ?thesis .
qed

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))}
proof -
  have Inl -' set xs  $\subseteq$  AD
    using assms
  by (auto simp: fo_nmlzd_def)
moreover have Inr -' set xs = {.. $\text{card}$  (Inr -' set xs)}
  using assms
  by (auto simp: Let_def fo_nmlzd_def nats_def length_rremdups set_map_filter_sum rremdups_set
    dest!: arg_cong[of _ _ set])
ultimately have set xs = set xs  $\cap$  Inl ' AD  $\cup$  Inr ' {.. $\text{card}$  (Inr -' set xs)}
  by auto (metis (no_types, lifting) UNIV_I UNIV_sum UnE image_iff subset_iff vimageI)
then show ?thesis
  using card_Inr_vimage_le_length[of xs]
  by (metis min.absorb2)
qed

lemma map_filter_take:  $\exists j. \text{List.map\_filter } f (\text{take } i \text{ xs}) = \text{take } j (\text{List.map\_filter } f \text{ xs})$ 
  apply (induction xs arbitrary: i)
  apply (auto simp: List.map_filter_simps split: option.splits)
  apply (metis map_filter_simps(1) option.case(1) take0 take_Cons')
  apply (metis map_filter_simps(1) map_filter_simps(2) option.case(2) take_Cons' take_Suc_Cons)
  done

lemma fo_nmlzd_take: assumes fo_nmlzd AD xs
  shows fo_nmlzd AD (take i xs)
proof -
  have aux: rremdups zs = [0.. $\text{length}$  (rremdups zs)]  $\implies$  rremdups (take j zs) =
    [0.. $\text{length}$  (rremdups (take j zs))] for j zs
    using rremdups_take[of j zs]
  by (auto simp add: min_def) (metis add_0 linorder_le_cases take_upt)
show ?thesis
  using assms map_filter_take[of case_sum Map.empty Some i xs] set_take_subset
  using aux[where ?zs=List.map_filter (case_sum Map.empty Some) xs]

```

by (fastforce simp: fo\_nmlzd\_def vimage\_def nats\_def Let\_def)  
qed

**lemma** map\_filter\_app:  $List.map\_filter\ f\ (xs\ @\ [x]) = List.map\_filter\ f\ xs\ @\ (case\ f\ x\ of\ Some\ y\ \Rightarrow\ [y]\ |\ \_ \Rightarrow\ [])$   
by (induction xs) (auto simp: List.map\_filter\_simps split: option.splits)

**lemma** fo\_nmlzd\_app\_Inr:  $Inr\ n\ \notin\ set\ xs \Rightarrow Inr\ n' \notin set\ xs \Rightarrow fo\_nmlzd\ AD\ (xs\ @\ [Inr\ n]) \Rightarrow fo\_nmlzd\ AD\ (xs\ @\ [Inr\ n']) \Rightarrow n = n'$   
by (auto simp: List.map\_filter\_simps fo\_nmlzd\_def nats\_def Let\_def map\_filter\_app rremdups\_app set\_map\_filter\_sum)

**fun** all\_tuples :: 'c set  $\Rightarrow$  nat  $\Rightarrow$  'c table **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 **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  $\Rightarrow$  finite (all\_tuples xs n)  
by (induction xs n rule: all\_tuples.induct) auto

**lemma** nall\_tuples\_finite: finite AD  $\Rightarrow$  finite (nall\_tuples AD n)  
by (auto simp: nall\_tuples\_def all\_tuples\_finite)

**lemma** all\_tuplesI: length vs = n  $\Rightarrow$  set vs  $\subseteq$  xs  $\Rightarrow$  vs  $\in$  all\_tuples xs n

**proof** (induction xs n arbitrary: vs rule: all\_tuples.induct)  
case (2 xs n)  
then obtain w ws **where** vs = w # ws length ws = n set ws  $\subseteq$  xs w  $\in$  xs  
by (metis Suc\_length\_conv contra\_subsetD list.set\_intros(1) order\_trans set\_subset\_Cons)  
**with** 2(1) **show** ?case  
by auto  
qed auto

**lemma** nall\_tuplesI: length vs = n  $\Rightarrow$  fo\_nmlzd AD vs  $\Rightarrow$  vs  $\in$  nall\_tuples AD n  
**using** fo\_nmlzd\_set[of AD vs]  
by (auto simp: nall\_tuples\_def intro!: all\_tuplesI)

**lemma** all\_tuplesD: vs  $\in$  all\_tuples xs n  $\Rightarrow$  length vs = n  $\wedge$  set vs  $\subseteq$  xs  
by (induction xs n arbitrary: vs rule: all\_tuples.induct) auto+

**lemma** all\_tuples\_setD: vs  $\in$  all\_tuples xs n  $\Rightarrow$  set vs  $\subseteq$  xs  
by (auto dest: all\_tuplesD)

**lemma** nall\_tuplesD: vs  $\in$  nall\_tuples AD n  $\Rightarrow$   
length vs = n  $\wedge$  set vs  $\subseteq$  Inl 'AD  $\cup$  Inr ' {..<n}  $\wedge$  fo\_nmlzd AD vs  
by (auto simp: nall\_tuples\_def dest: all\_tuplesD)

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

**proof** (induction xs n rule: all\_tuples.induct)  
case (2 xs n)  
**show** ?case  
**proof** (rule subset\_antisym; rule subsetI)  
fix ys  
assume ys  $\in$  all\_tuples xs (Suc n)  
then show ys  $\in$  {ys. length ys = Suc n  $\wedge$  set ys  $\subseteq$  xs}  
using 2 by auto  
next

```

fix ys
assume ys ∈ {ys. length ys = Suc n ∧ set ys ⊆ xs}
then have assm: length ys = Suc n set ys ⊆ xs
  by auto
then obtain z zs where zs_def: ys = z # zs z ∈ xs length zs = n set zs ⊆ xs
  by (cases ys) auto
with 2 have zs ∈ all_tuples xs n
  by auto
with zs_def(1,2) show ys ∈ all_tuples xs (Suc n)
  by auto
qed
qed auto

lemma nall_tuples_set: nall_tuples AD n = {ys. length ys = n ∧ fo_nmlzd AD ys}
using fo_nmlzd_set[of AD] card_Inr_vimage_le_length
by (auto simp: nall_tuples_def all_tuples_set) (smt UnE nall_tuplesD nall_tuplesI subsetD)

fun pos :: 'a ⇒ 'a list ⇒ nat option where
  pos a [] = None
| pos a (x # xs) =
  (if a = x then Some 0 else (case pos a xs of Some n ⇒ Some (Suc n) | _ ⇒ None))

lemma pos_set: pos a xs = Some i ⇒ a ∈ set xs
by (induction a xs arbitrary: i rule: pos.induct) (auto split: if_splits option.splits)

lemma pos_length: pos a xs = Some i ⇒ i < length xs
by (induction a xs arbitrary: i rule: pos.induct) (auto split: if_splits option.splits)

lemma pos_sound: pos a xs = Some i ⇒ i < length xs ∧ xs ! i = a
by (induction a xs arbitrary: i rule: pos.induct) (auto split: if_splits option.splits)

lemma pos_complete: pos a xs = None ⇒ a ∉ set xs
by (induction a xs rule: pos.induct) (auto split: if_splits option.splits)

fun rem_nth :: nat ⇒ 'a list ⇒ 'a 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 < length xs ⇒ length (rem_nth i xs) = length xs - 1
by (induction i xs rule: rem_nth.induct) auto

lemma rem_nth_take_drop: i < length xs ⇒ rem_nth i xs = take i xs @ drop (Suc i) xs
by (induction i xs rule: rem_nth.induct) auto

lemma rem_nth_sound: distinct xs ⇒ pos n xs = Some i ⇒
  rem_nth i (map σ xs) = map σ (filter ((≠) n) xs)
apply (induction xs arbitrary: i)
apply (auto simp: pos_set split: option.splits)
by (metis (mono_tags, lifting) filter_True)

fun add_nth :: nat ⇒ 'a ⇒ 'a list ⇒ 'a list where
  add_nth 0 a xs = a # xs
| add_nth (Suc n) a xs = (case xs of x # xs ⇒ x # add_nth n a xs)

lemma add_nth_length: i ≤ length xs ⇒ length (add_nth i z xs) = Suc (length xs)
by (induction i z xs rule: add_nth.induct) (auto split: list.splits)

```



**lemma** *add\_nth\_take\_drop*:  $i \leq \text{length } zs \implies \text{add\_nth } i \ v \ zs = \text{take } i \ zs \ @ \ v \ # \ \text{drop } i \ zs$   
**by** (*induction*  $i \ v \ zs$  *rule*: *add\_nth.induct*) (*auto split*: *list.splits*)

**lemma** *add\_nth\_rem\_nth\_map*:  $\text{distinct } xs \implies \text{pos } n \ xs = \text{Some } i \implies$   
 $\text{add\_nth } i \ a \ (\text{rem\_nth } i \ (\text{map } \sigma \ xs)) = \text{map } (\sigma(n := a)) \ xs$   
**by** (*induction*  $xs$  *arbitrary*:  $i$ ) (*auto simp*: *pos\_set split*: *option.splits*)

**lemma** *add\_nth\_rem\_nth\_self*:  $i < \text{length } xs \implies \text{add\_nth } i \ (xs ! i) \ (\text{rem\_nth } i \ xs) = xs$   
**by** (*induction*  $i \ xs$  *rule*: *rem\_nth.induct*) *auto*

**lemma** *rem\_nth\_add\_nth*:  $i \leq \text{length } zs \implies \text{rem\_nth } i \ (\text{add\_nth } i \ z \ zs) = zs$   
**by** (*induction*  $i \ z \ zs$  *rule*: *add\_nth.induct*) (*auto split*: *list.splits*)

**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*  $(n, x) \ # \ \text{merge } nxs \ ((m, y) \ # \ mys)$   
  *else*  $(m, y) \ # \ \text{merge } ((n, x) \ # \ nxs) \ mys$ )

**lemma** *merge\_Nil2*[*simp*]:  $\text{merge } nxs \ [] = nxs$   
**by** (*cases*  $nxs$ ) *auto*

**lemma** *merge\_length*:  $\text{length } (\text{merge } nxs \ mys) = \text{length } (\text{map } \text{fst } nxs \ @ \ \text{map } \text{fst } mys)$   
**by** (*induction*  $nxs \ mys$  *rule*: *merge.induct*) *auto*

**lemma** *insort\_aux\_le*:  $\forall x \in \text{set } nxs. n \leq \text{fst } x \implies \forall x \in \text{set } mys. m \leq \text{fst } x \implies n \leq m \implies$   
 $\text{insort } n \ (\text{sort } (\text{map } \text{fst } nxs \ @ \ m \ # \ \text{map } \text{fst } mys)) = n \ # \ \text{sort } (\text{map } \text{fst } nxs \ @ \ m \ # \ \text{map } \text{fst } mys)$   
**by** (*induction*  $nxs$ ) (*auto simp*: *insort\_is\_Cons insort\_left\_comm*)

**lemma** *insort\_aux\_gt*:  $\forall x \in \text{set } nxs. n \leq \text{fst } x \implies \forall x \in \text{set } mys. m \leq \text{fst } x \implies \neg n \leq m \implies$   
 $\text{insort } n \ (\text{sort } (\text{map } \text{fst } nxs \ @ \ m \ # \ \text{map } \text{fst } mys)) =$   
 $m \ # \ \text{insort } n \ (\text{sort } (\text{map } \text{fst } nxs \ @ \ \text{map } \text{fst } mys))$   
**apply** (*induction*  $nxs$ )  
**apply** (*auto simp*: *insort\_is\_Cons*)  
**by** (*metis* *dual\_order.trans insort\_key.simps(2) insort\_left\_comm*)

**lemma** *map\_fst\_merge*:  $\text{sorted\_distinct } (\text{map } \text{fst } nxs) \implies \text{sorted\_distinct } (\text{map } \text{fst } mys) \implies$   
 $\text{map } \text{snd } (\text{merge } nxs \ mys) = \text{sort } (\text{map } \text{fst } nxs \ @ \ \text{map } \text{fst } mys)$   
**by** (*induction*  $nxs \ mys$  *rule*: *merge.induct*)  
  (*auto simp add*: *sorted\_sort\_id insort\_is\_Cons insort\_aux\_le insort\_aux\_gt*)

**lemma** *merge\_map'*:  $\text{sorted\_distinct } (\text{map } \text{fst } nxs) \implies \text{sorted\_distinct } (\text{map } \text{fst } mys) \implies$   
 $\text{fst } ' \ \text{set } nxs \cap \text{fst } ' \ \text{set } mys = \{\} \implies$   
 $\text{map } \text{snd } nxs = \text{map } \sigma \ (\text{map } \text{fst } nxs) \implies \text{map } \text{snd } mys = \text{map } \sigma \ (\text{map } \text{fst } mys) \implies$   
 $\text{map } \text{snd } (\text{merge } nxs \ mys) = \text{map } \sigma \ (\text{sort } (\text{map } \text{fst } nxs \ @ \ \text{map } \text{fst } mys))$   
**by** (*induction*  $nxs \ mys$  *rule*: *merge.induct*)  
  (*auto simp*: *sorted\_sort\_id insort\_is\_Cons insort\_aux\_le insort\_aux\_gt*)

**lemma** *merge\_map*:  $\text{sorted\_distinct } ns \implies \text{sorted\_distinct } ms \implies \text{set } ns \cap \text{set } ms = \{\} \implies$   
 $\text{map } \text{snd } (\text{merge } (\text{zip } ns \ (\text{map } \sigma \ ns)) \ (\text{zip } ms \ (\text{map } \sigma \ ms))) = \text{map } \sigma \ (\text{sort } (ns \ @ \ ms))$   
**using** *merge\_map'*[*of*  $\text{zip } ns \ (\text{map } \sigma \ ns) \ \text{zip } ms \ (\text{map } \sigma \ ms) \ \sigma$ ]  
**by** *auto* (*metis* *length\_map list.set\_map map\_fst\_zip*)

**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**  
*fo\_nmlz\_rec*  $i \ m \ AD \ [] = []$   
| *fo\_nmlz\_rec*  $i \ m \ AD \ (\text{Inl } x \ # \ xs) = (\text{if } x \in AD \ \text{then } \text{Inl } x \ # \ \text{fo\_nmlz\_rec } i \ m \ AD \ xs \ \text{else}$

```

    (case m (Inl x) of None ⇒ Inr i # fo_nmlz_rec (Suc i) (m(Inl x ↦ i)) AD xs
    | Some j ⇒ Inr j # fo_nmlz_rec i m AD xs)
| fo_nmlz_rec i m AD (Inr n # xs) = (case m (Inr n) of None ⇒
    Inr i # fo_nmlz_rec (Suc i) (m(Inr n ↦ i)) AD xs
    | Some j ⇒ Inr j # fo_nmlz_rec i m AD xs)

lemma fo_nmlz_rec_sound: ran m ⊆ {..} ⇒ filter ((≤) i) (rremdups
(List.map_filter (case_sum Map.empty Some) (fo_nmlz_rec i m AD xs))) = ns ⇒
ns = [i.. + length ns]
proof (induction i m AD xs arbitrary: ns rule: fo_nmlz_rec.induct)
case 2 i m AD x xs
then show ?case
proof (cases x ∈ AD)
case False
show ?thesis
proof (cases m (Inl x))
case None
have pred: ran (m(Inl x ↦ i)) ⊆ {..

```

```

next
  case (Some j)
  then have j_lt_i: j < i
    using 3(3)
    by (auto simp: ran_def)
  have ns_def: ns = filter ((≤) i) (rremdups
    (List.map_filter (case_sum Map.empty Some) (fo_nmlz_rec i m AD xs)))
    using 3(4) Some j_lt_i
    by (auto simp: List.map_filter_simps filter_rremdups) (metis leD)
  show ?thesis
    by (rule 3(2)[OF Some 3(3) ns_def[symmetric]])
qed
qed (auto simp: List.map_filter_simps)

definition id_map :: nat ⇒ ('a + nat → nat) where
  id_map n = (λx. case x of Inl x ⇒ None | Inr x ⇒ if x < n then Some x else None)

lemma fo_nmlz_rec_idem: Inl -' set ys ⊆ AD ⇒
  rremdups (List.map_filter (case_sum Map.empty Some) ys) = ns ⇒
  set (filter (λn. n < i) ns) ⊆ {..} ⇒ filter ((≤) i) ns = [i..proof (induction ys arbitrary: i k ns)
  case (Cons y ys)
  show ?case
  proof (cases y)
  case (Inl a)
  show ?thesis
    using Cons(1)[OF _ _ Cons(4,5)] Cons(2,3)
    by (auto simp: Inl List.map_filter_simps)
  next
  case (Inr j)
  show ?thesis
  proof (cases j < i)
  case False
  have j_i: j = i
    using False Cons(3,5)
    by (auto simp: Inr List.map_filter_simps filter_rremdups in_mono split: if_splits)
    (metis (no_types, lifting) upt_eq_Cons_conv)
  obtain kk where k_def: k = Suc kk
    using Cons(3,5)
    by (cases k) (auto simp: Inr List.map_filter_simps j_i)
  define ns' where ns' = rremdups (List.map_filter (case_sum Map.empty Some) ys)
  have id_map_None: id_map i (Inr i) = None
    by (auto simp: id_map_def)
  have id_map_upd: (id_map i)(Inr i ↦ i) = id_map (Suc i)
    by (auto simp: id_map_def split: sum.splits)
  have set (filter (λn. n < Suc i) ns') ⊆ {..}
    using Cons(2,3)
    by auto
  moreover have filter ((≤) (Suc i)) ns' = [Suc i..

```

```

    using Cons(1)[OF _ ns'_def[symmetric], of Suc i kk]
    by (auto simp: ns'_def k_def id_map_upd split: if_splits)
  then show ?thesis
    by (auto simp: Inr j_i id_map_None)
next
case True
define ns' where ns' = rremdups (List.map_filter (case_sum Map.empty Some) ys)
have set (filter (λy. y < i) ns') ⊆ set (filter (λy. y < i) ns)
  filter ((≤) i) ns' = filter ((≤) i) ns
using Cons(3) True
by (auto simp: Inr List.map_filter_simps filter_rremdups[symmetric] ns'_def[symmetric])
  (smt filter_cong leD)
then have fo_nmlz_rec i (id_map i) AD ys = ys
  using Cons(1)[OF _ ns'_def[symmetric]] Cons(3,5) Cons(2)
  by (auto simp: vimage_def)
then show ?thesis
  using True
  by (auto simp: Inr id_map_def)
qed
qed
qed (auto simp: List.map_filter_simps intro!: exI[of _ []])

lemma fo_nmlz_rec_length: length (fo_nmlz_rec i m AD xs) = length xs
  by (induction i m AD xs rule: fo_nmlz_rec.induct) (auto simp: fun_upd_def split: option_splits)

lemma insert_Inr: ∧X. insert (Inr i) (X ∪ Inr ' {..<i}) = X ∪ Inr ' {..<Suc i}
  by auto

lemma fo_nmlz_rec_set: ran m ⊆ {..<i} ⇒ set (fo_nmlz_rec i m AD xs) ∪ Inr ' {..<i} =
  set xs ∩ Inl ' AD ∪ Inr ' {..<i + card (set xs - Inl ' AD - dom m)}
proof (induction i m AD xs rule: fo_nmlz_rec.induct)
case (2 i m AD x xs)
have fin: finite (set (Inl x # xs) - Inl ' AD - dom m)
  by auto
show ?case
  using 2(1)[OF _ 2(4)]
proof (cases x ∈ AD)
case True
have card (set (Inl x # xs) - Inl ' AD - dom m) = card (set xs - Inl ' AD - dom m)
  using True
  by auto
then show ?thesis
  using 2(1)[OF True 2(4)] True
  by auto
next
case False
show ?thesis
proof (cases m (Inl x))
case None
have pred: ran (m(Inl x ↦ i)) ⊆ {..<Suc i}
  using 2(4) None
  by (auto simp: inj_on_def dom_def ran_def)
have set (Inl x # xs) - Inl ' AD - dom m =
  {Inl x} ∪ (set xs - Inl ' AD - dom (m(Inl x ↦ i)))
  using None False
  by (auto simp: dom_def)
then have Suc: Suc i + card (set xs - Inl ' AD - dom (m(Inl x ↦ i))) =
  i + card (set (Inl x # xs) - Inl ' AD - dom m)

```

```

    using None
    by auto
  show ?thesis
    using 2(2)[OF False None pred] False None
    unfolding Suc
    by (auto simp: fun_upd_def[symmetric] insert_Inr)
next
case (Some j)
then have j_lt_i: j < i
  using 2(4)
  by (auto simp: ran_def)
have card (set (Inl x # xs) - Inl ' AD - dom m) = card (set xs - Inl ' AD - dom m)
  by (auto simp: Some intro: arg_cong[of _ _ card])
then show ?thesis
  using 2(3)[OF False Some 2(4)] False Some j_lt_i
  by auto
qed
qed
next
case (3 i m AD k xs)
then show ?case
proof (cases m (Inr k))
case None
have preds: ran (m(Inr k ↦ i)) ⊆ {..

```

```

(∀ n. m n ≠ None → m' n = m n) ∧ (∀ (x, y) ∈ set (zip xs (fo_nmlz_rec i m AD xs)).
  (case x of Inl x' ⇒ if x' ∈ AD then x = y else ∃ j. m' (Inl x') = Some j ∧ y = Inr j
  | Inr n ⇒ ∃ j. m' (Inr n) = Some j ∧ y = Inr j))
proof (induction i m AD xs rule: fo_nmlz_rec.induct)
case (2 i m AD x xs)
show ?case
  using 2(1)[OF_ 2(4,5)]
proof (cases x ∈ AD)
  case False
  show ?thesis
  proof (cases m (Inl x))
  case None
  have preds: inj_on (m(Inl x ↦ i)) (dom (m(Inl x ↦ i))) ran (m(Inl x ↦ i)) ⊆ {..<Suc i}
    using 2(4,5)
    by (auto simp: inj_on_def ran_def)
  show ?thesis
    using 2(2)[OF False None preds] False None
    apply safe
    subgoal for m'
      by (auto simp: fun_upd_def split: sum.splits intro!: exI[of _ m'])
    done
  next
  case (Some j)
  show ?thesis
    using 2(3)[OF False Some 2(4,5)] False Some
    apply safe
    subgoal for m'
      by (auto split: sum.splits intro!: exI[of _ m'])
    done
  qed
qed auto
next
case (3 i m AD n xs)
show ?case
proof (cases m (Inr n))
  case None
  have preds: inj_on (m(Inr n ↦ i)) (dom (m(Inr n ↦ i))) ran (m(Inr n ↦ i)) ⊆ {..<Suc i}
    using 3(3,4)
    by (auto simp: inj_on_def ran_def)
  show ?thesis
    using 3(1)[OF None preds] None
    apply safe
    subgoal for m'
      by (auto simp: fun_upd_def intro!: exI[of _ m'] split: sum.splits)
    done
  next
  case (Some j)
  show ?thesis
    using 3(2)[OF Some 3(3,4)] Some
    apply safe
    subgoal for m'
      by (auto simp: fun_upd_def intro!: exI[of _ m'] split: sum.splits)
    done
  qed
qed auto

lemma adAgr_map:
  assumes length xs = length ys inj_on m (dom m)

```

$\bigwedge x y. (x, y) \in \text{set } (\text{zip } xs \ ys) \implies (\text{case } x \text{ of } \text{Inl } x' \implies$   
 $\text{if } x' \in AD \text{ then } x = y \text{ else } m \ x = \text{Some } y \wedge (\text{case } y \text{ of } \text{Inl } z \implies z \notin AD \mid \text{Inr } \_ \implies \text{True}))$   
 $\mid \text{Inr } n \implies m \ x = \text{Some } y \wedge (\text{case } y \text{ of } \text{Inl } z \implies z \notin AD \mid \text{Inr } \_ \implies \text{True}))$   
**shows**  $\text{ad\_agr\_list } AD \ xs \ ys$

**proof** –

**have**  $\text{ad\_equiv\_pair } AD \ (a, b) \text{ if } (a, b) \in \text{set } (\text{zip } xs \ ys) \text{ for } a \ b$   
**unfolding**  $\text{ad\_equiv\_pair.simps}$   
**using**  $\text{assms}(3)[\text{OF that}]$   
**by**  $(\text{auto split: sum.splits if\_splits})$   
**moreover have**  $\text{False if } (a, c) \in \text{set } (\text{zip } xs \ ys) \ (b, c) \in \text{set } (\text{zip } xs \ ys) \ a \neq b \text{ for } a \ b \ c$   
**using**  $\text{assms}(3)[\text{OF that}(1)] \ \text{assms}(3)[\text{OF that}(2)] \ \text{assms}(2) \ \text{that}(3)$   
**by**  $(\text{auto split: sum.splits if\_splits}) \ (\text{metis domI inj\_onD that}(3))+$   
**moreover have**  $\text{False if } (a, b) \in \text{set } (\text{zip } xs \ ys) \ (a, c) \in \text{set } (\text{zip } xs \ ys) \ b \neq c \text{ for } a \ b \ c$   
**using**  $\text{assms}(3)[\text{OF that}(1)] \ \text{assms}(3)[\text{OF that}(2)] \ \text{assms}(2) \ \text{that}(3)$   
**by**  $(\text{auto split: sum.splits if\_splits})$   
**ultimately show**  $\text{?thesis}$   
**using**  $\text{assms}$   
**by**  $(\text{fastforce simp: ad\_agr\_list\_def ad\_equiv\_list\_def sp\_equiv\_list\_def pairwise\_def})$

**qed**

**lemma**  $\text{fo\_nmlz\_rec\_take: take } n \ (\text{fo\_nmlz\_rec } i \ m \ AD \ xs) = \text{fo\_nmlz\_rec } i \ m \ AD \ (\text{take } n \ xs)$   
**by**  $(\text{induction } i \ m \ AD \ xs \ \text{arbitrary: } n \ \text{rule: fo\_nmlz\_rec.induct})$   
 $(\text{auto simp: take\_Cons' split: option.splits})$

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

**lemma**  $\text{fo\_nmlz\_Nil[simp]: fo\_nmlz } AD \ [] = []$   
**by**  $(\text{auto simp: fo\_nmlz\_def})$

**lemma**  $\text{fo\_nmlz\_Cons: fo\_nmlz } AD \ [x] =$   
 $(\text{case } x \text{ of } \text{Inl } x \implies \text{if } x \in AD \ \text{then } [\text{Inl } x] \ \text{else } [\text{Inr } 0] \mid \_ \implies [\text{Inr } 0])$   
**by**  $(\text{auto simp: fo\_nmlz\_def split: sum.splits})$

**lemma**  $\text{fo\_nmlz\_Cons\_Cons: fo\_nmlz } AD \ [x, x] =$   
 $(\text{case } x \text{ of } \text{Inl } x \implies \text{if } x \in AD \ \text{then } [\text{Inl } x, \text{Inl } x] \ \text{else } [\text{Inr } 0, \text{Inr } 0] \mid \_ \implies [\text{Inr } 0, \text{Inr } 0])$   
**by**  $(\text{auto simp: fo\_nmlz\_def split: sum.splits})$

**lemma**  $\text{fo\_nmlz\_sound: fo\_nmlzd } AD \ (\text{fo\_nmlz } AD \ xs)$   
**using**  $\text{fo\_nmlz\_rec\_sound[of Map.empty } 0] \ \text{fo\_nmlz\_rec\_set[of Map.empty } 0 \ AD \ xs]$   
**by**  $(\text{auto simp: fo\_nmlzd\_def fo\_nmlz\_def nats\_def Let\_def})$

**lemma**  $\text{fo\_nmlz\_length: length } (\text{fo\_nmlz } AD \ xs) = \text{length } xs$   
**using**  $\text{fo\_nmlz\_rec\_length}$   
**by**  $(\text{auto simp: fo\_nmlz\_def})$

**lemma**  $\text{fo\_nmlz\_map: } \exists \tau. \text{fo\_nmlz } AD \ (\text{map } \sigma \ ns) = \text{map } \tau \ ns$

**proof** –

**obtain**  $m' \text{ where } m'\_def: \forall (x, y) \in \text{set } (\text{zip } (\text{map } \sigma \ ns) \ (\text{fo\_nmlz } AD \ (\text{map } \sigma \ ns))).$   
 $\text{case } x \text{ of } \text{Inl } x' \implies \text{if } x' \in AD \ \text{then } x = y \ \text{else } \exists j. m' \ (\text{Inl } x') = \text{Some } j \wedge y = \text{Inr } j$   
 $\mid \text{Inr } n \implies \exists j. m' \ (\text{Inr } n) = \text{Some } j \wedge y = \text{Inr } j$   
**using**  $\text{fo\_nmlz\_rec\_map[of Map.empty } 0, \text{ of map } \sigma \ ns]$   
**by**  $(\text{auto simp: fo\_nmlz\_def})$   
**define**  $\tau \text{ where } \tau \equiv (\lambda n. \text{case } \sigma \ n \ \text{of } \text{Inl } x \implies \text{if } x \in AD \ \text{then } \text{Inl } x \ \text{else } \text{Inr } (\text{the } (m' \ (\text{Inl } x))))$   
 $\mid \text{Inr } j \implies \text{Inr } (\text{the } (m' \ (\text{Inr } j))))$   
**have**  $\text{fo\_nmlz } AD \ (\text{map } \sigma \ ns) = \text{map } \tau \ ns$   
**proof**  $(\text{rule nth\_equalityI})$   
**show**  $\text{length } (\text{fo\_nmlz } AD \ (\text{map } \sigma \ ns)) = \text{length } (\text{map } \tau \ ns)$

```

    using fo_nmlz_length[of AD map  $\sigma$  ns]
    by auto
  fix i
  assume i < length (fo_nmlz AD (map  $\sigma$  ns))
  then show fo_nmlz AD (map  $\sigma$  ns) ! i = map  $\tau$  ns ! i
    using m'_def fo_nmlz_length[of AD map  $\sigma$  ns]
    apply (auto simp: set_zip  $\tau$ _def split: sum.splits)
    apply (metis nth_map)
    apply (metis nth_map option.sel)+
  done
qed
then show ?thesis
  by auto
qed

lemma card_set_minus: card (set xs - X)  $\leq$  length xs
  by (meson Diff_subset List.finite_set card_length card_mono order_trans)

lemma fo_nmlz_set: set (fo_nmlz AD xs) =
  set xs  $\cap$  Inl ' AD  $\cup$  Inr ' {.. $\min$  (length xs) (card (set xs - Inl ' AD))}
  using fo_nmlz_rec_set[of Map.empty 0 AD xs]
  by (auto simp add: fo_nmlz_def card_set_minus)

lemma fo_nmlz_set_rev: set (fo_nmlz AD xs)  $\subseteq$  Inl ' AD  $\implies$  set xs  $\subseteq$  Inl ' AD
  using fo_nmlz_rec_set_rev[of 0 Map.empty AD xs]
  by (auto simp: fo_nmlz_def)

lemma inj_on_empty: inj_on Map.empty (dom Map.empty) and ran_empty_upto: ran Map.empty  $\subseteq$ 
  {.. $0$ }
  by auto

lemma fo_nmlz_ad_agr: ad_agr_list AD xs (fo_nmlz AD xs)
  using fo_nmlz_rec_map[OF inj_on_empty ran_empty_upto, of xs AD]
  unfolding fo_nmlz_def
  apply safe
  subgoal for m'
    by (fastforce simp: inj_on_def dom_def split: sum.splits if_splits
      intro!: ad_agr_map[OF fo_nmlz_rec_length[symmetric], of map_option Inr  $\circ$  m'])
  done

lemma fo_nmlzd_mono: Inl - ' set xs  $\subseteq$  AD  $\implies$  fo_nmlzd AD' xs  $\implies$  fo_nmlzd AD xs
  by (auto simp: fo_nmlzd_def)

lemma fo_nmlz_idem: fo_nmlzd AD ys  $\implies$  fo_nmlz AD ys = ys
  using fo_nmlz_rec_idem[where ?i=0]
  by (auto simp: fo_nmlzd_def fo_nmlz_def id_map_def nats_def Let_def)

lemma fo_nmlz_take: take n (fo_nmlz AD xs) = fo_nmlz AD (take n xs)
  using fo_nmlz_rec_take
  by (auto simp: fo_nmlz_def)

fun nall_tuples_rec :: 'a set  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  ('a + nat) table where
  nall_tuples_rec AD i 0 = {[]}
| nall_tuples_rec AD i (Suc n) =  $\bigcup$  (( $\lambda$ as. ( $\lambda$ x. x # as) ' (Inl ' AD  $\cup$  Inr ' {.. $i$ )) '
  nall_tuples_rec AD i n)  $\cup$  ( $\lambda$ as. Inr i # as) ' nall_tuples_rec AD (Suc i) n

lemma nall_tuples_rec_Inl: vs  $\in$  nall_tuples_rec AD i n  $\implies$  Inl - ' set vs  $\subseteq$  AD
  by (induction AD i n arbitrary: vs rule: nall_tuples_rec.induct) (fastforce simp: vimage_def)+

```



```

lemma nall_tuples_rec_length:  $xs \in \text{nall\_tuples\_rec } AD \ i \ n \implies \text{length } xs = n$ 
  by (induction  $AD \ i \ n$  arbitrary:  $xs$  rule:  $\text{nall\_tuples\_rec.induct}$ ) auto

lemma fun_upd_id_map:  $(\text{id\_map } i)(\text{Inr } i \mapsto i) = \text{id\_map } (\text{Suc } i)$ 
  by (rule ext) (auto simp:  $\text{id\_map\_def}$  split:  $\text{sum.splits}$ )

lemma id_mapD:  $\text{id\_map } j \ (\text{Inr } i) = \text{None} \implies j \leq i$ 
 $\text{id\_map } j \ (\text{Inr } i) = \text{Some } x \implies i < j \wedge i = x$ 
  by (auto simp:  $\text{id\_map\_def}$  split:  $\text{if\_splits}$ )

lemma nall_tuples_rec_fo_nmlz_rec_sound:  $i \leq j \implies xs \in \text{nall\_tuples\_rec } AD \ i \ n \implies$ 
 $\text{fo\_nmlz\_rec } j \ (\text{id\_map } j) \ AD \ xs = xs$ 
  apply (induction  $n$  arbitrary:  $i \ j \ xs$ )
  apply (auto simp:  $\text{fun\_upd\_id\_map}$  dest!:  $\text{id\_mapD}$  split:  $\text{option.splits}$ )
  apply (meson  $\text{dual\_order.strict\_trans2}$   $\text{id\_mapD}(1)$   $\text{not\_Some\_eq}$   $\text{sup.strict\_order\_iff}$ )
  using  $\text{Suc\_leI}$  apply blast+
  done

lemma nall_tuples_rec_fo_nmlz_rec_complete:
  assumes  $\text{fo\_nmlz\_rec } j \ (\text{id\_map } j) \ AD \ xs = xs$ 
  shows  $xs \in \text{nall\_tuples\_rec } AD \ j \ (\text{length } xs)$ 
  using assms
proof (induction  $xs$  arbitrary:  $j$ )
  case ( $\text{Cons } x \ xs$ )
  show ?case
  proof (cases  $x$ )
  case ( $\text{Inl } a$ )
  have  $a\_AD$ :  $a \in AD$ 
  using  $\text{Cons}(2)$ 
  by (auto simp:  $\text{Inl}$  split:  $\text{if\_splits}$   $\text{option.splits}$ )
  show ?thesis
  using  $\text{Cons } a\_AD$ 
  by (auto simp:  $\text{Inl}$ )
  next
  case ( $\text{Inr } b$ )
  have  $b\_j$ :  $b \leq j$ 
  using  $\text{Cons}(2)$ 
  by (auto simp:  $\text{Inr}$  split:  $\text{option.splits}$  dest:  $\text{id\_mapD}$ )
  show ?thesis
  proof (cases  $b = j$ )
  case  $\text{True}$ 
  have  $\text{preds}$ :  $\text{fo\_nmlz\_rec } (\text{Suc } j) \ (\text{id\_map } (\text{Suc } j)) \ AD \ xs = xs$ 
  using  $\text{Cons}(2)$ 
  by (auto simp:  $\text{Inr True}$   $\text{fun\_upd\_id\_map}$  dest:  $\text{id\_mapD}$  split:  $\text{option.splits}$ )
  show ?thesis
  using  $\text{Cons}(1)[\text{OF } \text{preds}]$ 
  by (auto simp:  $\text{Inr True}$ )
  next
  case  $\text{False}$ 
  have  $b\_lt\_j$ :  $b < j$ 
  using  $b\_j$   $\text{False}$ 
  by auto
  have  $\text{id\_map}$ :  $\text{id\_map } j \ (\text{Inr } b) = \text{Some } b$ 
  using  $b\_lt\_j$ 
  by (auto simp:  $\text{id\_map\_def}$ )
  have  $\text{preds}$ :  $\text{fo\_nmlz\_rec } j \ (\text{id\_map } j) \ AD \ xs = xs$ 
  using  $\text{Cons}(2)$ 
  by (auto simp:  $\text{Inr id\_map}$ )

```

```

show ?thesis
  using Cons(1)[OF preds] b_lt_j
  by (auto simp: Inr)
qed
qed
qed auto

lemma nall_tuples_rec_fo_nmlz:  $xs \in \text{nall\_tuples\_rec AD } 0 \text{ (length } xs) \longleftrightarrow \text{fo\_nmlz AD } xs = xs$ 
  using nall_tuples_rec_fo_nmlz_rec_sound[of 0 0 xs AD length xs]
  nall_tuples_rec_fo_nmlz_rec_complete[of 0 AD xs]
  by (auto simp: fo_nmlz_def id_map_def)

lemma fo_nmlzd_code[code]:  $\text{fo\_nmlzd AD } xs \longleftrightarrow \text{fo\_nmlz AD } xs = xs$ 
  using fo_nmlz_idem fo_nmlz_sound
  by metis

lemma nall_tuples_code[code]:  $\text{nall\_tuples AD } n = \text{nall\_tuples\_rec AD } 0 \ n$ 
  unfolding nall_tuples_set
  using nall_tuples_rec_length trans[OF nall_tuples_rec_fo_nmlz fo_nmlzd_code[symmetric]]
  by fastforce

lemma exists_map:  $\text{length } xs = \text{length } ys \implies \text{distinct } xs \implies \exists f. ys = \text{map } f \ xs$ 
proof (induction xs ys rule: list_induct2)
  case (Cons x xs y ys)
  then obtain f where f_def:  $ys = \text{map } f \ xs$ 
  by auto
  with Cons(3) have  $y \# ys = \text{map } (f(x := y)) \ (x \# xs)$ 
  by auto
  then show ?case
  by metis
qed auto

lemma exists_fo_nmlzd:
  assumes  $\text{length } xs = \text{length } ys \ \text{distinct } xs \ \text{fo\_nmlzd AD } ys$ 
  shows  $\exists f. ys = \text{fo\_nmlz AD } (\text{map } f \ xs)$ 
  using fo_nmlz_idem[OF assms(3)] exists_map[OF _ assms(2)] assms(1)
  by metis

lemma list_induct2_rev[consumes 1]:  $\text{length } xs = \text{length } ys \implies (P \ [] \ []) \implies$ 
 $(\bigwedge x \ y \ xs \ ys. P \ xs \ ys \implies P \ (xs \ @ \ [x]) \ (ys \ @ \ [y])) \implies P \ xs \ ys$ 
proof (induction length xs arbitrary: xs ys)
  case (Suc n)
  then show ?case
  by (cases xs rule: rev_cases; cases ys rule: rev_cases) auto
qed auto

lemma adAgr_list_fo_nmlzd:
  assumes  $\text{ad\_Agr\_list AD } vs \ vs' \ \text{fo\_nmlzd AD } vs \ \text{fo\_nmlzd AD } vs'$ 
  shows  $vs = vs'$ 
  using adAgr_list_length[OF assms(1)] assms
proof (induction vs vs' rule: list_induct2_rev)
  case (2 x y xs ys)
  have norms:  $\text{fo\_nmlzd AD } xs \ \text{fo\_nmlzd AD } ys$ 
  using 2(3,4)
  by (auto simp: fo_nmlzd_def nats_def Let_def map_filter_app rremdups_app
    split: sum.splits if_splits)
  have adAgr:  $\text{ad\_Agr\_list AD } xs \ ys$ 
  using 2(2)

```

```

    by (auto simp: ad_agr_list_def ad_equiv_list_def sp_equiv_list_def pairwise_def)
  note xs_ys = 2(1)[OF ad_agr_norms]
  have x = y
  proof (cases isl x  $\vee$  isl y)
    case True
    then have isl x  $\longrightarrow$  proj1 x  $\in$  AD isl y  $\longrightarrow$  proj1 y  $\in$  AD
      using 2(3,4)
      by (auto simp: fo_nmlzd_def)
    then show ?thesis
      using 2(2) True
      apply (auto simp: ad_agr_list_def ad_equiv_list_def isl_def)
      unfolding ad_equiv_pair_simps
      by blast+
  next
  case False
  then obtain x' y' where inr: x = Inr x' y = Inr y'
    by (cases x; cases y) auto
  show ?thesis
    using 2(2) xs_ys
  proof (cases x  $\in$  set xs  $\vee$  y  $\in$  set ys)
    case False
    then show ?thesis
      using fo_nmlzd_app_Inr 2(3,4)
      unfolding inr xs_ys
      by auto
  qed (auto simp: ad_agr_list_def sp_equiv_list_def pairwise_def set_zip in_set_conv_nth)
  qed
  then show ?case
    using xs_ys
    by auto
  qed auto

lemma fo_nmlz_eqI:
  assumes ad_agr_list AD vs vs'
  shows fo_nmlz AD vs = fo_nmlz AD vs'
  using ad_agr_list_fo_nmlzd[OF
    ad_agr_list_trans[OF ad_agr_list_trans[OF
      ad_agr_list_comm[OF fo_nmlz_ad_agr[of AD vs]] assms]
    fo_nmlz_ad_agr[of AD vs']] fo_nmlz_sound fo_nmlz_sound] .

lemma fo_nmlz_eqD:
  assumes fo_nmlz AD vs = fo_nmlz AD vs'
  shows ad_agr_list AD vs vs'
  using ad_agr_list_trans[OF fo_nmlz_ad_agr[of AD vs, unfolded assms]
    ad_agr_list_comm[OF fo_nmlz_ad_agr[of AD vs']]] .

lemma fo_nmlz_eq: fo_nmlz AD vs = fo_nmlz AD vs'  $\longleftrightarrow$  ad_agr_list AD vs vs'
  using fo_nmlz_eqI[where ?AD=AD] fo_nmlz_eqD[where ?AD=AD]
  by blast

lemma fo_nmlz_mono:
  assumes AD  $\subseteq$  AD' Inl -' set xs  $\subseteq$  AD
  shows fo_nmlz AD' xs = fo_nmlz AD xs
  proof -
  have fo_nmlz AD (fo_nmlz AD' xs) = fo_nmlz AD' xs
    apply (rule fo_nmlz_idem[OF fo_nmlzd_mono[OF _ fo_nmlz_sound]])
    using assms
    by (auto simp: fo_nmlz_set)
  
```

**moreover have**  $fo\_nmlz\ AD\ xs = fo\_nmlz\ AD\ (fo\_nmlz\ AD'\ xs)$   
**apply** (rule  $fo\_nmlz\_eqI$ )  
**apply** (rule  $ad\_agr\_list\_mono[OF\ assms(1)]$ )  
**apply** (rule  $fo\_nmlz\_ad\_agr$ )  
**done**  
**ultimately show**  $?thesis$   
**by auto**  
**qed**

**definition**  $proj\_vals :: 'c\ val\ set \Rightarrow nat\ list \Rightarrow 'c\ table$  **where**  
 $proj\_vals\ R\ ns = (\lambda\tau.\ map\ \tau\ ns)$  ‘  $R$

**definition**  $proj\_fmla :: ('a,\ 'b)\ fo\_fmla \Rightarrow 'c\ val\ set \Rightarrow 'c\ table$  **where**  
 $proj\_fmla\ \varphi\ R = proj\_vals\ R\ (fv\_fo\_fmla\_list\ \varphi)$

**lemmas**  $proj\_fmla\_map = proj\_fmla\_def[unfolded\ proj\_vals\_def]$

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

**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**  $ext\_tuple\_eq:\ length\ fv\_sub = length\ as \Longrightarrow$   
 $ext\_tuple\ AD\ fv\_sub\ fv\_sub\_comp\ as =$   
 $(\lambda fs.\ map\ snd\ (merge\ (zip\ fv\_sub\ as)\ (zip\ fv\_sub\_comp\ fs))))$  ‘  
 $(nall\_tuples\_rec\ AD\ (card\ (Inr\ -' set\ as))\ (length\ fv\_sub\_comp))$   
**using**  $fo\_nmlz\_idem[of\ AD\ as]$   
**by** (auto simp:  $ext\_tuple\_def$ )

**lemma**  $map\_map\_of:\ length\ xs = length\ ys \Longrightarrow distinct\ xs \Longrightarrow$   
 $ys = map\ (the\ \circ\ (map\_of\ (zip\ xs\ ys)))\ xs$   
**by** (induction  $xs\ ys$  rule:  $list\_induct2$ ) (auto simp:  $fun\_upd\_comp$ )

**lemma**  $id\_map\_empty:\ id\_map\ 0 = Map.empty$   
**by** (rule  $ext$ ) (auto simp:  $id\_map\_def\ split:\ sum.splits$ )

**lemma**  $fo\_nmlz\_rec\_shift:$   
**fixes**  $xs :: ('a + nat)\ list$   
**shows**  $fo\_nmlz\_rec\ i\ (id\_map\ i)\ AD\ xs = xs \Longrightarrow$   
 $i' = card\ (Inr\ -' (\{..\lt i\} \cup set\ (take\ n\ xs))) \Longrightarrow n \leq length\ xs \Longrightarrow$   
 $fo\_nmlz\_rec\ i'\ (id\_map\ i')\ AD\ (drop\ n\ xs) = drop\ n\ xs$   
**proof** (induction  $i\ id\_map\ i :: 'a + nat \rightarrow nat\ AD\ xs$  arbitrary:  $n$  rule:  $fo\_nmlz\_rec.induct$ )  
**case** ( $2\ i\ AD\ x\ xs$ )  
**have**  $preds:\ x \in AD\ fo\_nmlz\_rec\ i\ (id\_map\ i)\ AD\ xs = xs$   
**using**  $2(4)$   
**by** (auto split:  $if\_splits\ option.splits$ )  
**show**  $?case$   
**using**  $2(4,5)$   
**proof** (cases  $n$ )  
**case** ( $Suc\ k$ )  
**have**  $k\_le:\ k \leq length\ xs$   
**using**  $2(6)$   
**by** (auto simp:  $Suc$ )  
**have**  $i'\_def:\ i' = card\ (Inr\ -' (\{..\lt i\} \cup set\ (take\ k\ xs)))$   
**using**  $2(5)$

```

    by (auto simp: Suc vimage_def)
  show ?thesis
    using 2(1)[OF preds i'_def k_le]
    by (auto simp: Suc)
qed (auto simp: inj_vimage_image_eq)
next
case (3 i AD j xs)
show ?case
using 3(3,4)
proof (cases n)
case (Suc k)
have k_le: k ≤ length xs
  using 3(5)
  by (auto simp: Suc)
have j_le_i: j ≤ i
  using 3(3)
  by (auto split: option.splits dest: id_mapD)
show ?thesis
proof (cases j = i)
case True
have id_map: id_map i (Inr j) = None (id_map i)(Inr j ↦ i) = id_map (Suc i)
  unfolding True fun_upd_id_map
  by (auto simp: id_map_def)
have norm_xs: fo_nmlz_rec (Suc i) (id_map (Suc i)) AD xs = xs
  using 3(3)
  by (auto simp: id_map split: option.splits dest: id_mapD)
have i'_def: i' = card (Inr -' (Inr ' {..

```

```

fun proj_tuple :: nat list ⇒ (nat × ('a + nat)) list ⇒ ('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*:  $\text{proj\_tuple } (\text{map fst } nxs) \ nxs = \text{map snd } nxs$   
**by** (*induction nxs*) *auto*

**lemma** *proj\_tuple\_merge*:  $\text{sorted\_distinct } (\text{map fst } nxs) \implies \text{sorted\_distinct } (\text{map fst } mys) \implies$   
 $\text{set } (\text{map fst } nxs) \cap \text{set } (\text{map fst } mys) = \{\}$   $\implies$   
 $\text{proj\_tuple } (\text{map fst } nxs) \ (\text{merge } nxs \ mys) = \text{map snd } nxs$   
**using** *proj\_tuple\_idle*  
**by** (*induction nxs mys rule: merge.induct*) *auto+*

**lemma** *proj\_tuple\_map*:

**assumes**  $\text{sorted\_distinct } ns \ \text{sorted\_distinct } ms \ \text{set } ns \subseteq \text{set } ms$   
**shows**  $\text{proj\_tuple } ns \ (\text{zip } ms \ (\text{map } \sigma \ ms)) = \text{map } \sigma \ ns$

**proof** –

**define** *ns'* **where**  $ns' = \text{filter } (\lambda n. n \notin \text{set } ns) \ ms$

**have** *sd\_ns'*:  $\text{sorted\_distinct } ns'$   
**using** *assms(2)* *sorted\_filter[of id]*  
**by** (*auto simp: ns'\_def*)

**have** *disj*:  $\text{set } ns \cap \text{set } ns' = \{\}$   
**by** (*auto simp: ns'\_def*)

**have** *ms\_def*:  $ms = \text{sort } (ns \ @ \ ns')$   
**apply** (*rule sorted\_distinct\_set\_unique*)  
**using** *assms*  
**by** (*auto simp: ns'\_def*)

**have** *zip*:  $\text{zip } ms \ (\text{map } \sigma \ ms) = \text{merge } (\text{zip } ns \ (\text{map } \sigma \ ns)) \ (\text{zip } ns' \ (\text{map } \sigma \ ns'))$   
**unfolding** *merge\_map[OF assms(1) sd\_ns' disj, folded ms\_def, symmetric]*  
**using** *map\_fst\_merge assms(1)*

**by** (*auto simp: ms\_def*) (*smt length\_map map\_fst\_merge map\_fst\_zip sd\_ns' zip\_map\_fst\_snd*)

**show** *?thesis*

**unfolding** *zip*  
**using** *proj\_tuple\_merge*  
**by** (*smt assms(1) disj length\_map map\_fst\_zip map\_snd\_zip sd\_ns'*)

**qed**

**lemma** *proj\_tuple\_length*:

**assumes**  $\text{sorted\_distinct } ns \ \text{sorted\_distinct } ms \ \text{set } ns \subseteq \text{set } ms \ \text{length } ms = \text{length } ns$   
**shows**  $\text{length } (\text{proj\_tuple } ns \ (\text{zip } ms \ ns)) = \text{length } ns$

**proof** –

**obtain**  $\sigma$  **where**  $\sigma: ns = \text{map } \sigma \ ms$   
**using** *exists\_map[OF assms(4)] assms(2)*  
**by** *auto*

**show** *?thesis*

**unfolding**  $\sigma$   
**by** (*auto simp: proj\_tuple\_map[OF assms(1–3)]*)

**qed**

**lemma** *ext\_tuple\_sound*:

**assumes**  $\text{sorted\_distinct } fv\_sub \ \text{sorted\_distinct } fv\_sub\_comp \ \text{sorted\_distinct } fv\_all$   
 $\text{set } fv\_sub \cap \text{set } fv\_sub\_comp = \{\}$   $\text{set } fv\_sub \cup \text{set } fv\_sub\_comp = \text{set } fv\_all$   
 $ass = fo\_nmlz \ AD \ ' \ \text{proj\_vals } R \ fv\_sub$

$\bigwedge \sigma \ \tau. \ \text{ad\_agr\_sets } (\text{set } fv\_sub) \ (\text{set } fv\_sub) \ AD \ \sigma \ \tau \implies \sigma \in R \longleftrightarrow \tau \in R$   
 $xs \in fo\_nmlz \ AD \ ' \ \bigcup (\text{ext\_tuple } AD \ fv\_sub \ fv\_sub\_comp \ ' \ ass)$

**shows**  $fo\_nmlz \ AD \ (\text{proj\_tuple } fv\_sub \ (\text{zip } fv\_all \ xs)) \in ass$   
 $xs \in fo\_nmlz \ AD \ ' \ \text{proj\_vals } R \ fv\_all$

**proof** –

**have** *fv\_all\_sort*:  $fv\_all = \text{sort } (fv\_sub \ @ \ fv\_sub\_comp)$   
**using** *assms(1,2,3,4,5)*  
**by** (*simp add: sorted\_distinct\_set\_unique*)

```

have len_in_ass:  $\bigwedge xs. xs \in \text{ass} \implies xs = \text{fo\_nmlz AD } xs \wedge \text{length } xs = \text{length } \text{fv\_sub}$ 
  by (auto simp: assms(6) proj_vals_def fo_nmlz_length fo_nmlz_idem fo_nmlz_sound)
obtain as fs where as_fs_def:  $as \in \text{ass}$ 
  fs  $\in \text{nall\_tuples\_rec AD (card (Inr - ' set as)) (length } \text{fv\_sub\_comp})$ 
  xs = fo_nmlz AD (map snd (merge (zip fv_sub as) (zip fv_sub_comp fs)))
  using fo_nmlz_sound len_in_ass assms(8)
  by (auto simp: ext_tuple_def split: if_splits)
then have vs_norm: fo_nmlzd AD xs
  using fo_nmlz_sound
  by auto
obtain  $\sigma$  where  $\sigma\_def: \sigma \in R$  as = fo_nmlz AD (map  $\sigma$  fv_sub)
  using as_fs_def(1) assms(6)
  by (auto simp: proj_vals_def)
then obtain  $\tau$  where  $\tau\_def: as = \text{map } \tau \text{ fv\_sub ad\_agr\_list AD (map } \sigma \text{ fv\_sub) (map } \tau \text{ fv\_sub)}$ 
  using fo_nmlz_map fo_nmlz_ad_agr
  by metis
have  $\tau\_R: \tau \in R$ 
  using assms(7) ad_agr_list_link  $\sigma\_def(1)$   $\tau\_def(2)$ 
  by fastforce
define  $\sigma'$  where  $\sigma' \equiv \lambda n. \text{if } n \in \text{set } \text{fv\_sub\_comp} \text{ then the (map\_of (zip } \text{fv\_sub\_comp } \text{fs) } n)$ 
  else  $\tau \ n$ 
then have  $\forall n \in \text{set } \text{fv\_sub}. \tau \ n = \sigma' \ n$ 
  using assms(4) by auto
then have  $\sigma'\_S: \sigma' \in R$ 
  using assms(7)  $\tau\_R$ 
  by (fastforce simp: ad_agr_sets_def sp_equiv_def pairwise_def ad_equiv_pair.simps)
have length_as:  $\text{length } as = \text{length } \text{fv\_sub}$ 
  using as_fs_def(1) assms(6)
  by (auto simp: proj_vals_def fo_nmlz_length)
have length_fs:  $\text{length } fs = \text{length } \text{fv\_sub\_comp}$ 
  using as_fs_def(2)
  by (auto simp: nall_tuples_rec_length)
have map_fv_sub:  $\text{map } \sigma' \text{ fv\_sub} = \text{map } \tau \text{ fv\_sub}$ 
  using assms(4)  $\tau\_def(2)$ 
  by (auto simp:  $\sigma'\_def$ )
have fs_map_map_of:  $fs = \text{map (the } \circ (\text{map\_of (zip } \text{fv\_sub\_comp } \text{fs})) \text{) } \text{fv\_sub\_comp}$ 
  using map_map_of length_fs assms(2)
  by metis
have fs_map:  $fs = \text{map } \sigma' \text{ fv\_sub\_comp}$ 
  using  $\sigma'\_def$  length_fs by (subst fs_map_map_of) simp
have vs_map_fv_all:  $xs = \text{fo\_nmlz AD (map } \sigma' \text{ fv\_all)}$ 
  unfolding as_fs_def(3)  $\tau\_def(1)$  map_fv_sub[symmetric] fs_map fv_all_sort
  using merge_map[OF assms(1,2,4)]
  by metis
show  $xs \in \text{fo\_nmlz AD ' proj\_vals } R \text{ fv\_all}$ 
  using  $\sigma'\_S$  vs_map_fv_all
  by (auto simp: proj_vals_def)
obtain  $\sigma''$  where  $\sigma''\_def: xs = \text{map } \sigma'' \text{ fv\_all}$ 
  using exists_map[of fv_all xs] fo_nmlz_map vs_map_fv_all
  by blast
have proj:  $\text{proj\_tuple } \text{fv\_sub} (\text{zip } \text{fv\_all } xs) = \text{map } \sigma'' \text{ fv\_sub}$ 
  using proj_tuple_map assms(1,3,5)
  unfolding  $\sigma''\_def$ 
  by blast
have  $\sigma''\_S': \text{fo\_nmlz AD (map } \sigma'' \text{ fv\_sub)} = as$ 
  using  $\sigma''\_def$  vs_map_fv_all  $\sigma\_def(2)$ 
  by (metis  $\tau\_def(2)$  ad_agr_list_subset assms(5) fo_nmlz_ad_agr fo_nmlz_eqI map_fv_sub sup_ge1)
show fo_nmlz AD (proj_tuple fv_sub (zip fv_all xs))  $\in \text{ass}$ 

```

**unfolding** *proj*  $\sigma''$   $\sigma'$  *map\_fv\_sub*  
**by** (*rule as\_fs\_def*(1))  
**qed**

**lemma** *ext\_tuple\_complete*:

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

**proof** –

**have** *fv\_all\_sort*:  $fv\_all = sort\ (fv\_sub\ @\ fv\_sub\_comp)$   
**using** *assms*(1,2,3,4,5)  
**by** (*simp add: sorted\_distinct\_set\_unique*)  
**note**  $\sigma\_def = assms(9,8)$   
**have** *vs\_norm*:  $fo\_nmlz\ AD\ xs$   
**using**  $\sigma\_def(2)$  *fo\_nmlz\_sound*  
**by** *auto*  
**define** *fs* **where**  $fs = map\ \sigma\ fv\_sub\_comp$   
**define** *as* **where**  $as = map\ \sigma\ fv\_sub$   
**define** *nos* **where**  $nos = fo\_nmlz\ AD\ (as\ @\ fs)$   
**define** *as'* **where**  $as' = take\ (length\ fv\_sub)\ nos$   
**define** *fs'* **where**  $fs' = drop\ (length\ fv\_sub)\ nos$   
**have** *length\_as'*:  $length\ as' = length\ fv\_sub$   
**by** (*auto simp: as'\_def nos\_def as\_def fo\_nmlz\_length*)  
**have** *length\_fs'*:  $length\ fs' = length\ fv\_sub\_comp$   
**by** (*auto simp: fs'\_def nos\_def as\_def fs\_def fo\_nmlz\_length*)  
**have** *len\_fv\_sub\_nos*:  $length\ fv\_sub \leq length\ nos$   
**by** (*auto simp: nos\_def fo\_nmlz\_length as\_def*)  
**have** *norm\_as'*:  $fo\_nmlz\ AD\ as'$   
**using** *fo\_nmlz\_take*[*OF fo\_nmlz\_sound*]  
**by** (*auto simp: as'\_def nos\_def*)  
**have** *as'\_norm\_as*:  $as' = fo\_nmlz\ AD\ as$   
**by** (*auto simp: as'\_def nos\_def as\_def fo\_nmlz\_take*)  
**have** *ad\_agr\_as'*:  $ad\_agr\_list\ AD\ as\ as'$   
**using** *fo\_nmlz\_ad\_agr*  
**unfolding** *as'\_norm\_as* .  
**have** *nos\_as'\_fs'*:  $nos = as'\ @\ fs'$   
**using** *length\_as' length\_fs'*  
**by** (*auto simp: as'\_def fs'\_def*)  
**obtain**  $\tau$  **where**  $\tau\_def: as' = map\ \tau\ fv\_sub\ fs' = map\ \tau\ fv\_sub\_comp$   
**using** *exists\_map*[*of fv\_sub @ fv\_sub\_comp as' @ fs'*] *assms*(1,2,4) *length\_as' length\_fs'*  
**by** *auto*  
**have**  $length\ fv\_sub + length\ fv\_sub\_comp \leq length\ fv\_all$   
**using** *assms*(1,2,3,4,5)  
**by** (*metis distinct\_append distinct\_card eq\_iff length\_append set\_append*)  
**then have** *nos\_sub*:  $set\ nos \subseteq Inl\ 'AD \cup Inr\ '\{..<length\ fv\_all\}$   
**using** *fo\_nmlz\_set*[*of AD as @ fs*]  
**by** (*auto simp: nos\_def as\_def fs\_def*)  
**have** *len\_fs'*:  $length\ fs' = length\ fv\_sub\_comp$   
**by** (*auto simp: fs'\_def nos\_def fo\_nmlz\_length as\_def fs\_def*)  
**have** *norm\_nos\_idem*:  $fo\_nmlz\_rec\ 0\ (id\_map\ 0)\ AD\ nos = nos$   
**using** *fo\_nmlz\_idem*[*of AD nos*] *fo\_nmlz\_sound*  
**by** (*auto simp: nos\_def fo\_nmlz\_def id\_map\_empty*)  
**have** *fs'\_all*:  $fs' \in nall\_tuples\_rec\ AD\ (card\ (Inr\ -\ 'set\ as'))\ (length\ fv\_sub\_comp)$   
**unfolding** *len\_fs'*[*symmetric*]  
**by** (*rule nall\_tuples\_rec\_fo\_nmlz\_rec\_complete*)



```

    (rule fo_nmlz_rec_shift[OF norm_nos_idem, simplified, OF refl len_fv_sub_nos,
      folded as'_def fs'_def])
  have as' ∈ nall_tuples AD (length fv_sub)
    using length_as'
    apply (rule nall_tuplesI)
    using norm_as' .
  then have as'_ass: as' ∈ ass
    using as'_norm_as σ_def(1) as_def
    unfolding assms(6)
    by (auto simp: proj_vals_def)
  have vs_norm: xs = fo_nmlz AD (map snd (merge (zip fv_sub as) (zip fv_sub_comp fs)))
    using assms(1,2,4) σ_def(2)
    by (auto simp: merge_map as_def fs_def fv_all_sort)
  have set_sort': set (sort (fv_sub @ fv_sub_comp)) = set (fv_sub @ fv_sub_comp)
    by auto
  have xs = fo_nmlz AD (map snd (merge (zip fv_sub as') (zip fv_sub_comp fs')))
    unfolding vs_norm as_def fs_def τ_def
      merge_map[OF assms(1,2,4)]
    apply (rule fo_nmlz_eqI)
    apply (rule adAgr_list_subset[OF equalityD1, OF set_sort'])
    using fo_nmlz_adAgr[of AD as @ fs, folded nos_def, unfolded nos_as'_fs']
    unfolding as_def fs_def τ_def map_append[symmetric] .
  then show ?thesis
    using as'_ass fs'_all
    by (auto simp: ext_tuple_def length_as')
qed

definition ext_tuple_set AD ns ns' X = (if ns' = [] then X else fo_nmlz AD ' ∪ (ext_tuple AD ns ns'
' X))

lemma ext_tuple_set_eq: Ball X (fo_nmlzd AD) ⇒ ext_tuple_set AD ns ns' X = fo_nmlz AD '
∪ (ext_tuple AD ns ns' ' X)
  by (auto simp: ext_tuple_set_def ext_tuple_def fo_nmlzd_code)

lemma ext_tuple_set_mono: A ⊆ B ⇒ ext_tuple_set AD ns ns' A ⊆ ext_tuple_set AD ns ns' B
  by (auto simp: ext_tuple_set_def)

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
    ∧ σ τ. adAgr_sets (set fv_sub) (set fv_sub) AD σ τ ⇒ σ ∈ R ↔ τ ∈ R
  shows ext_tuple_set AD fv_sub fv_sub_comp ass = fo_nmlz AD ' proj_vals R fv_all
proof (rule set_eqI, rule iffI)
  fix xs
  assume xs_in: xs ∈ ext_tuple_set AD fv_sub fv_sub_comp ass
  show xs ∈ fo_nmlz AD ' proj_vals R fv_all
    using ext_tuple_sound(2)[OF assms] xs_in
    by (auto simp: ext_tuple_set_def ext_tuple_def assms(6) fo_nmlz_idem[OF fo_nmlz_sound] im-
age_iff
      split: if_splits)
next
  fix xs
  assume xs ∈ fo_nmlz AD ' proj_vals R fv_all
  then obtain σ where σ_def: xs = fo_nmlz AD (map σ fv_all) σ ∈ R
    by (auto simp: proj_vals_def)
  show xs ∈ ext_tuple_set AD fv_sub fv_sub_comp ass
    using ext_tuple_complete[OF assms σ_def]

```

by (auto simp: ext\_tuple\_set\_def ext\_tuple\_def assms(6) fo\_nmlz\_idem[OF fo\_nmlz\_sound] image\_iff split: if\_splits)  
qed

**lemma proj\_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  
 $\wedge \sigma \tau$ . ad\_agr\_sets (set fv\_sub) (set fv\_sub) 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))  $\in$  ass  
**shows** xs  $\in$  fo\_nmlz AD '  $\bigcup$  (ext\_tuple AD fv\_sub fv\_sub\_comp ' ass)

**proof** –

**have** fv\_all\_sort: fv\_all = sort (fv\_sub @ fv\_sub\_comp)  
**using** assms(1,2,3,4,5)  
**by** (simp add: sorted\_distinct\_set\_unique)  
**obtain**  $\sigma$  **where**  $\sigma\_def$ : xs = map  $\sigma$  fv\_all  
**using** exists\_map[of fv\_all xs] assms(3,9)  
**by** auto  
**have** xs\_norm: xs = fo\_nmlz AD (map  $\sigma$  fv\_all)  
**using** assms(8)  
**by** (auto simp:  $\sigma\_def$ )  
**have** proj: proj\_tuple fv\_sub (zip fv\_all xs) = map  $\sigma$  fv\_sub  
**unfolding**  $\sigma\_def$   
**apply** (rule proj\_tuple\_map[OF assms(1,3)])  
**using** assms(5)  
**by** blast  
**obtain**  $\tau$  **where**  $\tau\_def$ : fo\_nmlz AD (map  $\sigma$  fv\_sub) = fo\_nmlz AD (map  $\tau$  fv\_sub)  $\tau \in R$   
**using** assms(10)  
**by** (auto simp: assms(6) proj proj\_vals\_def)  
**have**  $\sigma\_R$ :  $\sigma \in R$   
**using** assms(7) fo\_nmlz\_eqD[OF  $\tau\_def$ (1)]  $\tau\_def$ (2)  
**unfolding** ad\_agr\_list\_link[symmetric]  
**by** auto  
**show** ?thesis  
**by** (rule ext\_tuple\_complete[OF assms(1,2,3,4,5,6,7) xs\_norm  $\sigma\_R$ ]) assumption  
qed

**lemma proj\_tuple\_correct:**

**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  
 $\wedge \sigma \tau$ . ad\_agr\_sets (set fv\_sub) (set fv\_sub) AD  $\sigma \tau \implies \sigma \in R \longleftrightarrow \tau \in R$   
fo\_nmlz AD xs = xs length xs = length fv\_all  
**shows** xs  $\in$  fo\_nmlz AD '  $\bigcup$  (ext\_tuple AD fv\_sub fv\_sub\_comp ' ass)  $\longleftrightarrow$   
fo\_nmlz AD (proj\_tuple fv\_sub (zip fv\_all xs))  $\in$  ass  
**using** ext\_tuple\_sound(1)[OF assms(1,2,3,4,5,6,7)] proj\_tuple\_sound[OF assms]  
**by** blast

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

| `unify_vals_terms _ _ _ = None`

**lemma** `unify_vals_terms_extends`: `unify_vals_terms vs ts  $\sigma$  = Some  $\sigma'$   $\implies$  extends_subst  $\sigma$   $\sigma'$`   
**unfolding** `extends_subst_def`  
**by** (`induction vs ts  $\sigma$  arbitrary:  $\sigma'$  rule: unify_vals_terms.induct`)  
(`force split: if_splits option.splits`)**+**

**lemma** `unify_vals_terms_sound`: `unify_vals_terms vs ts  $\sigma$  = Some  $\sigma'$   $\implies$  (the  $\circ$   $\sigma'$ )  $\odot_e$  ts = vs`  
**using** `unify_vals_terms_extends`  
**by** (`induction vs ts  $\sigma$  arbitrary:  $\sigma'$  rule: unify_vals_terms.induct`)  
(`force simp: eval_eterms_def extends_subst_def fv_fo_terms_set_def`  
`split: if_splits option.splits`)**+**

**lemma** `unify_vals_terms_complete`:  `$\sigma'' \odot_e$  ts = vs  $\implies$  ( $\bigwedge n. \sigma n \neq \text{None} \implies \sigma n = \text{Some} (\sigma'' n)$ )  $\implies$`   
 `$\exists \sigma'. \text{unify\_vals\_terms vs ts } \sigma = \text{Some } \sigma'$`   
**by** (`induction vs ts  $\sigma$  rule: unify_vals_terms.induct`)  
(`force simp: eval_eterms_def extends_subst_def split: if_splits option.splits`)**+**

**definition** `eval_table` :: `'a fo_term list  $\Rightarrow$  ('a + 'c) table  $\Rightarrow$  ('a + 'c) table` **where**  
`eval_table ts X = (let fvs = fv_fo_terms_list ts in`  
 `$\bigcup$  (( $\lambda$ vs. case unify_vals_terms vs ts Map.empty of Some  $\sigma \Rightarrow$`   
`{map (the  $\circ$   $\sigma$ ) fvs} | _  $\Rightarrow$  { }) ' X))`

**lemma** `eval_table`:

**fixes** `X` :: `('a + 'c) table`  
**shows** `eval_table ts X = proj_vals { $\sigma. \sigma \odot_e$  ts  $\in$  X} (fv_fo_terms_list ts)`  
**proof** (`rule set_eqI`, `rule iffI`)  
**fix** `vs`  
**assume** `vs  $\in$  eval_table ts X`  
**then obtain** `as  $\sigma$  where as_def: as  $\in$  X unify_vals_terms as ts Map.empty = Some  $\sigma$`   
`vs = map (the  $\circ$   $\sigma$ ) (fv_fo_terms_list ts)`  
**by** (`auto simp: eval_table_def split: option.splits`)  
**have** `(the  $\circ$   $\sigma$ )  $\odot_e$  ts  $\in$  X`  
**using** `unify_vals_terms_sound[OF as_def(2)] as_def(1)`  
**by** `auto`  
**with** `as_def(3)` **show** `vs  $\in$  proj_vals { $\sigma. \sigma \odot_e$  ts  $\in$  X} (fv_fo_terms_list ts)`  
**by** (`fastforce simp: proj_vals_def`)

**next**

**fix** `vs` :: `('a + 'c) list`  
**assume** `vs  $\in$  proj_vals { $\sigma. \sigma \odot_e$  ts  $\in$  X} (fv_fo_terms_list ts)`  
**then obtain**  `$\sigma$  where  $\sigma\_def$ : vs = map  $\sigma$  (fv_fo_terms_list ts)  $\sigma \odot_e$  ts  $\in$  X`  
**by** (`auto simp: proj_vals_def`)  
**obtain**  `$\sigma'$  where  $\sigma'\_def$ : unify_vals_terms ( $\sigma \odot_e$  ts) ts Map.empty = Some  $\sigma'$`   
**using** `unify_vals_terms_complete[OF refl, of Map.empty  $\sigma$  ts]`  
**by** `auto`  
**have** `(the  $\circ$   $\sigma'$ )  $\odot_e$  ts = ( $\sigma \odot_e$  ts)`  
**using** `unify_vals_terms_sound[OF  $\sigma'\_def$ (1)]`  
**by** `auto`  
**then have** `vs = map (the  $\circ$   $\sigma'$ ) (fv_fo_terms_list ts)`  
**using** `fv_fo_terms_set_list eval_eterms_fv_fo_terms_set`  
**unfolding**  `$\sigma\_def$ (1)`  
**by** `fastforce`  
**then show** `vs  $\in$  eval_table ts X`  
**using**  `$\sigma\_def$ (2)  $\sigma'\_def$`   
**by** (`force simp: eval_table_def`)

**qed**

**fun** `ad_agr_close_rec` :: `nat  $\Rightarrow$  (nat  $\rightarrow$  'a + nat)  $\Rightarrow$  'a set  $\Rightarrow$`

```

('a + nat) list ⇒ ('a + nat) list set where
ad_agr_close_rec i m AD [] = {}
| ad_agr_close_rec i m AD (Inl x # xs) = (λxs. 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 ⇒ ⋃((λx. (λxs. Inl x # xs) ‘
  ad_agr_close_rec i (m(n := Some (Inl x)))) (AD - {x}) xs) ‘ AD) ∪
  (λxs. Inr i # xs) ‘ ad_agr_close_rec (Suc i) (m(n := Some (Inr i))) AD xs
| Some v ⇒ (λxs. v # xs) ‘ ad_agr_close_rec i m AD xs

lemma ad_agr_close_rec_length: ys ∈ ad_agr_close_rec i m AD xs ⇒ length xs = length ys
by (induction i m AD xs arbitrary: ys rule: ad_agr_close_rec.induct) (auto split: option.splits)

lemma ad_agr_close_rec_sound: ys ∈ ad_agr_close_rec i m AD xs ⇒
fo_nmlz_rec j (id_map j) X xs = xs ⇒ X ∩ AD = {} ⇒ X ∩ Y = {} ⇒ Y ∩ AD = {} ⇒
inj_on m (dom m) ⇒ dom m = {..<j} ⇒ ran m ⊆ Inl ‘ Y ∪ Inr ‘ {..<i} ⇒ i ≤ j ⇒
fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) ys = ys ∧
(∃ m'. inj_on m' (dom m') ∧ (∀ n v. m n = Some v → m' (Inr n) = Some v) ∧
(∀ (x, y) ∈ set (zip xs ys). case x of Inl x' ⇒
  if x' ∈ X then x = y else m' x = Some y ∧ (case y of Inl z ⇒ z ∉ X | Inr x ⇒ True)
| Inr n ⇒ m' x = Some y ∧ (case y of Inl z ⇒ z ∉ X | Inr x ⇒ True)))
proof (induction i m AD xs arbitrary: Y j ys rule: ad_agr_close_rec.induct)
case (1 i m AD)
then show ?case
by (auto simp: ad_agr_list_def ad_equiv_list_def sp_equiv_list_def inj_on_def dom_def
split: sum.splits intro!: exI[of _ case_sum Map.empty m])
next
case (2 i m AD x xs)
obtain zs where ys_def: ys = Inl x # zs zs ∈ ad_agr_close_rec i m AD xs
using 2(2)
by auto
have preds: fo_nmlz_rec j (id_map j) X xs = xs x ∈ X
using 2(3)
by (auto split: if_splits option.splits)
show ?case
using 2(1)[OF ys_def(2) preds(1) 2(4,5,6,7,8,9,10)] preds(2)
by (auto simp: ys_def(1))
next
case (3 i m AD n xs)
show ?case
proof (cases m n)
case None
obtain v zs where ys_def: ys = v # zs
using 3(4)
by (auto simp: None)
have n_ge_j: j ≤ n
using 3(9,10) None
by (metis domIff leI lessThan_iff)
show ?thesis
proof (cases v)
case (Inl x)
have zs_def: zs ∈ ad_agr_close_rec i (m(n ↦ Inl x)) (AD - {x}) xs x ∈ AD
using 3(4)
by (auto simp: None ys_def Inl)
have preds: fo_nmlz_rec (Suc j) (id_map (Suc j)) X xs = xs X ∩ (AD - {x}) = {}
  X ∩ (Y ∪ {x}) = {} (Y ∪ {x}) ∩ (AD - {x}) = {} dom (m(n ↦ Inl x)) = {..<Suc j}
  ran (m(n ↦ Inl x)) ⊆ Inl ‘ (Y ∪ {x}) ∪ Inr ‘ {..<i}
  i ≤ Suc j n = j
using 3(5,6,7,8,10,11,12) n_ge_j zs_def(2)
by (auto simp: fun_upd_id_map ran_def dest: id_mapD split: option.splits)

```

```

have inj: inj_on (m(n ↦ Inl x)) (dom (m(n ↦ Inl x)))
  using 3(8,9,10,11,12) preds(8) zs_def(2)
  by (fastforce simp: inj_on_def dom_def ran_def)
have sets_unfold: X ∪ (Y ∪ {x}) ∪ (AD - {x}) = X ∪ Y ∪ AD
  using zs_def(2)
  by auto
note IH = 3(1)[OF None zs_def(2,1) preds(1,2,3,4) inj preds(5,6,7), unfolded sets_unfold]
have norm_ys: fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) ys = ys
  using conjunct1[OF IH] zs_def(2)
  by (auto simp: ys_def(1) Inl split: option.splits)
show ?thesis
  using norm_ys conjunct2[OF IH] None zs_def(2) 3(6)
  unfolding ys_def(1)
  apply safe
  subgoal for m'
    apply (auto simp: Inl dom_def intro!: exI[of _ m'] split: if_splits)
    apply (metis option.distinct(1))
    apply (fastforce split: prod.splits sum.splits)
    done
  done
next
case (Inr k)
have zs_def: zs ∈ ad_agr_close_rec (Suc i) (m(n ↦ Inr i)) AD xs i = k
  using 3(4)
  by (auto simp: None ys_def Inr)
have preds: fo_nmlz_rec (Suc n) (id_map (Suc n)) X xs = xs
  dom (m(n ↦ Inr i)) = {..<Suc n}
  ran (m(n ↦ Inr i)) ⊆ Inl ' Y ∪ Inr ' {..<Suc i} Suc i ≤ Suc n
  using 3(5,10,11,12) n_ge_j
  by (auto simp: fun_upd_id_map ran_def dest: id_mapD split: option.splits)
have inj: inj_on (m(n ↦ Inr i)) (dom (m(n ↦ Inr i)))
  using 3(9,11)
  by (auto simp: inj_on_def dom_def ran_def)
note IH = 3(2)[OF None zs_def(1) preds(1) 3(6,7,8) inj preds(2,3,4)]
have norm_ys: fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) ys = ys
  using conjunct1[OF IH] zs_def(2)
  by (auto simp: ys_def Inr fun_upd_id_map dest: id_mapD split: option.splits)
show ?thesis
  using norm_ys conjunct2[OF IH] None
  unfolding ys_def(1) zs_def(2)
  apply safe
  subgoal for m'
    apply (auto simp: Inr dom_def intro!: exI[of _ m'] split: if_splits)
    apply (metis option.distinct(1))
    apply (fastforce split: prod.splits sum.splits)
    done
  done
qed
next
case (Some v)
obtain zs where ys_def: ys = v # zs zs ∈ ad_agr_close_rec i m AD xs
  using 3(4)
  by (auto simp: Some)
have preds: fo_nmlz_rec j (id_map j) X xs = xs n < j
  using 3(5,8,10) Some
  by (auto simp: dom_def split: option.splits)
note IH = 3(3)[OF Some ys_def(2) preds(1) 3(6,7,8,9,10,11,12)]
have norm_ys: fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) ys = ys

```

```

using conjunct1[OF IH] 3(11) Some
by (auto simp: ys_def(1) ran_def id_map_def)
have case v of Inl z  $\Rightarrow$  z  $\notin$  X | Inr x  $\Rightarrow$  True
using 3(7,11) Some
by (auto simp: ran_def split: sum.splits)
then show ?thesis
using norm_ys conjunct2[OF IH] Some
unfolding ys_def(1)
apply safe
subgoal for m'
  by (auto intro!: exI[of _ m'] split: sum.splits)
done
qed
qed

lemma adAgrCloseRecComplete:
  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 = {..\Longrightarrow ran m = Inl ' Y  $\cup$  Inr ' {..\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  $\Longrightarrow$  adAgrList X xs ys  $\Longrightarrow$ 
  ys  $\in$  adAgrCloseRec i m AD xs
proof (induction j id_map j :: 'a + nat  $\Rightarrow$  nat option X xs arbitrary: m i ys AD Y
  rule: fo_nmlz_rec.induct)
case (2 j X x xs)
have x_X: x  $\in$  X fo_nmlz_rec j (id_map j) X xs = xs
  using 2(4)
  by (auto split: if_splits option.splits)
obtain z zs where ys_def: ys = Inl z # zs z = x
  using 2(14) x_X(1)
  by (cases ys) (auto simp: adAgrList_def adEquivList_def adEquivPair_simps)
have norm_zs: fo_nmlz_rec i (id_map i) (X  $\cup$  Y  $\cup$  AD) zs = zs
  using 2(13) ys_def(2) x_X(1)
  by (auto simp: ys_def(1))
have adAgr: adAgrList X xs zs
  using 2(14)
  by (auto simp: ys_def adAgrList_def adEquivList_def spEquivList_def pairwise_def)
show ?case
  using 2(1)[OF x_X 2(5,6,7,8,9,10,11) _ norm_zs adAgr] 2(12)
  by (auto simp: ys_def)
next
case (3 j X n xs)
obtain z zs where ys_def: ys = z # zs
  using 3(13)
  apply (cases ys)
  apply (auto simp: adAgrList_def)
  done
show ?case
proof (cases j  $\leq$  n)
case True
  then have n_j: n = j
  using 3(3)
  by (auto split: option.splits dest: id_mapD)
have id_map: id_map j (Inr n) = None (id_map j)(Inr n  $\mapsto$  j) = id_map (Suc j)
  unfolding n_j fun_upd_id_map
  by (auto simp: id_map_def)
have norm_xs: fo_nmlz_rec (Suc j) (id_map (Suc j)) X xs = xs

```

```

using 3(3)
by (auto simp: ys_def fun_upd_id_map id_map(1) split: option.splits)
have None: m n = None
using 3(8)
by (auto simp: dom_def n_j)
have z_out: z ∉ Inl ' Y ∪ Inr ' {..i}
using 3(11) None
by (force simp: ys_def 3(9))
show ?thesis
proof (cases z)
  case (Inl a)
    have a_in: a ∈ AD
      using 3(12,13) z_out
      by (auto simp: ys_def Inl ad_agr_list_def ad_equiv_list_def ad_equiv_pair.simps
        split: if_splits option.splits)
    have norm_zs: fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) zs = zs
      using 3(12) a_in
      by (auto simp: ys_def Inl)
    have preds: X ∩ (AD - {a}) = {} X ∩ (Y ∪ {a}) = {} (Y ∪ {a}) ∩ (AD - {a}) = {}
      using 3(4,5,6) a_in
      by auto
    have inj: inj_on (m(n := Some (Inl a))) (dom (m(n := Some (Inl a))))
      using 3(6,7,9) None a_in
      by (auto simp: inj_on_def dom_def ran_def) blast+
    have preds': dom (m(n ↦ Inl a)) = {..Suc j}
      ran (m(n ↦ Inl a)) = Inl ' (Y ∪ {a}) ∪ Inr ' {..i} i ≤ Suc j
      using 3(6,8,9,10) None less_Suc_eq a_in
      apply (auto simp: n_j dom_def ran_def)
      apply (smt Un_iff image_eqI mem_Collect_eq option.simps(3))
      apply (smt 3(8) domIff image_subset_iff lessThan_iff mem_Collect_eq sup_ge2)
      done
    have a_unfold: X ∪ (Y ∪ {a}) ∪ (AD - {a}) = X ∪ Y ∪ AD Y ∪ {a} ∪ (AD - {a}) = Y ∪ AD
      using a_in
      by auto
    have ad_agr: ad_agr_list X xs zs
      using 3(13)
      by (auto simp: ys_def Inl ad_agr_list_def ad_equiv_list_def sp_equiv_list_def pairwise_def)
    have zs ∈ ad_agr_close_rec i (m(n ↦ Inl a)) (AD - {a}) xs
      apply (rule 3(1)[OF id_map norm_xs preds inj preds' __ ad_agr])
      using 3(11,13) norm_zs
    unfolding 3(9) preds'(2) a_unfold
      apply (auto simp: None Inl ys_def ad_agr_list_def sp_equiv_list_def pairwise_def
        split: option.splits)
      apply (metis Un_iff image_eqI option.simps(4))
      apply (metis image_subset_iff lessThan_iff option.simps(4) sup_ge2)
      apply fastforce
      done
    then show ?thesis
      using a_in
      by (auto simp: ys_def Inl None)
  next
    case (Inr b)
    have i_b: i = b
      using 3(12) z_out
      by (auto simp: ys_def Inr split: option.splits dest: id_mapD)
    have norm_zs: fo_nmlz_rec (Suc i) (id_map (Suc i)) (X ∪ Y ∪ AD) zs = zs
      using 3(12)
      by (auto simp: ys_def Inr i_b fun_upd_id_map split: option.splits dest: id_mapD)

```

```

have adAgr: adAgr_list X xs zs
  using 3(13)
  by (auto simp: ys_def adAgr_list_def ad_equiv_list_def sp_equiv_list_def pairwise_def)
define m' where m' ≡ m(n := Some (Inr i))
have preds: inj_on m' (dom m') dom m' = {.. $Suc\ j$ }  $Suc\ i \leq Suc\ j$ 
  using 3(7,8,9,10)
  by (auto simp: m'_def n_j inj_on_def dom_def ran_def image_iff)
    (metis 3(8) domI lessThan_iff less_SucI)
have ran: ran m' = Inl ' Y  $\cup$  Inr ' {.. $Suc\ i$ }
  using 3(9) None
  by (auto simp: m'_def)
have zs ∈ adAgr_close_rec (Suc i) m' AD xs
  apply (rule 3(1)[OF id_map norm_xs 3(4,5,6) preds(1,2) ran preds(3) _ norm_zs adAgr])
  using 3(11,13)
  unfolding 3(9) ys_def Inr i_b m'_def
  unfolding ran[unfolded m'_def i_b]
  apply (auto simp: adAgr_list_def sp_equiv_list_def pairwise_def split: option.splits)
    apply (metis Un_upper1 image_subset_iff option.simps(4))
    apply (metis UnI1 image_eqI insert_iff lessThan_Suc lessThan_iff option.simps(4)
      sp_equiv_pair.simps sum.inject(2) sup_commute)
  apply fastforce
done
then show ?thesis
  by (auto simp: ys_def Inr None m'_def i_b)
qed
next
case False
have id_map: id_map j (Inr n) = Some n
  using False
  by (auto simp: id_map_def)
have norm_xs: fo_nmlz_rec j (id_map j) X xs = xs
  using 3(3)
  by (auto simp: id_map)
have Some: m n = Some z
  using False 3(11)[unfolded ys_def]
  by (metis (mono_tags) 3(8) domD insert_iff leI lessThan_iff list.simps(15)
    option.simps(5) zip_Cons_Cons)
have z_in: z ∈ Inl ' Y  $\cup$  Inr ' {.. $i$ }
  using 3(9) Some
  by (auto simp: ran_def)
have adAgr: adAgr_list X xs zs
  using 3(13)
  by (auto simp: adAgr_list_def ys_def ad_equiv_list_def sp_equiv_list_def pairwise_def)
show ?thesis
proof (cases z)
case (Inl a)
have a_in: a ∈ Y  $\cup$  AD
  using 3(12,13)
  by (auto simp: ys_def Inl adAgr_list_def ad_equiv_list_def ad_equiv_pair.simps
    split: if_splits option.splits)
have norm_zs: fo_nmlz_rec i (id_map i) (X  $\cup$  Y  $\cup$  AD) zs = zs
  using 3(12) a_in
  by (auto simp: ys_def Inl)
show ?thesis
  using 3(2)[OF id_map norm_xs 3(4,5,6,7,8,9,10) _ norm_zs adAgr] 3(11) a_in
  by (auto simp: ys_def Inl Some split: option.splits)
next
case (Inr b)

```



```

have b_lt: b < i
  using z_in
  by (auto simp: Inr)
have norm_zs: fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) zs = zs
  using 3(12) b_lt
  by (auto simp: ys_def Inr split: option.splits)
show ?thesis
  using 3(2)[OF id_map norm_xs 3(4,5,6,7,8,9,10) _ norm_zs adAgr] 3(11)
  by (auto simp: ys_def Inr Some)
qed
qed
qed (auto simp: adAgr_list_def)

definition adAgr_close :: 'a set ⇒ ('a + nat) list ⇒ ('a + nat) list set where
  adAgr_close AD xs = adAgr_close_rec 0 Map.empty AD xs

lemma adAgr_close_sound:
assumes ys ∈ adAgr_close Y xs fo_nmlzd X xs X ∩ Y = {}
shows fo_nmlzd (X ∪ Y) ys ∧ adAgr_list X xs ys
using adAgr_close_rec_sound[OF assms(1)[unfolded adAgr_close_def]
  fo_nmlz_idem[OF assms(2), unfolded fo_nmlz_def, folded id_map_empty] assms(3)
  Int_empty_right Int_empty_left]
  adAgr_map[OF adAgr_close_rec_length[OF assms(1)[unfolded adAgr_close_def]], of _ X]
  fo_nmlzd_code[unfolded fo_nmlz_def, folded id_map_empty, of X ∪ Y ys]
by (auto simp: fo_nmlz_def)

lemma adAgr_close_complete:
assumes X ∩ Y = {} fo_nmlzd X xs fo_nmlzd (X ∪ Y) ys adAgr_list X xs ys
shows ys ∈ adAgr_close Y xs
using adAgr_close_rec_complete[OF fo_nmlz_idem[OF assms(2),
  unfolded fo_nmlz_def, folded id_map_empty] assms(1) Int_empty_right Int_empty_left _ _
  order.refl _ _ assms(4), of Map.empty]
  fo_nmlzd_code[unfolded fo_nmlz_def, folded id_map_empty, of X ∪ Y ys]
  assms(3)
unfolding adAgr_close_def
by (auto simp: fo_nmlz_def)

lemma adAgr_close_empty: fo_nmlzd X xs ⇒ adAgr_close {} xs = {xs}
using adAgr_close_complete[where ?X=X and ?Y={} and ?xs=xs and ?ys=xs]
  adAgr_close_sound[where ?X=X and ?Y={} and ?xs=xs] adAgr_list_refl adAgr_list_fo_nmlzd
by fastforce

lemma adAgr_close_set_correct:
assumes AD' ⊆ AD sorted_distinct ns
  ∧σ τ. adAgr_sets (set ns) (set ns) AD' σ τ ⇒ σ ∈ R ↔ τ ∈ R
shows ⋃(adAgr_close (AD - AD') ' fo_nmlz AD' ' proj_vals R ns) = fo_nmlz AD ' proj_vals R ns
proof (rule set_eqI, rule iffI)
  fix vs
  assume vs ∈ ⋃(adAgr_close (AD - AD') ' fo_nmlz AD' ' proj_vals R ns)
  then obtain σ where σ_def: vs ∈ adAgr_close (AD - AD') (fo_nmlz AD' (map σ ns)) σ ∈ R
    by (auto simp: proj_vals_def)
  have vs: fo_nmlzd AD vs adAgr_list AD' (fo_nmlz AD' (map σ ns)) vs
    using adAgr_close_sound[OF σ_def(1) fo_nmlz_sound] assms(1) Diff_partition
    by fastforce+
  obtain τ where τ_def: vs = map τ ns
    using exists_map[of ns vs] assms(2) vs(2)
    by (auto simp: adAgr_list_def fo_nmlz_length)
  show vs ∈ fo_nmlz AD ' proj_vals R ns

```

```

apply (subst fo_nmlz_idem[OF vs(1), symmetric])
using iffD1[OF assms(3)  $\sigma\_def(2)$ , OF iffD2[OF ad_agr_list_link ad_agr_list_trans[OF
  fo_nmlz_ad_agr[of AD' map  $\sigma$  ns] vs(2), unfolded  $\tau\_def$ ]]]
unfolding  $\tau\_def$ 
by (auto simp: proj_vals_def)
next
fix vs
assume vs  $\in$  fo_nmlz AD ' proj_vals R ns
then obtain  $\sigma$  where  $\sigma\_def: vs = fo\_nmlz AD (map \sigma ns) \sigma \in R$ 
by (auto simp: proj_vals_def)
define xs where xs = fo_nmlz AD' vs
have preds: AD'  $\cap$  (AD - AD') = {} fo_nmlzd AD' xs fo_nmlzd (AD'  $\cup$  (AD - AD')) vs
using assms(1) fo_nmlz_sound Diff_partition
by (fastforce simp:  $\sigma\_def(1)$  xs_def)+
obtain  $\tau$  where  $\tau\_def: vs = map \tau ns$ 
using exists_map[of ns vs] assms(2)  $\sigma\_def(1)$ 
by (auto simp: fo_nmlz_length)
have vs  $\in$  ad_agr_close (AD - AD') xs
using ad_agr_close_complete[OF preds] ad_agr_list_comm[OF fo_nmlz_ad_agr]
by (auto simp: xs_def)
then show vs  $\in$   $\bigcup$ (ad_agr_close (AD - AD') ' fo_nmlz AD' ' proj_vals R ns)
unfolding xs_def  $\tau\_def$ 
using iffD1[OF assms(3)  $\sigma\_def(2)$ , OF ad_agr_sets_mono[OF assms(1) iffD2[OF ad_agr_list_link
  fo_nmlz_ad_agr[of AD map  $\sigma$  ns, folded  $\sigma\_def(1)$ , unfolded  $\tau\_def$ ]]]]
by (auto simp: proj_vals_def)
qed

```

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

**assumes** AD'  $\subseteq$  AD

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

**shows**  $\bigcup$ (ad\_agr\_close (AD - AD') ' fo\_nmlz AD' ' proj\_fmla  $\varphi$  R) = fo\_nmlz AD ' proj\_fmla  $\varphi$  R

**using** ad\_agr\_close\_set\_correct[OF sorted\_distinct\_fv\_list, OF assms]

**by** (auto simp: proj\_fmla\_def)

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

**lemma** ad\_agr\_close\_set\_eq: Ball X (fo\_nmlzd AD')  $\implies$  ad\_agr\_close\_set AD X =  $\bigcup$ (ad\_agr\_close AD ' X)

**by** (force simp: ad\_agr\_close\_set\_def Set.is\_empty\_def ad\_agr\_close\_empty)

**lemma** Ball\_fo\_nmlzd: Ball (fo\_nmlz AD ' X) (fo\_nmlzd AD)

**by** (auto simp: fo\_nmlz\_sound)

**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 **where**

eval\_pred ts X = (let AD =  $\bigcup$ (set (map set\_fo\_term ts))  $\cup$   $\bigcup$ (set ' X) in  
 (AD, length (fv\_fo\_terms\_list ts), eval\_table ts (map Inl ' X)))

**definition** eval\_bool :: bool  $\Rightarrow$  ('a, 'c) fo\_t **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 **where**

eval\_eq t t' = (case t of Var n  $\Rightarrow$

(case t' 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  $t'$  of  $Var\ n' \Rightarrow (\{c\}, 1, \{[Inl\ c]\})$   
   |  $Const\ c' \Rightarrow$  if  $c = c'$  then  $(\{c\}, 0, \{\emptyset\})$  else  $(\{c, c'\}, 0, \{\})$ )

**fun**  $eval\_neg :: nat\ list \Rightarrow ('a, nat)\ fo\_t \Rightarrow ('a, nat)\ fo\_t$  **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  $cx = filter\ (\lambda(n, x). n \notin set\ ns\psi \wedge isl\ x)\ (zip\ ns\ \varphi\ xs)$ ;  
    $nxs = map\ fst\ (filter\ (\lambda(n, x). n \notin set\ ns\psi \wedge \neg isl\ x)\ (zip\ ns\ \varphi\ xs))$ ;  
    $cys = filter\ (\lambda(n, y). n \notin set\ ns\ \varphi \wedge isl\ y)\ (zip\ ns\ \psi\ ys)$ ;  
    $nys = map\ fst\ (filter\ (\lambda(n, y). n \notin set\ ns\ \varphi \wedge \neg isl\ y)\ (zip\ ns\ \psi\ ys))$  in  
 $fo\_nmlz\ AD\ 'ext\_tuple\ \{\}\ (sort\ (ns\ \varphi @ map\ fst\ cys))\ nys\ (map\ snd\ (merge\ (zip\ ns\ \varphi\ xs)\ cys)) \cap$   
 $fo\_nmlz\ AD\ 'ext\_tuple\ \{\}\ (sort\ (ns\ \psi @ map\ fst\ cx))\ nxs\ (map\ snd\ (merge\ (zip\ ns\ \psi\ ys)\ cx))$ )

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

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

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

**fun**  $eval\_ajoin :: nat\ list \Rightarrow ('a, nat)\ fo\_t \Rightarrow nat\ list \Rightarrow ('a, nat)\ fo\_t \Rightarrow$   
 ('a, nat)  $fo\_t$  **where**  
 $eval\_ajoin\ ns\ \varphi\ (AD\ \varphi, \_, X\ \varphi)\ ns\psi\ (AD\ \psi, \_, X\ \psi) = (let\ AD = AD\ \varphi \cup AD\ \psi; AD\ \Delta\ \varphi = AD - AD\ \varphi;$   
 $AD\ \Delta\ \psi = AD - AD\ \psi;$   
    $ns = filter\ (\lambda n. n \in set\ ns\psi)\ ns\ \varphi; ns\ \varphi' = filter\ (\lambda n. n \notin set\ ns\ \varphi)\ ns\psi;$   
    $idx\ \varphi = cluster\ (Some \circ (\lambda xs. fo\_nmlz\ AD\ \psi\ (proj\_tuple\ ns\ (zip\ ns\ \varphi\ xs))))\ (ad\_agr\_close\_set\ AD\ \Delta\ \varphi\ X\ \varphi);$   
    $idx\ \psi = cluster\ (Some \circ (\lambda ys. fo\_nmlz\ AD\ \psi\ (proj\_tuple\ ns\ (zip\ ns\ \psi\ ys))))\ X\ \psi$  in  
 $(AD, card\ (set\ ns\ \varphi \cup set\ ns\ \psi), set\_of\_idx\ (Mapping.map\_values\ (\lambda xs\ X. case\ Mapping.lookup\ idx\ \psi\ xs\ of\ Some\ Y \Rightarrow$   
 $idx\_join\ AD\ ns\ ns\ \varphi\ X\ ns\psi\ (ad\_agr\_close\_set\ AD\ \Delta\ \psi\ (ext\_tuple\_set\ AD\ \psi\ ns\ ns\ \varphi'\ \{xs\} - Y)) \mid \_$   
 $\Rightarrow ext\_tuple\_set\ AD\ ns\ \varphi'\ X)\ idx\ \varphi))$ )

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

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

```

fun eval_forall :: nat  $\Rightarrow$  nat list  $\Rightarrow$  ('a, nat) fo_t  $\Rightarrow$  ('a, nat) fo_t where
  eval_forall i ns (AD, _, X) = (case pos i ns of Some j  $\Rightarrow$ 
    let n = card AD in
    (AD, length ns - 1, Mapping.keys (Mapping.filter ( $\lambda$ t Z. n + card (Inr - ' set t) + 1  $\leq$  card Z)
      (cluster (Some  $\circ$  ( $\lambda$ ts. fo_nmlz AD (rem_nth j ts))) X)))
    | None  $\Rightarrow$  (AD, length ns, X))

```

```

lemma combine_map2: assumes length ys = length xs length ys' = length xs'
  distinct xs distinct xs' set xs  $\cap$  set xs' = {}
shows  $\exists$ f. ys = map f xs  $\wedge$  ys' = map f xs'

```

```

proof -
obtain f g where fg_def: ys = map f xs ys' = map g xs'
using assms exists_map
by metis
show ?thesis
using assms
by (auto simp: fg_def intro!: exI[of _  $\lambda$ x. if x  $\in$  set xs then f x else g x])

```

qed

```

lemma combine_map3: assumes length ys = length xs length ys' = length xs' length ys'' = length xs''
  distinct xs distinct xs' distinct xs'' set xs  $\cap$  set xs' = {} set xs  $\cap$  set xs'' = {} set xs'  $\cap$  set xs'' = {}
shows  $\exists$ f. ys = map f xs  $\wedge$  ys' = map f xs'  $\wedge$  ys'' = map f xs''

```

```

proof -
obtain f g h where fgh_def: ys = map f xs ys' = map g xs' ys'' = map h xs''
using assms exists_map
by metis
show ?thesis
using assms
by (auto simp: fgh_def intro!: exI[of _  $\lambda$ x. if x  $\in$  set xs then f x else if x  $\in$  set xs' then g x else h x])

```

qed

```

lemma distinct_set_zip: length nsx = length xs  $\implies$  distinct nsx  $\implies$ 
  (a, b)  $\in$  set (zip nsx xs)  $\implies$  (a, ba)  $\in$  set (zip nsx xs)  $\implies$  b = ba
by (induction nsx xs rule: list_induct2) (auto dest: set_zip_leftD)

```

```

lemma fo_nmlz_idem_isl:
assumes  $\bigwedge$ x. x  $\in$  set xs  $\implies$  (case x of Inl z  $\Rightarrow$  z  $\in$  X | _  $\Rightarrow$  False)
shows fo_nmlz X xs = xs

```

```

proof -
have F1: Inl x  $\in$  set xs  $\implies$  x  $\in$  X for x
using assms[of Inl x]
by auto
have F2: List.map_filter (case_sum Map.empty Some) xs = []
using assms
by (induction xs) (fastforce simp: List.map_filter_def split: sum.splits)+
show ?thesis
by (rule fo_nmlz_idem) (auto simp: fo_nmlzd_def nats_def F2 intro: F1)

```

qed

```

lemma set_zip_mapI: x  $\in$  set xs  $\implies$  (f x, g x)  $\in$  set (zip (map f xs) (map g xs))
by (induction xs) auto

```

```

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  $\in$  set xs isl (f x)
shows f x = g x

```

```

proof -
have AD: ad_equiv_pair X (f x, g x)

```

```

using assms(1) set_zip_mapI[OF assms(3)]
by (auto simp: ad_agr_list_def ad_equiv_list_def split: sum.splits)
then show ?thesis
using assms(2-)
by (auto simp: fo_nmlzd_def) (metis AD ad_equiv_pair.simps ad_equiv_pair_mono image_eqI
sum.collapse(1) vimageI)
qed

```

**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 -
define cxs where cxs = filter ( $\lambda(n, x).$   $n \notin$  set nsy  $\wedge$  isl x) (zip nsx xs')
define nxs where nxs = map fst (filter ( $\lambda(n, x).$   $n \notin$  set nsy  $\wedge$   $\neg$ isl x) (zip nsx xs'))
define cys where cys = filter ( $\lambda(n, y).$   $n \notin$  set nsx  $\wedge$  isl y) (zip nsy ys')
define nys where nys = map fst (filter ( $\lambda(n, y).$   $n \notin$  set nsx  $\wedge$   $\neg$ isl y) (zip nsy ys'))
define both where both = sorted_list_of_set (set nsx  $\cup$  set nsy)
have close: fo_nmlzd (X  $\cup$  Y) xs' ad_agr_list X xs xs' fo_nmlzd (X  $\cup$  Y) ys' ad_agr_list Y ys ys'
using ad_agr_close_sound[OF assms(10) assms(1)] ad_agr_close_sound[OF assms(11) assms(2)]
by (auto simp add: sup_left_commute)
have close': length xs' = length xs length ys' = length ys
using close
by (auto simp: ad_agr_list_length)
have len_sort: length (sort (nsx @ map fst cys)) = length (map snd (merge (zip nsx xs') cys))
length (sort (nsy @ map fst cxs)) = length (map snd (merge (zip nsy ys') cxs))
by (auto simp: merge_length assms(3,4) close')
{
fix zs
assume zs  $\in$  fo_nmlz (X  $\cup$  Y) ' ( $\lambda$ fs. map snd (merge (zip (sort (nsx @ map fst cys)) (map snd
(merge (zip nsx xs') cys))) (zip nys fs))) ' (length nys)
nall_tuples_rec {} (card (Inr - ' set (map snd (merge (zip nsx xs') cys))) (length nys)
zs  $\in$  fo_nmlz (X  $\cup$  Y) ' ( $\lambda$ fs. map snd (merge (zip (sort (nsy @ map fst cxs)) (map snd (merge (zip
nsy ys') cxs))) (zip nxs fs))) ' (length nxs)
nall_tuples_rec {} (card (Inr - ' set (map snd (merge (zip nsy ys') cxs))) (length nxs)
then obtain zxs zys where nall: zxs  $\in$  nall_tuples_rec {} (card (Inr - ' set (map snd (merge (zip
nsx xs') cys))) (length nys)
zs = fo_nmlz (X  $\cup$  Y) (map snd (merge (zip (sort (nsx @ map fst cys)) (map snd (merge (zip nsx
xs') cys))) (zip nys zxs)))
zys  $\in$  nall_tuples_rec {} (card (Inr - ' set (map snd (merge (zip nsy ys') cxs))) (length nxs)
zs = fo_nmlz (X  $\cup$  Y) (map snd (merge (zip (sort (nsy @ map fst cxs)) (map snd (merge (zip nsy
ys') cxs))) (zip nxs zys)))
by auto
have len_zs: length zxs = length nys length zys = length nxs
using nall(1,3)
by (auto dest: nall_tuples_rec_length)
have aux: sorted_distinct (map fst cxs) sorted_distinct nxs sorted_distinct nsy
sorted_distinct (map fst cys) sorted_distinct nys sorted_distinct nsx
set (map fst cxs)  $\cap$  set nsy = {} set (map fst cxs)  $\cap$  set nxs = {} set nsy  $\cap$  set nxs = {}
set (map fst cys)  $\cap$  set nsx = {} set (map fst cys)  $\cap$  set nys = {} set nsx  $\cap$  set nys = {}
using assms(3,4,5,6) close' distinct_set_zip
by (auto simp: cxs_def nxs_def cys_def nys_def sorted_filter distinct_mapfst_filter)

```

```

    (smt (z3) distinct_set_zip)+
  obtain xf where xf_def: map snd cxs = map xf (map fst cxs) ys' = map xf nsy zys = map xf nxs
    using combine_map3[where ?ys=map snd cxs and ?xs=map fst cxs and ?ys'=ys' and ?xs'=nsy
and ?ys''=zys and ?xs''=nxs] assms(4) aux close'
  by (auto simp: len_zs)
  obtain ysf where ysf_def: ys = map ysf nsy
    using assms(4,6) exists_map
  by auto
  obtain xg where xg_def: map snd cys = map xg (map fst cys) xs' = map xg nsx zxs = map xg nys
    using combine_map3[where ?ys=map snd cys and ?xs=map fst cys and ?ys'=xs' and ?xs'=nsx
and ?ys''=zxs and ?xs''=nys] assms(3) aux close'
  by (auto simp: len_zs)
  obtain xf where xf_def: xs = map xf nsx
    using assms(3,5) exists_map
  by auto
  have set_cxs_nxs: set (map fst cxs @ nxs) = set nsx - set nsy
    using assms(3)
  unfolding cxs_def nxs_def close'[symmetric]
  by (induction nsx xs' rule: list_induct2) auto
  have set_cys_nys: set (map fst cys @ nys) = set nsy - set nsx
    using assms(4)
  unfolding cys_def nys_def close'[symmetric]
  by (induction nsy ys' rule: list_induct2) auto
  have sort_sort_both_xs: sort (sort (nsy @ map fst cxs) @ nxs) = both
    apply (rule sorted_distinct_set_unique)
    using assms(3,5,6) close' set_cxs_nxs
  by (auto simp: both_def nxs_def cxs_def intro: distinct_map_fst_filter)
    (metis (no_types, lifting) distinct_set_zip)
  have sort_sort_both_ys: sort (sort (nsx @ map fst cys) @ nys) = both
    apply (rule sorted_distinct_set_unique)
    using assms(4,5,6) close' set_cys_nys
  by (auto simp: both_def nys_def cys_def intro: distinct_map_fst_filter)
    (metis (no_types, lifting) distinct_set_zip)
  have map_snd (merge (zip nsy ys') cxs) = map xf (sort (nsy @ map fst cxs))
    using merge_map[where ? $\sigma$ =xf and ?ns=nsy and ?ms=map fst cxs] assms(6) aux
  unfolding xf_def(1)[symmetric] xf_def(2)
  by (auto simp: zip_map_fst_snd)
  then have zs_xf: zs = fo_nmlz (X  $\cup$  Y) (map xf both)
    using merge_map[where  $\sigma$ =xf and ?ns=sort (nsy @ map fst cxs) and ?ms=nxs] aux
  by (fastforce simp: nall(4) xf_def(3) sort_sort_both_xs)
  have map_snd (merge (zip nsx xs') cys) = map xg (sort (nsx @ map fst cys))
    using merge_map[where ? $\sigma$ =xg and ?ns=nsx and ?ms=map fst cys] assms(5) aux
  unfolding xg_def(1)[symmetric] xg_def(2)
  by (fastforce simp: zip_map_fst_snd)
  then have zs_xg: zs = fo_nmlz (X  $\cup$  Y) (map xg both)
    using merge_map[where  $\sigma$ =xg and ?ns=sort (nsx @ map fst cys) and ?ms=nys] aux
  by (fastforce simp: nall(2) xg_def(3) sort_sort_both_ys)
  have proj_map: proj_tuple ns (zip nsx xs') = map xg ns proj_tuple ns (zip nsy ys') = map xf ns
    proj_tuple ns (zip nsx xs) = map xf ns proj_tuple ns (zip nsy ys) = map ysf ns
  unfolding xf_def(2) xg_def(2) xf_def ysf_def
  using assms(5,6,7,8) proj_tuple_map
  by auto
  have adAgr_list (X  $\cup$  Y) (map xg both) (map xf both)
    using zs_xg zs_xf
    by (fastforce dest: fo_nmlz_eqD)
  then have adAgr_list (X  $\cup$  Y) (proj_tuple ns (zip nsx xs')) (proj_tuple ns (zip nsy ys'))
    using assms(8)
    unfolding proj_map

```

```

    by (fastforce simp: both_def intro: ad_agr_list_subset[rotated])
    then have fo_nmlz_Un: fo_nmlz (X ∪ Y) (proj_tuple ns (zip nsx xs')) = fo_nmlz (X ∪ Y)
(proj_tuple ns (zip nsy ys'))
    by (auto intro: fo_nmlz_eqI)
    have False
    using assms(9)
    proof (rule disjE)
      assume c: fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz (X ∩ Y) (proj_tuple ns (zip
nsy ys))
      have fo_nmlz_Int: fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsx xs')) = fo_nmlz (X ∩ Y) (proj_tuple
ns (zip nsy ys'))
      using fo_nmlz_Un
      by (rule fo_nmlz_eqI[OF ad_agr_list_mono, rotated, OF fo_nmlz_eqD]) auto
      have proj_xs: fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsx xs)) = fo_nmlz (X ∩ Y) (proj_tuple ns
(zip nsx xs'))
      unfolding proj_map
      apply (rule fo_nmlz_eqI)
      apply (rule ad_agr_list_mono[OF Int_lower1])
      apply (rule ad_agr_list_subset[OF _ close(2)][unfolded xsf_def xg_def(2)])
      using assms(8)
      apply (auto)
      done
      have proj_ys: fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsy ys)) = fo_nmlz (X ∩ Y) (proj_tuple ns
(zip nsy ys'))
      unfolding proj_map
      apply (rule fo_nmlz_eqI)
      apply (rule ad_agr_list_mono[OF Int_lower2])
      apply (rule ad_agr_list_subset[OF _ close(4)][unfolded ysf_def xf_def(2)])
      using assms(8)
      apply (auto)
      done
    show False
    using c fo_nmlz_Int proj_xs proj_ys
    by auto
  next
    assume c: proj_tuple ns (zip nsx xs) ≠ proj_tuple ns (zip nsy ys) ∧
(∀ x ∈ set (proj_tuple ns (zip nsx xs)). isl x) ∧ (∀ y ∈ set (proj_tuple ns (zip nsy ys)). isl y)
    have case_x of Inl z ⇒ z ∈ X ∪ Y | Inr b ⇒ False if x ∈ set (proj_tuple ns (zip nsx xs')) for x
      using close(2) assms(1,8) c that ad_agr_list_fo_nmlzd_isl[where ?X=X and ?f=xf and
?g=xg and ?xs=nsx]
      unfolding proj_map
      unfolding xsf_def xg_def(2)
      apply (auto simp: fo_nmlzd_def split: sum.splits)
      apply (metis image_eqI subsetD vimageI)
      apply (metis subsetD sum.disc(2))
      done
    then have E1: fo_nmlz (X ∪ Y) (proj_tuple ns (zip nsx xs')) = proj_tuple ns (zip nsx xs')
      by (rule fo_nmlz_idem_isl)
    have case_y of Inl z ⇒ z ∈ X ∪ Y | Inr b ⇒ False if y ∈ set (proj_tuple ns (zip nsy ys')) for y
      using close(4) assms(2,8) c that ad_agr_list_fo_nmlzd_isl[where ?X=Y and ?f=ysf and
?g=xf and ?xs=nsy]
      unfolding proj_map
      unfolding ysf_def xf_def(2)
      apply (auto simp: fo_nmlzd_def split: sum.splits)
      apply (metis image_eqI subsetD vimageI)
      apply (metis subsetD sum.disc(2))
      done
    then have E2: fo_nmlz (X ∪ Y) (proj_tuple ns (zip nsy ys')) = proj_tuple ns (zip nsy ys')

```

```

    by (rule fo_nmlz_idem_isl)
  have ad: adAgrList X (map xsf ns) (map xg ns)
    using assms(8) close(2)[unfolded xsf_def xg_def(2)] adAgrList_subset
    by blast
  have  $\forall x \in \text{set } (\text{proj\_tuple } ns \ (\text{zip } nsx \ xs)). \text{isl } x$ 
    using c
    by auto
  then have E3: proj_tuple ns (zip nsx xs) = proj_tuple ns (zip nsx xs^)
    using assms(8)
    unfolding proj_map
    apply (induction ns)
    using adAgrList_fo_nmlzd_isl[OF close(2)[unfolded xsf_def xg_def(2)] assms(1)[unfolded
xsf_def]]
    by auto
  have  $\forall x \in \text{set } (\text{proj\_tuple } ns \ (\text{zip } nsy \ ys)). \text{isl } x$ 
    using c
    by auto
  then have E4: proj_tuple ns (zip nsy ys) = proj_tuple ns (zip nsy ys^)
    using assms(8)
    unfolding proj_map
    apply (induction ns)
    using adAgrList_fo_nmlzd_isl[OF close(4)[unfolded ysf_def xf_def(2)] assms(2)[unfolded
ysf_def]]
    by auto
  show False
    using c fo_nmlz_Un
    unfolding E1 E2 E3 E4
    by auto
  qed
}
then show ?thesis
  by (auto simp: eval_conj_tuple_def Let_def cxs_def[symmetric] nxs_def[symmetric] cys_def[symmetric]
nys_def[symmetric]
    ext_tuple_eq[OF len_sort(1)] ext_tuple_eq[OF len_sort(2)])
qed

```

**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 ( $\lambda n. n \in \text{set } nsy$ ) 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'  $\in$  adAgr\_close ((X  $\cup$  Y) - X) xs ys'  $\in$  adAgr\_close ((X  $\cup$  Y) - Y) ys*  
**shows** *eval\_conj\_tuple (X  $\cup$  Y) nsx nsy xs' ys' = {}*

**proof** –  
**have** *aux: sorted\_distinct ns set ns  $\subseteq$  set nsx  $\cap$  set nsy*  
**using** *assms(5) sorted\_filter[of id]*  
**by** *(auto simp: assms(7))*  
**show** *?thesis*  
**using** *eval\_conj\_tuple\_close\_empty2[OF assms(1–6) aux] assms(8–)*  
**by** *auto*  
**qed**

**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**  $eval\_conj\_tuple\ Z\ nsx\ nsy\ xs\ ys = \{\}$   
**using**  $eval\_conj\_tuple\_close\_empty2[OF\ assms(1-8)]\ assms(9)\ ad\_agr\_close\_empty\ assms(1-2)$   
**by** *fastforce*

**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** -

**have**  $aux: sorted\_distinct\ ns\ set\ ns \subseteq set\ nsx \cap set\ nsy$   
**using**  $assms(5)\ sorted\_filter[of\ id]$   
**by**  $(auto\ simp: assms(7))$   
**show** *?thesis*  
**using**  $eval\_conj\_tuple\_empty2[OF\ assms(1-6)\ aux]\ assms(8-)$   
**by** *auto*

**qed**

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

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

**proof**  $(induction\ xs\ arbitrary: n\ ys)$

**case**  $(Cons\ x\ xs)$

**then show** *?case*

**proof**  $(cases\ x)$

**case**  $(Inr\ b)$

**have**  $b\_le\_i: b \leq n$

**using**  $Cons(2)$

**by**  $(auto\ simp: Inr)$

**obtain**  $zs$  **where**  $ys\_def: ys = Inr\ b \# zs\ zs = filter\ (\lambda x.\ \neg isl\ x)\ xs$

**using**  $Cons(3)$

**by**  $(auto\ simp: Inr)$

**show** *?thesis*

**proof**  $(cases\ b < n)$

**case**  $True$

**then show** *?thesis*

**using**  $Cons(1)[OF\ _\ ys\_def(2),\ of\ n]\ Cons(2)$

**by**  $(auto\ simp: Inr\ ys\_def(1))$

**next**

**case**  $False$

**then show** *?thesis*

**using**  $Cons(1)[OF\ _\ ys\_def(2),\ of\ Suc\ n]\ Cons(2)$

**by**  $(auto\ simp: Inr\ ys\_def(1))$

**qed**

**qed** *auto*

**qed** *auto*

**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**  $xs \in nall\_tuples\_rec\ AD\ n\ (length\ xs)$   
**using**  $assms$

```

proof (induction xs arbitrary: n ys)
  case (Cons x xs)
  show ?case
proof (cases x)
  case (Inl a)
  have a_AD: a ∈ AD
    using Cons(4)
  by (auto simp: Inl)
  show ?thesis
    using Cons(1)[OF Cons(2)] Cons(3,4) a_AD
    by (auto simp: Inl)
next
  case (Inr b)
  obtain zs where ys_def: ys = Inr b # zs zs = filter (λx. ¬ isl x) xs
    using Cons(3)
  by (auto simp: Inr)
  show ?thesis
    using Cons(1)[OF _ ys_def(2)] Cons(2,4)
    by (fastforce simp: ys_def(1) Inr)
qed
qed auto

```

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

```

fixes AD :: 'a set
assumes nsφ'_def: nsφ' = filter (λn. n ∉ set nsφ) nsψ
  and nsψ'_def: nsψ' = filter (λn. n ∉ set nsψ) nsφ
  and Xφ_def: Xφ = fo_nmlz AD ' proj_vals Rφ nsφ
  and Xψ_def: Xψ = fo_nmlz AD ' proj_vals Rψ nsψ
  and distinct: sorted_distinct nsφ sorted_distinct nsψ
  and cxs_def: cxs = filter (λ(n, x). n ∉ set nsψ ∧ isl x) (zip nsφ xs)
  and nxs_def: nxs = map fst (filter (λ(n, x). n ∉ set nsψ ∧ ¬isl x) (zip nsφ xs))
  and cys_def: cys = filter (λ(n, y). n ∉ set nsφ ∧ isl y) (zip nsψ ys)
  and nys_def: nys = map fst (filter (λ(n, y). n ∉ set nsφ ∧ ¬isl y) (zip nsψ ys))
  and xs_ys_def: xs ∈ Xφ ys ∈ Xψ
  and σxs_def: xs = map σxs nsφ fsφ = map σxs nsφ'
  and σys_def: ys = map σys nsψ fsψ = map σys nsψ'
  and fsφ_def: fsφ ∈ nall_tuples_rec AD (card (Inr - ' set xs)) (length nsφ')
  and fsψ_def: fsψ ∈ nall_tuples_rec AD (card (Inr - ' set ys)) (length nsψ')
  and adAgr: adAgr_list AD (map σys (sort (nsψ @ nsψ'))) (map σxs (sort (nsφ @ nsφ')))
shows
  map snd (merge (zip nsφ xs) (zip nsφ' fsφ)) =
    map snd (merge (zip (sort (nsφ @ map fst cys)) (map σxs (sort (nsφ @ map fst cys))))
      (zip nys (map σxs nys))) and
  map snd (merge (zip nsφ xs) cys) = map σxs (sort (nsφ @ map fst cys)) and
  map σxs nys ∈
    nall_tuples_rec {} (card (Inr - ' set (map σxs (sort (nsφ @ map fst cys)))) (length nys)

```

**proof** –

```

have len_xs_ys: length xs = length nsφ length ys = length nsψ
  using xs_ys_def
  by (auto simp: Xφ_def Xψ_def proj_vals_def fo_nmlz_length)
have len_fsφ: length fsφ = length nsφ'
  using σxs_def(2)
  by auto
have set_nsφ': set nsφ' = set (map fst cys) ∪ set nys
  using len_xs_ys(2)
  by (auto simp: nsφ'_def cys_def nys_def dest: set_zip_leftD)
  (metis (no_types, lifting) image_eqI in_set_impl_in_set_zip1 mem_Collect_eq
    prod.sel(1) split_conv)

```

```

have  $\bigwedge x. \text{Inl } x \in \text{set } xs \cup \text{set } fs\varphi \implies x \in AD \bigwedge y. \text{Inl } y \in \text{set } ys \cup \text{set } fs\psi \implies y \in AD$ 
  using xs_ys_def fo_nmlz_set[of AD] nall_tuples_rec_Inl[OF fs\varphi_def]
  nall_tuples_rec_Inl[OF fs\psi_def]
  by (auto simp: X\varphi_def X\psi_def)
then have Inl_xs_ys:
   $\bigwedge n. n \in \text{set } ns\varphi \cup \text{set } ns\psi \implies \text{isl } (\sigma xs \ n) \longleftrightarrow (\exists x. \sigma xs \ n = \text{Inl } x \wedge x \in AD)$ 
   $\bigwedge n. n \in \text{set } ns\varphi \cup \text{set } ns\psi \implies \text{isl } (\sigma ys \ n) \longleftrightarrow (\exists y. \sigma ys \ n = \text{Inl } y \wedge y \in AD)$ 
  unfolding \sigma xs_def \sigma ys_def ns\varphi'_def ns\psi'_def
  by (auto simp: isl_def) (smt imageI mem_Collect_eq)+
have sort_sort: sort (ns\varphi @ ns\varphi') = sort (ns\psi @ ns\psi')
  apply (rule sorted_distinct_set_unique)
  using distinct
  by (auto simp: ns\varphi'_def ns\psi'_def)
have isl_iff:  $\bigwedge n. n \in \text{set } ns\varphi' \cup \text{set } ns\psi' \implies \text{isl } (\sigma xs \ n) \vee \text{isl } (\sigma ys \ n) \implies \sigma xs \ n = \sigma ys \ n$ 
  using adAgr Inl_xs_ys
  unfolding sort_sort[symmetric] adAgr_list_link[symmetric]
  unfolding ns\varphi'_def ns\psi'_def
  apply (auto simp: adAgr_sets_def)
  unfolding adEquivPair.simps
  apply (metis (no_types, lifting) UnI2 image_eqI mem_Collect_eq)
  apply (metis (no_types, lifting) UnI2 image_eqI mem_Collect_eq)
  apply (metis (no_types, lifting) UnI1 image_eqI)+
  done
have  $\bigwedge n. n \in \text{set } (\text{map } fst \ cys) \implies \text{isl } (\sigma xs \ n)$ 
   $\bigwedge n. n \in \text{set } (\text{map } fst \ cxs) \implies \text{isl } (\sigma ys \ n)$ 
  using isl_iff
  by (auto simp: cys_def ns\varphi'_def \sigma ys_def(1) cxs_def ns\psi'_def \sigma xs_def(1) set_zip)
  (metis nth_mem)+
then have Inr_sort: Inr - ' set (map \sigma xs (sort (ns\varphi @ map fst cys))) = Inr - ' set xs
  unfolding \sigma xs_def(1) \sigma ys_def(1)
  by (auto simp: zip_mapfst_snd dest: set_zip_leftD)
  (metis fst_conv image_iff sum.disc(2))+
have map_nys: map \sigma xs nys = filter (\lambda x. \neg isl x) fs\varphi
  using isl_iff[unfolded ns\varphi'_def]
  unfolding nys_def \sigma ys_def(1) \sigma xs_def(2) ns\varphi'_def filter_map
  by (induction ns\psi) force+
have map_nys_in_nall: map \sigma xs nys \in nall_tuples_rec \{\} (card (Inr - ' set xs)) (length nys)
  using nall_tuples_rec_filter[OF fs\varphi_def[folded len_fs\varphi] map_nys]
  by auto
have map_cys: map snd cys = map \sigma xs (map fst cys)
  using isl_iff
  by (auto simp: cys_def set_zip ns\varphi'_def \sigma ys_def(1)) (metis nth_mem)
show merge_xs_cys: map snd (merge (zip ns\varphi xs) cys) = map \sigma xs (sort (ns\varphi @ map fst cys))
  apply (subst zip_mapfst_snd[of cys, symmetric])
  unfolding \sigma xs_def(1) map_cys
  apply (rule merge_map)
  using distinct
  by (auto simp: cys_def \sigma ys_def sorted_filter distinct_map_filter mapfst_zip_take)
have merge_nys_prems: sorted_distinct (sort (ns\varphi @ map fst cys)) sorted_distinct nys
  set (sort (ns\varphi @ map fst cys)) \cap set nys = \{\}
  using distinct len_xs_ys(2)
  by (auto simp: cys_def nys_def distinct_map_filter sorted_filter)
  (metis eq_key_imp_eq_value mapfst_zip)
have map_snd_merge_nys: map \sigma xs (sort (sort (ns\varphi @ map fst cys) @ nys)) =
  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)))
  by (rule merge_map[OF merge_nys_prem, symmetric])
have sort_sort_nys: sort (sort (ns\varphi @ map fst cys) @ nys) = sort (ns\varphi @ ns\varphi')

```

```

apply (rule sorted_distinct_set_unique)
using distinct merge_nys_premis set_nsφ'
by (auto simp: cys_def nys_def nsφ'_def dest: set_zip_leftD)
have map_merge_fsφ: map_snd (merge (zip nsφ xs) (zip nsφ' fsφ)) = map σxs (sort (nsφ @ nsφ'))
unfolding σxs_def
apply (rule merge_map)
using distinct sorted_filter[of id]
by (auto simp: nsφ'_def)
show map_snd (merge (zip nsφ xs) (zip nsφ' fsφ)) =
  map_snd (merge (zip (sort (nsφ @ map fst cys)) (map σxs (sort (nsφ @ map fst cys))))
    (zip nys (map σxs nys)))
unfolding map_merge_fsφ map_snd_merge_nys[unfolded sort_sort_nys]
by auto
show map σxs nys ∈ nall_tuples_rec {}
  (card (Inr -' set (map σxs (sort (nsφ @ map fst cys)))) (length nys))
using map_nys_in_nall
unfolding Inr_sort[symmetric]
by auto
qed

```

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

```

fixes AD :: 'a set
assumes nsφ'_def: nsφ' = filter (λn. n ∉ set nsφ) nsψ
and nsψ'_def: nsψ' = filter (λn. n ∉ set nsψ) nsφ
and Xφ_def: Xφ = fo_nmlz AD 'proj_vals Rφ nsφ
and Xψ_def: Xψ = fo_nmlz AD 'proj_vals Rψ nsψ
and distinct: sorted_distinct nsφ sorted_distinct nsψ
and cxs_def: cxs = filter (λ(n, x). n ∉ set nsψ ∧ isl x) (zip nsφ xs)
and nxs_def: nxs = map fst (filter (λ(n, x). n ∉ set nsψ ∧ ¬isl x) (zip nsφ xs))
and cys_def: cys = filter (λ(n, y). n ∉ set nsφ ∧ isl y) (zip nsψ ys)
and nys_def: nys = map fst (filter (λ(n, y). n ∉ set nsφ ∧ ¬isl y) (zip nsψ ys))
and xs_ys_def: xs ∈ Xφ ys ∈ Xψ
and σxs_def: xs = map σxs nsφ map_snd cys = map σxs (map fst cys)
  ysψ = map σxs nys
and σys_def: ys = map σys nsψ map_snd cxs = map σys (map fst cxs)
  xsφ = map σys nxs
and fsφ_def: fsφ = map σxs nsφ'
and fsψ_def: fsψ = map σys nsψ'
and ysψ_def: map σxs nys ∈ nall_tuples_rec {}
  (card (Inr -' set (map σxs (sort (nsφ @ map fst cys)))) (length nys))
and Inl_set_AD: Inl -' (set (map_snd cxs) ∪ set xsφ) ⊆ AD
  Inl -' (set (map_snd cys) ∪ set ysψ) ⊆ AD
and ad_agr: ad_agr_list AD (map σys (sort (nsψ @ nsψ'))) (map σxs (sort (nsφ @ nsφ')))

```

**shows**

```

  map_snd (merge (zip nsφ xs) (zip nsφ' fsφ)) =
    map_snd (merge (zip (sort (nsφ @ map fst cys)) (map σxs (sort (nsφ @ map fst cys))))
      (zip nys (map σxs nys))) and
  map_snd (merge (zip nsφ xs) cys) = map σxs (sort (nsφ @ map fst cys))
  fsφ ∈ nall_tuples_rec AD (card (Inr -' set xs)) (length nsφ')

```

**proof** –

```

have len_xs_ys: length xs = length nsφ length ys = length nsψ
using xs_ys_def
by (auto simp: Xφ_def Xψ_def proj_vals_def fo_nmlz_length)
have len_fsφ: length fsφ = length nsφ'
by (auto simp: fsφ_def)
have set_ns: set nsφ' = set (map fst cys) ∪ set nys
  set nsψ' = set (map fst cxs) ∪ set nxs
using len_xs_ys

```

```

by (auto simp: nsφ'_def cys_def nys_def nsψ'_def cxs_def nxs_def dest: set_zip_leftD)
  (metis (no_types, lifting) image_eqI in_set_impl_in_set_zip1 mem_Collect_eq
    prod.sel(1) split_conv)+
then have set_σ_ns: σxs ' set nsψ' ∪ σxs ' set nsφ' ⊆ set xs ∪ set (map snd cys) ∪ set ysψ
  σys ' set nsφ' ∪ σys ' set nsψ' ⊆ set ys ∪ set (map snd cxs) ∪ set xsφ
  by (auto simp: σxs_def σys_def nsφ'_def nsψ'_def)
have Inl_sub_AD:  $\bigwedge x. \text{Inl } x \in \text{set } xs \cup \text{set } (\text{map } \text{snd } cys) \cup \text{set } ys\psi \implies x \in AD$ 
 $\bigwedge y. \text{Inl } y \in \text{set } ys \cup \text{set } (\text{map } \text{snd } cxs) \cup \text{set } xs\phi \implies y \in AD$ 
  using xs_ys_def fo_nmlz_set[of AD] Inl_set_AD
by (auto simp: Xφ_def Xψ_def) (metis in_set_zipE set_map_subset_eq vimageI zip_map_fst_snd)+
then have Inl_xs_ys:
   $\bigwedge n. n \in \text{set } ns\phi' \cup \text{set } ns\psi' \implies \text{isl } (\sigma xs \ n) \longleftrightarrow (\exists x. \sigma xs \ n = \text{Inl } x \wedge x \in AD)$ 
 $\bigwedge n. n \in \text{set } ns\phi' \cup \text{set } ns\psi' \implies \text{isl } (\sigma ys \ n) \longleftrightarrow (\exists y. \sigma ys \ n = \text{Inl } y \wedge y \in AD)$ 
  using set_σ_ns
  by (auto simp: isl_def rev_image_eqI)
have sort_sort:  $\text{sort } (ns\phi \ @ \ ns\phi') = \text{sort } (ns\psi \ @ \ ns\psi')$ 
  apply (rule sorted_distinct_set_unique)
  using distinct
  by (auto simp: nsφ'_def nsψ'_def)
have isl_iff:  $\bigwedge n. n \in \text{set } ns\phi' \cup \text{set } ns\psi' \implies \text{isl } (\sigma xs \ n) \vee \text{isl } (\sigma ys \ n) \implies \sigma xs \ n = \sigma ys \ n$ 
  using ad_agr Inl_xs_ys
  unfolding sort_sort[symmetric] ad_agr_list_link[symmetric]
  unfolding nsφ'_def nsψ'_def
  apply (auto simp: ad_agr_sets_def)
  unfolding ad_equiv_pair.simps
  apply (metis (no_types, lifting) UnI2 image_eqI mem_Collect_eq)
  apply (metis (no_types, lifting) UnI2 image_eqI mem_Collect_eq)
  apply (metis (no_types, lifting) UnI1 image_eqI)+
  done
have  $\bigwedge n. n \in \text{set } (\text{map } \text{fst } cys) \implies \text{isl } (\sigma xs \ n)$ 
 $\bigwedge n. n \in \text{set } (\text{map } \text{fst } cxs) \implies \text{isl } (\sigma ys \ n)$ 
  using isl_iff
  by (auto simp: cys_def nsφ'_def σys_def(1) cxs_def nsψ'_def σxs_def(1) set_zip)
  (metis nth_mem)+
then have Inr_sort:  $\text{Inr } - ' \text{set } (\text{map } \sigma xs \ (\text{sort } (ns\phi \ @ \ \text{map } \text{fst } cys))) = \text{Inr } - ' \text{set } xs$ 
  unfolding σxs_def(1) σys_def(1)
  by (auto simp: zip_map_fst_snd dest: set_zip_leftD)
  (metis fst_conv image_iff sum.disc(2))+
have map_nys:  $\text{map } \sigma xs \ nys = \text{filter } (\lambda x. \neg \text{isl } x) \ \text{fs}\phi$ 
  using isl_iff[unfolded nsφ'_def]
  unfolding nys_def σys_def(1) fsφ_def nsφ'_def
  by (induction nsψ) force+
have map_cys:  $\text{map } \text{snd } cys = \text{map } \sigma xs \ (\text{map } \text{fst } cys)$ 
  using isl_iff
  by (auto simp: cys_def set_zip nsφ'_def σys_def(1)) (metis nth_mem)
show merge_xs_cys:  $\text{map } \text{snd } (\text{merge } (\text{zip } ns\phi \ xs) \ cys) = \text{map } \sigma xs \ (\text{sort } (ns\phi \ @ \ \text{map } \text{fst } cys))$ 
  apply (subst zip_map_fst_snd[of cys, symmetric])
  unfolding σxs_def(1) map_cys
  apply (rule merge_map)
  using distinct
  by (auto simp: cys_def σys_def sorted_filter distinct_map_filter map_fst_zip_take)
have merge_nys_prem:  $\text{sorted\_distinct } (\text{sort } (ns\phi \ @ \ \text{map } \text{fst } cys)) \ \text{sorted\_distinct } nys$ 
 $\text{set } (\text{sort } (ns\phi \ @ \ \text{map } \text{fst } cys)) \cap \text{set } nys = \{\}$ 
  using distinct len_xs_ys(2)
  by (auto simp: cys_def nys_def distinct_map_filter sorted_filter)
  (metis eq_key_imp_eq_value map_fst_zip)
have map_snd_merge_nys:  $\text{map } \sigma xs \ (\text{sort } (ns\phi \ @ \ \text{map } \text{fst } cys) \ @ \ nys) =$ 
 $\text{map } \text{snd } (\text{merge } (\text{zip } (\text{sort } (ns\phi \ @ \ \text{map } \text{fst } cys)) \ (\text{map } \sigma xs \ (\text{sort } (ns\phi \ @ \ \text{map } \text{fst } cys))))$ 

```

```

    (zip nys (map  $\sigma xs$  nys)))
  by (rule merge_map[OF merge_nys_premis, symmetric])
have sort_sort_nys: sort (sort (ns $\varphi$  @ map fst cys) @ nys) = sort (ns $\varphi$  @ ns $\varphi'$ )
  apply (rule sorted_distinct_set_unique)
  using distinct merge_nys_premis set_ns
  by (auto simp: cys_def nys_def ns $\varphi'$ _def dest: set_zip_leftD)
have map_merge_fs $\varphi$ : map snd (merge (zip ns $\varphi$  xs) (zip ns $\varphi'$  fs $\varphi$ )) = map  $\sigma xs$  (sort (ns $\varphi$  @ ns $\varphi'$ ))
  unfolding  $\sigma xs$ _def fs $\varphi$ _def
  apply (rule merge_map)
  using distinct sorted_filter[of id]
  by (auto simp: ns $\varphi'$ _def)
show 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)))
  unfolding map_merge_fs $\varphi$  map_snd_merge_nys[unfolded sort_sort_nys]
  by auto
have Inl - ' set fs $\varphi$   $\subseteq$  AD
  using Inl_sub_AD(1) set_ $\sigma$ _ns
  by (force simp: fs $\varphi$ _def)
then show fs $\varphi$   $\in$  nall_tuples_rec AD (card (Inr - ' set xs)) (length ns $\varphi'$ )
  unfolding len_fs $\varphi$ [symmetric]
  using nall_tuples_rec_filter_rev[OF _ map_nys] ys $\psi$ _def[unfolded Inr_sort]
  by auto
qed

```

**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 $\varphi'$  X $\varphi$   $\cap$  ext_tuple_set AD ns $\psi$ 
ns $\psi'$  X $\psi$ 
proof -
  have aux: ext_tuple_set AD ns $\varphi$  ns $\varphi'$  X $\varphi$  = fo_nmlz AD '  $\bigcup$  (ext_tuple AD ns $\varphi$  ns $\varphi'$  ' X $\varphi$ )
  ext_tuple_set AD ns $\psi$  ns $\psi'$  X $\psi$  = fo_nmlz AD '  $\bigcup$  (ext_tuple AD ns $\psi$  ns $\psi'$  ' X $\psi$ )
  by (auto simp: ext_tuple_set_def ext_tuple_def X $\varphi$ _def X $\psi$ _def image_iff fo_nmlz_idem[OF
fo_nmlz_sound])
  show ?thesis
  unfolding aux
proof (rule set_eqI, rule iffI)
  fix vs
  assume vs  $\in$  fo_nmlz AD '  $\bigcup$  (ext_tuple AD ns $\varphi$  ns $\varphi'$  ' X $\varphi$ )  $\cap$ 
fo_nmlz AD '  $\bigcup$  (ext_tuple AD ns $\psi$  ns $\psi'$  ' X $\psi$ )
  then obtain xs ys where xs_ys_def: xs  $\in$  X $\varphi$  vs  $\in$  fo_nmlz AD ' ext_tuple AD ns $\varphi$  ns $\varphi'$  xs
  ys  $\in$  X $\psi$  vs  $\in$  fo_nmlz AD ' ext_tuple AD ns $\psi$  ns $\psi'$  ys
  by auto
  have len_xs_ys: length xs = length ns $\varphi$  length ys = length ns $\psi$ 
  using xs_ys_def(1,3)
  by (auto simp: X $\varphi$ _def X $\psi$ _def proj_vals_def fo_nmlz_length)
  obtain fs $\varphi$  where fs $\varphi$ _def: vs = fo_nmlz AD (map_snd (merge (zip ns $\varphi$  xs) (zip ns $\varphi'$  fs $\varphi$ )))
  fs $\varphi$   $\in$  nall_tuples_rec AD (card (Inr - ' set xs)) (length ns $\varphi'$ )
  using xs_ys_def(1,2)
  by (auto simp: X $\varphi$ _def proj_vals_def ext_tuple_def split: if_splits)
  (metis fo_nmlz_map length_map map_snd_zip)
  obtain fs $\psi$  where fs $\psi$ _def: vs = fo_nmlz AD (map_snd (merge (zip ns $\psi$  ys) (zip ns $\psi'$  fs $\psi$ )))
  fs $\psi$   $\in$  nall_tuples_rec AD (card (Inr - ' set ys)) (length ns $\psi'$ )
  using xs_ys_def(3,4)

```

```

by (auto simp: Xψ_def proj_vals_def ext_tuple_def split: if_splits)
  (metis fo_nmlz_map length_map map_snd_zip)
note len_fsφ = nall_tuples_rec_length[OF fsφ_def(2)]
note len_fψ = nall_tuples_rec_length[OF fψ_def(2)]
obtain σxs where σxs_def: xs = map σxs nsφ fsφ = map σxs nsφ'
  using exists_map[of nsφ @ nsφ' xs @ fsφ] len_xs_ys(1) len_fsφ distinct
  by (auto simp: nsφ'_def)
obtain σys where σys_def: ys = map σys nsψ fψ = map σys nsψ'
  using exists_map[of nsψ @ nsψ' ys @ fψ] len_xs_ys(2) len_fψ distinct
  by (auto simp: nsψ'_def)
have map_merge_fsφ: map_snd (merge (zip nsφ xs) (zip nsφ' fsφ)) = map σxs (sort (nsφ @ nsφ'))
  unfolding σxs_def
  apply (rule merge_map)
  using distinct_sorted_filter[of id]
  by (auto simp: nsφ'_def)
have map_merge_fψ: map_snd (merge (zip nsψ ys) (zip nsψ' fψ)) = map σys (sort (nsψ @ nsψ'))
  unfolding σys_def
  apply (rule merge_map)
  using distinct_sorted_filter[of id]
  by (auto simp: nsψ'_def)
define cxs where cxs = filter (λ(n, x). n ∉ set nsψ ∧ isl x) (zip nsφ xs)
define nxs where nxs = map fst (filter (λ(n, x). n ∉ set nsψ ∧ ¬isl x) (zip nsφ xs))
define cys where cys = filter (λ(n, y). n ∉ set nsφ ∧ isl y) (zip nsψ ys)
define nys where nys = map fst (filter (λ(n, y). n ∉ set nsφ ∧ ¬isl y) (zip nsψ ys))
note ad_agr1 = fo_nmlz_eqD[OF trans[OF fsφ_def(1)[symmetric] fψ_def(1)],
  unfolded map_merge_fsφ map_merge_fψ]
note ad_agr2 = ad_agr_list_comm[OF ad_agr1]
obtain σxs where aux1:
  map_snd (merge (zip nsφ xs) (zip nsφ' fsφ)) =
  map_snd (merge (zip (sort (nsφ @ map fst cys)) (map σxs (sort (nsφ @ map fst cys))))
  (zip nys (map σxs nys)))
  map_snd (merge (zip nsφ xs) cys) = map σxs (sort (nsφ @ map fst cys))
  map σxs nys ∈ nall_tuples_rec {}
  (card (Inr - ' set (map σxs (sort (nsφ @ map fst cys)))) (length nys)
  using eval_conj_set_aux[OF nsφ'_def nsψ'_def Xφ_def Xψ_def distinct cxs_def nxs_def
  cys_def nys_def xs_ys_def(1,3) σxs_def σys_def fsφ_def(2) fψ_def(2) ad_agr2]
  by blast
obtain σys where aux2:
  map_snd (merge (zip nsψ ys) (zip nsψ' fψ)) =
  map_snd (merge (zip (sort (nsψ @ map fst cxs)) (map σys (sort (nsψ @ map fst cxs))))
  (zip nxs (map σys nxs)))
  map_snd (merge (zip nsψ ys) cxs) = map σys (sort (nsψ @ map fst cxs))
  map σys nxs ∈ nall_tuples_rec {}
  (card (Inr - ' set (map σys (sort (nsψ @ map fst cxs)))) (length nxs)
  using eval_conj_set_aux[OF nsψ'_def nsφ'_def Xψ_def Xφ_def distinct(2,1) cys_def nys_def
  cxs_def nxs_def xs_ys_def(3,1) σys_def σxs_def fψ_def(2) fsφ_def(2) ad_agr1]
  by blast
have vs_ext_nys: vs ∈ fo_nmlz AD ' ext_tuple {} (sort (nsφ @ map fst cys)) nys
  (map_snd (merge (zip nsφ xs) cys))
  using aux1(3)
  unfolding fsφ_def(1) aux1(1)
  by (simp add: ext_tuple_eq[OF length_map[symmetric]] aux1(2))
have vs_ext_nxs: vs ∈ fo_nmlz AD ' ext_tuple {} (sort (nsψ @ map fst cxs)) nxs
  (map_snd (merge (zip nsψ ys) cxs))
  using aux2(3)
  unfolding fψ_def(1) aux2(1)
  by (simp add: ext_tuple_eq[OF length_map[symmetric]] aux2(2))
show vs ∈ eval_conj_set AD nsφ Xφ nsψ Xψ

```

```

using vs_ext_nys vs_ext_nxs xs_ys_def(1,3)
by (auto simp: eval_conj_set_def eval_conj_tuple_def nys_def cys_def nxs_def cxs_def Let_def)
next
fix vs
assume vs ∈ eval_conj_set AD nsφ Xφ nsψ Xψ
then obtain xs ys cxs nxs cys nys where
  cxs_def: cxs = filter (λ(n, x). n ∉ set nsψ ∧ isl x) (zip nsφ xs) and
  nxs_def: nxs = map fst (filter (λ(n, x). n ∉ set nsψ ∧ ¬isl x) (zip nsφ xs)) and
  cys_def: cys = filter (λ(n, y). n ∉ set nsφ ∧ isl y) (zip nsψ ys) and
  nys_def: nys = map fst (filter (λ(n, y). n ∉ set nsφ ∧ ¬isl y) (zip nsψ ys)) and
  xs_def: xs ∈ Xφ vs ∈ fo_nmlz AD ‘ ext_tuple {} (sort (nsφ @ map fst cys)) nys
  (map snd (merge (zip nsφ xs) cys)) and
  ys_def: ys ∈ Xψ vs ∈ fo_nmlz AD ‘ ext_tuple {} (sort (nsψ @ map fst cxs)) nxs
  (map snd (merge (zip nsψ ys) cxs))
  by (auto simp: eval_conj_set_def eval_conj_tuple_def Let_def) (metis (no_types, lifting) im-
age_eqI)
have len_xs_ys: length xs = length nsφ length ys = length nsψ
using xs_def(1) ys_def(1)
by (auto simp: Xφ_def Xψ_def proj_vals_def fo_nmlz_length)
have len_merge_cys: length (map snd (merge (zip nsφ xs) cys)) =
length (sort (nsφ @ map fst cys))
using merge_length[of zip nsφ xs cys] len_xs_ys
by auto
obtain ysψ where ysψ_def: vs = fo_nmlz AD (map snd (merge (zip (sort (nsφ @ map fst cys))
(map snd (merge (zip nsφ xs) cys))) (zip nys ysψ)))
ysψ ∈ nall_tuples_rec {} (card (Inr - ‘ set (map snd (merge (zip nsφ xs) cys))))
(length nys)
using xs_def(2)
unfolding ext_tuple_eq[OF len_merge_cys[symmetric]]
by auto
have distinct_nys: distinct (nsφ @ map fst cys @ nys)
using distinct len_xs_ys
by (auto simp: cys_def nys_def sorted_filter distinct_map_filter)
(metis eq_key_imp_eq_value mapfst_zip)
obtain σxs where σxs_def: xs = map σxs nsφ map snd cys = map σxs (map fst cys)
ysψ = map σxs nys
using exists_map[OF _ distinct_nys, of xs @ map snd cys @ ysψ] len_xs_ys(1)
nall_tuples_rec_length[OF ysψ_def(2)]
by (auto simp: nsφ'_def)
have len_merge_cxs: length (map snd (merge (zip nsψ ys) cxs)) =
length (sort (nsψ @ map fst cxs))
using merge_length[of zip nsψ ys] len_xs_ys
by auto
obtain xsφ where xsφ_def: vs = fo_nmlz AD (map snd (merge (zip (sort (nsψ @ map fst cxs))
(map snd (merge (zip nsψ ys) cxs))) (zip nxs xsφ)))
xsφ ∈ nall_tuples_rec {} (card (Inr - ‘ set (map snd (merge (zip nsψ ys) cxs))))
(length nxs)
using ys_def(2)
unfolding ext_tuple_eq[OF len_merge_cxs[symmetric]]
by auto
have distinct_nxs: distinct (nsψ @ map fst cxs @ nxs)
using distinct len_xs_ys(1)
by (auto simp: cxs_def nxs_def sorted_filter distinct_map_filter)
(metis eq_key_imp_eq_value mapfst_zip)
obtain σys where σys_def: ys = map σys nsψ map snd cxs = map σys (map fst cxs)
xsφ = map σys nxs
using exists_map[OF _ distinct_nxs, of ys @ map snd cxs @ xsφ] len_xs_ys(2)
nall_tuples_rec_length[OF xsφ_def(2)]

```



```

by (auto simp: nsψ'_def)
have sd_cs_ns: sorted_distinct (map fst cxs) sorted_distinct nxs
  sorted_distinct (map fst cys) sorted_distinct nys
  sorted_distinct (sort (nsψ @ map fst cxs))
  sorted_distinct (sort (nsφ @ map fst cys))
using distinct len_xs_ys
by (auto simp: cxs_def nxs_def cys_def nys_def sorted_filter distinct_map_filter)
have set_cs_ns_disj: set (map fst cxs) ∩ set nxs = {} set (map fst cys) ∩ set nys = {}
  set (sort (nsφ @ map fst cys)) ∩ set nys = {}
  set (sort (nsψ @ map fst cxs)) ∩ set nxs = {}
using distinct nth_eq_iff_index_eq
by (auto simp: cxs_def nxs_def cys_def nys_def set_zip) blast+
have merge_sort_cxs: map snd (merge (zip nsψ ys) cxs) = map σys (sort (nsψ @ map fst cxs))
  unfolding σys_def(1)
  apply (subst zip_map_fst_snd[of cxs, symmetric])
  unfolding σys_def(2)
  apply (rule merge_map)
  using distinct(2) sd_cs_ns
  by (auto simp: cxs_def)
have merge_sort_cys: map snd (merge (zip nsφ xs) cys) = map σxs (sort (nsφ @ map fst cys))
  unfolding σxs_def(1)
  apply (subst zip_map_fst_snd[of cys, symmetric])
  unfolding σxs_def(2)
  apply (rule merge_map)
  using distinct(1) sd_cs_ns
  by (auto simp: cys_def)
have set_nsφ': set nsφ' = set (map fst cys) ∪ set nys
  using len_xs_ys(2)
  by (auto simp: nsφ'_def cys_def nys_def dest: set_zip_leftD)
  (metis (no_types, lifting) image_eqI in_set_impl_in_set_zip1 mem_Collect_eq
  prod.sel(1) split_conv)
have sort_sort_nys: sort (sort (nsφ @ map fst cys) @ nys) = sort (nsφ @ nsφ')
  apply (rule sorted_distinct_set_unique)
  using distinct sd_cs_ns set_cs_ns_disj set_nsφ'
  by (auto simp: cys_def nys_def nsφ'_def dest: set_zip_leftD)
have set_nsψ': set nsψ' = set (map fst cxs) ∪ set nxs
  using len_xs_ys(1)
  by (auto simp: nsψ'_def cxs_def nxs_def dest: set_zip_leftD)
  (metis (no_types, lifting) image_eqI in_set_impl_in_set_zip1 mem_Collect_eq
  prod.sel(1) split_conv)
have sort_sort_nxs: sort (sort (nsψ @ map fst cxs) @ nxs) = sort (nsψ @ nsψ')
  apply (rule sorted_distinct_set_unique)
  using distinct sd_cs_ns set_cs_ns_disj set_nsψ'
  by (auto simp: cxs_def nxs_def nsψ'_def dest: set_zip_leftD)
have ad_agr1: ad_agr_list AD (map σys (sort (nsψ @ nsψ'))) (map σxs (sort (nsφ @ nsφ')))
  using fo_nmlz_eqD[OF trans[OF xsφ_def(1)[symmetric] ysψ_def(1)]]
  unfolding σxs_def(3) σys_def(3) merge_sort_cxs merge_sort_cys
  unfolding merge_map[OF sd_cs_ns(5) sd_cs_ns(2) set_cs_ns_disj(4)]
  unfolding merge_map[OF sd_cs_ns(6) sd_cs_ns(4) set_cs_ns_disj(3)]
  unfolding sort_sort_nxs sort_sort_nys .
note ad_agr2 = ad_agr_list_comm[OF ad_agr1]
have Inl_set_AD: Inl -' (set (map snd cxs) ∪ set xsφ) ⊆ AD
  Inl -' (set (map snd cys) ∪ set ysψ) ⊆ AD
  using xs_def(1) nall_tuples_rec_Inl[OF xsφ_def(2)] ys_def(1)
  nall_tuples_rec_Inl[OF ysψ_def(2)] fo_nmlz_set[of AD]
  by (fastforce simp: cxs_def Xφ_def cys_def Xψ_def dest!: set_zip_rightD)+
note aux1 = eval_conj_set_aux'[OF nsφ'_def nsψ'_def Xφ_def Xψ_def distinct cxs_def nxs_def
  cys_def nys_def xs_def(1) ys_def(1) σxs_def σys_def refl refl]

```

```

    ysψ_def(2)[unfolded σxs_def(3) merge_sort_cys] Inl_set_AD ad_agr1]
  note aux2 = eval_conj_set_aux'[OF nsψ'_def nsφ'_def Xψ_def Xφ_def distinct(2,1) cys_def
nys_def
    cxs_def nxs_def ys_def(1) xs_def(1) σys_def σxs_def refl refl
    xsφ_def(2)[unfolded σys_def(3) merge_sort_cxs] Inl_set_AD(2,1) ad_agr2]
  show vs ∈ fo_nmlz AD ' ⋃(ext_tuple AD nsφ nsφ' ' Xφ) ∩
fo_nmlz AD ' ⋃(ext_tuple AD nsψ nsψ' ' Xψ)
  using xs_def(1) ys_def(1) ysψ_def(1) xsφ_def(1) aux1(3) aux2(3)
    ext_tuple_eq[OF len_xs_ys(1)[symmetric], of AD nsφ]
    ext_tuple_eq[OF len_xs_ys(2)[symmetric], of AD nsψ]
  unfolding aux1(2) aux2(2) σys_def(3) σxs_def(3) aux1(1)[symmetric] aux2(1)[symmetric]
  by blast
qed
qed

```

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

**proof** (*rule iffI*)

**assume** *assms*:  $n \notin \text{fv\_fo\_fmla } \varphi \text{ esat } (\text{Exists } n \varphi) I \sigma X$

**then obtain** *x* **where**  $\text{esat } \varphi I (\sigma(n := x)) X$

**by** *auto*

**with** *assms*(1) **show**  $\text{esat } \varphi I \sigma X$

**using** *esat\_fv\_cong*[*of*  $\varphi \sigma \sigma(n := x)$ ] **by** *fastforce*

**next**

**assume** *assms*:  $n \notin \text{fv\_fo\_fmla } \varphi X \neq \{\} \text{ esat } \varphi I \sigma X$

**from** *assms*(2) **obtain** *x* **where**  $x\_def: x \in X$

**by** *auto*

**with** *assms*(1,3) **have**  $\text{esat } \varphi I (\sigma(n := x)) X$

**using** *esat\_fv\_cong*[*of*  $\varphi \sigma \sigma(n := x)$ ] **by** *fastforce*

**with** *x\_def* **show**  $\text{esat } (\text{Exists } n \varphi) I \sigma X$

**by** *auto*

**qed**

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

**using** *esat\_exists\_not\_fv*[*of*  $n \text{ Neg } \varphi X I \sigma$ ]

**by** *auto*

**lemma** *proj\_sat\_vals*:  $\text{proj\_sat } \varphi I =$

$\text{proj\_vals } \{\sigma. \text{sat } \varphi I \sigma\} (\text{fv\_fo\_fmla\_list } \varphi)$

**by** (*auto simp: proj\_sat\_def proj\_vals\_def*)

**lemma** *fv\_fo\_fmla\_list\_Pred*:  $\text{remdups\_adj } (\text{sort } (\text{fv\_fo\_terms\_list } ts)) = \text{fv\_fo\_terms\_list } ts$

**unfolding** *fv\_fo\_terms\_list\_def*

**by** (*simp add: distinct\_remdups\_adj\_sort remdups\_adj\_distinct sorted\_sort\_id*)

**lemma** *ad\_agr\_list\_fv\_list'*:  $\bigcup(\text{set } (\text{map } \text{set\_fo\_term } ts)) \subseteq X \implies$

$\text{ad\_agr\_list } X (\text{map } \sigma (\text{fv\_fo\_terms\_list } ts)) (\text{map } \tau (\text{fv\_fo\_terms\_list } ts)) \implies$

$\text{ad\_agr\_list } X (\sigma \odot e ts) (\tau \odot e ts)$

**proof** (*induction ts*)

**case** (*Cons t ts*)

**have** *IH*:  $\text{ad\_agr\_list } X (\sigma \odot e ts) (\tau \odot e ts)$

**using** *Cons*

**by** (*auto simp: ad\_agr\_list\_def ad\_equiv\_list\_link[symmetric] fv\_fo\_terms\_set\_list*

*fv\_fo\_terms\_set\_def sp\_equiv\_list\_link sp\_equiv\_def pairwise\_def*) *blast+*

**have** *ad\_equiv*:  $\bigwedge i. i \in \text{fv\_fo\_term\_set } t \cup \bigcup(\text{fv\_fo\_term\_set } ' \text{set } ts) \implies$

$\text{ad\_equiv\_pair } X (\sigma i, \tau i)$

**using** *Cons*(3)

```

by (auto simp: adAgrListDef adEquivListLink[symmetric] fvFoTermsSetList
    fvFoTermsSetDef)
have spEquiv:  $\bigwedge i j. i \in \text{fv\_fo\_term\_set } t \cup \bigcup (\text{fv\_fo\_term\_set ' set } ts) \implies$ 
 $j \in \text{fv\_fo\_term\_set } t \cup \bigcup (\text{fv\_fo\_term\_set ' set } ts) \implies \text{sp\_equiv\_pair } (\sigma i, \tau i) (\sigma j, \tau j)$ 
using Cons(3)
by (auto simp: adAgrListDef spEquivListLink fvFoTermsSetList
    fvFoTermsSetDef spEquivDef pairwiseDef)
show ?case
proof (cases t)
case (Const c)
show ?thesis
using IH Cons(2)
apply (auto simp: adAgrListDef evalEtermsDef adEquivListDef Const
    spEquivListDef pairwiseDef setZip)
unfolding adEquivPair.simps
apply (metis nthMap revImageEqI)+
done
next
case (Var n)
note tDef = Var
have ad: adEquivPair X  $(\sigma n, \tau n)$ 
using adEquiv
by (auto simp: Var)
have  $\bigwedge y. y \in \text{set } (\text{zip } (\text{map } ((\cdot)e) \sigma) ts) (\text{map } ((\cdot)e) \tau) ts) \implies y \neq (\sigma n, \tau n) \implies$ 
 $\text{sp\_equiv\_pair } (\sigma n, \tau n) y \wedge \text{sp\_equiv\_pair } y (\sigma n, \tau n)$ 
proof -
fix y
assume  $y \in \text{set } (\text{zip } (\text{map } ((\cdot)e) \sigma) ts) (\text{map } ((\cdot)e) \tau) ts)$ 
then obtain t' where yDef:  $t' \in \text{set } ts \ y = (\sigma \cdot e t', \tau \cdot e t')$ 
using nthMem
by (auto simp: setZip) blast
show spEquivPair  $(\sigma n, \tau n) y \wedge \text{sp\_equiv\_pair } y (\sigma n, \tau n)$ 
proof (cases t')
case (Const c')
have c'_X:  $c' \in X$ 
using Cons(2) yDef(1)
by (auto simp: Const) (meson SUP_le_iff foTerm.setIntros subsetD)
then show ?thesis
using adEquiv[of n] yDef(1)
unfolding yDef
apply (auto simp: Const tDef)
unfolding adEquivPair.simps
apply fastforce+
apply force
apply (metis revImageEqI)
done
next
case (Var n')
show ?thesis
using spEquiv[of n n'] yDef(1)
unfolding yDef
by (fastforce simp: tDef Var)
qed
qed
then show ?thesis
using IH Cons(3)
by (auto simp: adAgrListDef evalEtermsDef adEquivListDef Var ad spEquivListDef
    pairwiseInsert)

```

qed

qed (auto simp: eval\_eterms\_def ad\_agr\_list\_def ad\_equiv\_list\_def sp\_equiv\_list\_def)

lemma ext\_tuple\_ad\_agr\_close:

assumes  $S\varphi\_def: S\varphi \equiv \{\sigma. \text{esat } \varphi \text{ I } \sigma \text{ UNIV}\}$

and  $AD\_sub: \text{act\_edom } \varphi \text{ I} \subseteq AD\varphi \text{ AD}\varphi \subseteq AD$

and  $X\varphi\_def: X\varphi = \text{fo\_nmlz } AD\varphi \text{ 'proj\_vals } S\varphi \text{ (fv\_fo\_fmla\_list } \varphi)$

and  $ns\varphi'\_def: ns\varphi' = \text{filter } (\lambda n. n \notin \text{fv\_fo\_fmla } \varphi) \text{ ns}\psi$

and  $sd\_ns\psi: \text{sorted\_distinct } ns\psi$

and  $fv\_Un: \text{fv\_fo\_fmla } \psi = \text{fv\_fo\_fmla } \varphi \cup \text{set } ns\psi$

shows  $\text{ext\_tuple\_set } AD \text{ (fv\_fo\_fmla\_list } \varphi) \text{ ns}\varphi' \text{ (ad\_agr\_close\_set } (AD - AD\varphi) \text{ X}\varphi) =$   
 $\text{fo\_nmlz } AD \text{ 'proj\_vals } S\varphi \text{ (fv\_fo\_fmla\_list } \psi)$   
 $\text{ad\_agr\_close\_set } (AD - AD\varphi) \text{ X}\varphi = \text{fo\_nmlz } AD \text{ 'proj\_vals } S\varphi \text{ (fv\_fo\_fmla\_list } \varphi)$

proof -

have  $ad\_agr\_varphi:$

$\bigwedge \sigma \tau. \text{ad\_agr\_sets } (\text{set } (\text{fv\_fo\_fmla\_list } \varphi)) (\text{set } (\text{fv\_fo\_fmla\_list } \varphi)) \text{ AD}\varphi \sigma \tau \implies$   
 $\sigma \in S\varphi \longleftrightarrow \tau \in S\varphi$

using  $\text{esat\_UNIV\_cong}[OF \text{ ad\_agr\_sets\_restrict, OF \_subset\_refl}] \text{ ad\_agr\_sets\_mono } AD\_sub$

unfolding  $S\varphi\_def$

by blast

show  $ad\_close\_alt: \text{ad\_agr\_close\_set } (AD - AD\varphi) \text{ X}\varphi = \text{fo\_nmlz } AD \text{ 'proj\_vals } S\varphi \text{ (fv\_fo\_fmla\_list } \varphi)$

using  $\text{ad\_agr\_close\_correct}[OF \text{ AD\_sub}(2) \text{ ad\_agr\_varphi}] \text{ AD\_sub}(2)$

unfolding  $X\varphi\_def \text{ S}\varphi\_def[\text{symmetric}] \text{ proj\_fmla\_def}$

by (auto simp:  $\text{ad\_agr\_close\_set\_def} \text{ Set.is\_empty\_def}$ )

have  $fv\_varphi: \text{set } (\text{fv\_fo\_fmla\_list } \varphi) \subseteq \text{set } (\text{fv\_fo\_fmla\_list } \psi)$

using  $fv\_Un$

by (auto simp:  $\text{fv\_fo\_fmla\_list\_set}$ )

have  $sd\_ns\varphi': \text{sorted\_distinct } ns\varphi'$

using  $sd\_ns\psi \text{ sorted\_filter}[of \text{ id}]$

by (auto simp:  $ns\varphi'\_def$ )

show  $\text{ext\_tuple\_set } AD \text{ (fv\_fo\_fmla\_list } \varphi) \text{ ns}\varphi' \text{ (ad\_agr\_close\_set } (AD - AD\varphi) \text{ X}\varphi) =$   
 $\text{fo\_nmlz } AD \text{ 'proj\_vals } S\varphi \text{ (fv\_fo\_fmla\_list } \psi)$

apply (rule  $\text{ext\_tuple\_correct}$ )

using  $\text{sorted\_distinct\_fv\_list } \text{ ad\_close\_alt } \text{ ad\_agr\_varphi } \text{ ad\_agr\_sets\_mono}[OF \text{ AD\_sub}(2)]$   
 $fv\_Un \text{ sd\_ns}\varphi'$

by (fastforce simp:  $ns\varphi'\_def \text{ fv\_fo\_fmla\_list\_set}$ )

qed

lemma proj\_ext\_tuple:

assumes  $S\varphi\_def: S\varphi \equiv \{\sigma. \text{esat } \varphi \text{ I } \sigma \text{ UNIV}\}$

and  $AD\_sub: \text{act\_edom } \varphi \text{ I} \subseteq AD$

and  $X\varphi\_def: X\varphi = \text{fo\_nmlz } AD \text{ 'proj\_vals } S\varphi \text{ (fv\_fo\_fmla\_list } \varphi)$

and  $ns\varphi'\_def: ns\varphi' = \text{filter } (\lambda n. n \notin \text{fv\_fo\_fmla } \varphi) \text{ ns}\psi$

and  $sd\_ns\psi: \text{sorted\_distinct } ns\psi$

and  $fv\_Un: \text{fv\_fo\_fmla } \psi = \text{fv\_fo\_fmla } \varphi \cup \text{set } ns\psi$

and  $Z\_props: \bigwedge xs. xs \in Z \implies \text{fo\_nmlz } AD \text{ xs} = xs \wedge \text{length } xs = \text{length } (\text{fv\_fo\_fmla\_list } \psi)$

shows  $Z \cap \text{ext\_tuple\_set } AD \text{ (fv\_fo\_fmla\_list } \varphi) \text{ ns}\varphi' \text{ X}\varphi =$   
 $\{xs \in Z. \text{fo\_nmlz } AD \text{ (proj\_tuple } (\text{fv\_fo\_fmla\_list } \varphi) \text{ (zip } (\text{fv\_fo\_fmla\_list } \psi) \text{ xs}))} \in X\varphi\}$   
 $Z - \text{ext\_tuple\_set } AD \text{ (fv\_fo\_fmla\_list } \varphi) \text{ ns}\varphi' \text{ X}\varphi =$   
 $\{xs \in Z. \text{fo\_nmlz } AD \text{ (proj\_tuple } (\text{fv\_fo\_fmla\_list } \varphi) \text{ (zip } (\text{fv\_fo\_fmla\_list } \psi) \text{ xs}))} \notin X\varphi\}$

proof -

have  $ad\_agr\_varphi:$

$\bigwedge \sigma \tau. \text{ad\_agr\_sets } (\text{set } (\text{fv\_fo\_fmla\_list } \varphi)) (\text{set } (\text{fv\_fo\_fmla\_list } \varphi)) \text{ AD } \sigma \tau \implies$   
 $\sigma \in S\varphi \longleftrightarrow \tau \in S\varphi$

using  $\text{esat\_UNIV\_cong}[OF \text{ ad\_agr\_sets\_restrict, OF \_subset\_refl}] \text{ ad\_agr\_sets\_mono } AD\_sub$

unfolding  $S\varphi\_def$

by blast

```

have  $sd\_ns\varphi'$ : sorted_distinct  $ns\varphi'$ 
  using  $sd\_ns\psi$  sorted_filter[of id]
  by (auto simp:  $ns\varphi'$ _def)
have  $disj$ : set (fv_fo_fm $la\_list$   $\varphi$ )  $\cap$  set  $ns\varphi'$  = {}
  by (auto simp:  $ns\varphi'$ _def fv_fo_fm $la\_list\_set$ )
have  $Un$ : set (fv_fo_fm $la\_list$   $\varphi$ )  $\cup$  set  $ns\varphi'$  = set (fv_fo_fm $la\_list$   $\psi$ )
  using fv_Un
  by (auto simp:  $ns\varphi'$ _def fv_fo_fm $la\_list\_set$ )
note  $proj$  = proj_tuple_correct[OF sorted_distinct_fv_list  $sd\_ns\varphi'$  sorted_distinct_fv_list
  disj  $Un$   $X\varphi\_def$  ad_agr_\varphi, simplified]
have  $fo\_nmlz$  AD ‘  $X\varphi$  =  $X\varphi$ 
  using fo_nmlz_idem[OF fo_nmlz_sound]
  by (auto simp:  $X\varphi\_def$  image_iff)
then have  $aux$ : ext_tuple_set AD (fv_fo_fm $la\_list$   $\varphi$ )  $ns\varphi'$   $X\varphi$  = fo_nmlz AD ‘  $\bigcup$  (ext_tuple AD
(fv_fo_fm $la\_list$   $\varphi$ )  $ns\varphi'$  ‘  $X\varphi$ )
  by (auto simp: ext_tuple_set_def ext_tuple_def)
show  $Z \cap$  ext_tuple_set AD (fv_fo_fm $la\_list$   $\varphi$ )  $ns\varphi'$   $X\varphi$  =
  { $xs \in Z$ . fo_nmlz AD (proj_tuple (fv_fo_fm $la\_list$   $\varphi$ ) (zip (fv_fo_fm $la\_list$   $\psi$ )  $xs$ ))  $\in X\varphi$ }
  using  $Z\_props$   $proj$ 
  by (auto simp:  $aux$ )
show  $Z -$  ext_tuple_set AD (fv_fo_fm $la\_list$   $\varphi$ )  $ns\varphi'$   $X\varphi$  =
  { $xs \in Z$ . fo_nmlz AD (proj_tuple (fv_fo_fm $la\_list$   $\varphi$ ) (zip (fv_fo_fm $la\_list$   $\psi$ )  $xs$ ))  $\notin X\varphi$ }
  using  $Z\_props$   $proj$ 
  by (auto simp:  $aux$ )
qed

```

```

lemma  $fo\_nmlz\_proj\_sub$ : fo_nmlz AD ‘ proj_fm $la$   $\varphi$   $R \subseteq$  nall_tuples AD ( $nfv$   $\varphi$ )
  by (auto simp: proj_fm $la\_map$  fo_nmlz_length fo_nmlz_sound  $nfv\_def$ 
  intro: nall_tuplesI)

```

```

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

```

```

proof (rule iffI, rule ccontr)

```

```

  assume  $fin$ : finite  $Z$ 
  assume  $\neg \bigcup$  (set ‘  $Z$ )  $\subseteq$  AD
  then obtain  $\sigma$   $i$   $vs$  where  $\sigma\_def$ : map  $\sigma$  [ $0..<n$ ]  $\in Z$   $i < n$   $\sigma$   $i \notin$  AD  $vs \in X$ 
  ad_agr_list AD (map (Inl  $\circ$   $\sigma$ ) [ $0..<n$ ])  $vs$ 
  using assms(2)

```

```

  by (auto simp: assms(3) in_set_conv_nth) (metis map_map map_nth)

```

```

define  $Y$  where  $Y \equiv$  AD  $\cup$   $\sigma$  ‘ { $0..<n$ }

```

```

have  $inf\_UNIV\_Y$ : infinite (UNIV -  $Y$ )
  using assms(1)

```

```

  by (auto simp:  $Y\_def$  infinite_UNIV)
have  $\bigwedge y$ .  $y \notin Y \implies$  map (( $\lambda z$ . if  $z = \sigma$   $i$  then  $y$  else  $z$ )  $\circ$   $\sigma$ ) [ $0..<n$ ]  $\in Z$ 
  using  $\sigma\_def$ (3)

```

```

  by (auto simp: assms(3) intro!: be $I$ [OF  $\sigma\_def$ (4)] ad_agr_list_trans[OF  $\sigma\_def$ (5)])
  (auto simp: ad_agr_list_def ad_equiv_list_def set_zip  $Y\_def$  ad_equiv_pair_simps
  sp_equiv_list_def pairwise_def split: if_splits)

```

```

then have ( $\lambda x'$ . map (( $\lambda z$ . if  $z = \sigma$   $i$  then  $x'$  else  $z$ )  $\circ$   $\sigma$ ) [ $0..<n$ ]) ‘
  (UNIV -  $Y$ )  $\subseteq Z$ 

```

```

  by auto

```

```

moreover have  $inj$  ( $\lambda x'$ . map (( $\lambda z$ . if  $z = \sigma$   $i$  then  $x'$  else  $z$ )  $\circ$   $\sigma$ ) [ $0..<n$ ])
  using  $\sigma\_def$ (2)

```

```

  by (auto simp:  $inj\_def$ )

```

```

ultimately show False

```

```

    using inf_UNIV_Y fin
    by (meson inj_on_diff inj_on_finite)
next
assume  $\bigcup (set 'Z) \subseteq AD$ 
then have  $Z \subseteq all\_tuples AD n$ 
    using assms(2)
    by (auto intro: all_tuplesI)
then show finite Z
    using all_tuples_finite[OF assms(1)] finite_subset
    by auto
qed

```

```

lemma proj_out_list:
fixes AD :: ('a :: infinite) set
    and  $\sigma :: nat \Rightarrow 'a + nat$ 
    and ns :: nat list
assumes finite AD
shows  $\exists \tau. ad\_agr\_list AD (map \sigma ns) (map (Inl \circ \tau) ns) \wedge$ 
    ( $\forall j x. j \in set ns \longrightarrow \sigma j = Inl x \longrightarrow \tau j = x$ )
proof -
have fin: finite (AD  $\cup$  Inl -' set (map  $\sigma$  ns))
    using assms(1) finite_Inl[OF finite_set]
    by blast
obtain f where f_def: inj (f :: nat  $\Rightarrow$  'a)
    range f  $\subseteq UNIV - (AD \cup Inl -' set (map \sigma ns))$ 
    using arb_countable_map[OF fin]
    by auto
define  $\tau$  where  $\tau = case\_sum id f \circ \sigma$ 
have f_out:  $\bigwedge i x. i < length ns \implies \sigma (ns ! i) = Inl (f x) \implies False$ 
    using f_def(2)
    by (auto simp: vimage_def)
    (metis (no_types, lifting) DiffE UNIV_I UnCI imageI image_subset_iff mem_Collect_eq nth_mem)
have (a, b)  $\in set (zip (map \sigma ns) (map (Inl \circ \tau) ns)) \implies ad\_equiv\_pair AD (a, b)$  for a b
    using f_def(2)
    by (auto simp: set_zip  $\tau\_def$  ad_equiv_pair.simps split: sum.splits)+
moreover have sp_equiv_list (map  $\sigma$  ns) (map (Inl  $\circ$   $\tau$ ) ns)
    using f_def(1) f_out
    by (auto simp: sp_equiv_list_def pairwise_def set_zip  $\tau\_def$  inj_def split: sum.splits)+
ultimately have ad_agr_list AD (map  $\sigma$  ns) (map (Inl  $\circ$   $\tau$ ) ns)
    by (auto simp: ad_agr_list_def ad_equiv_list_def)
then show ?thesis
    by (auto simp:  $\tau\_def$  intro!: exI[of _  $\tau$ ])
qed

```

```

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  $\varphi$ ))
using proj_out_list[OF finite_act_edom[OF assms(1)], of  $\sigma$  fv_fo_fmla_list  $\varphi$ ]
    esat_UNIV_ad_agr_list[OF _ subset_refl] assms(2)
unfolding fv_fo_fmla_list_set
by fastforce

```

```

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_fmula  $\varphi$  { $\sigma$ . esat  $\varphi$  I  $\sigma$  UNIV}  $\cap$  map Inl ' UNIV =
  map Inl ' proj_fmula  $\varphi$  { $\sigma$ . sat  $\varphi$  I  $\sigma$ }
unfolding sat_esat_conv[OF wf]
proof (rule set_eqI, rule iffI)
  fix vs
  assume vs  $\in$  proj_fmula  $\varphi$  { $\sigma$ . esat  $\varphi$  I  $\sigma$  UNIV}  $\cap$  map Inl ' UNIV
  then obtain  $\sigma$  where  $\sigma\_def$ : vs = map  $\sigma$  (fv_fo_fmula_list  $\varphi$ ) esat  $\varphi$  I  $\sigma$  UNIV
    set vs  $\subseteq$  range Inl
    by (auto simp: proj_fmula_map) (metis image_subset_iff list.set_map range_eqI)
  obtain  $\tau$  where  $\tau\_def$ : esat  $\varphi$  I (Inl  $\circ$   $\tau$ ) UNIV
     $\bigwedge$  i x. i  $\in$  fv_fo_fmula  $\varphi$   $\implies$   $\sigma$  i = Inl x  $\implies$   $\tau$  i = x
    using proj_out[OF assms  $\sigma\_def$ (2)]
    by fastforce
  have vs = map (Inl  $\circ$   $\tau$ ) (fv_fo_fmula_list  $\varphi$ )
    using  $\sigma\_def$ (1,3)  $\tau\_def$ (2)
    by (auto simp: fv_fo_fmula_list_set)
  then show vs  $\in$  map Inl ' proj_fmula  $\varphi$  { $\sigma$ . esat  $\varphi$  I (Inl  $\circ$   $\sigma$ ) UNIV}
    using  $\tau\_def$ (1)
    by (force simp: proj_fmula_map)
qed (auto simp: proj_fmula_map)

lemma norm_proj_fmula_esat_sat:
  fixes  $\varphi$  :: ('a :: infinite, 'b) fo_fmula
  assumes wf_fo_intp  $\varphi$  I
  shows fo_nmlz (act_edom  $\varphi$  I) ' proj_fmula  $\varphi$  { $\sigma$ . esat  $\varphi$  I  $\sigma$  UNIV} =
    fo_nmlz (act_edom  $\varphi$  I) ' map Inl ' proj_fmula  $\varphi$  { $\sigma$ . sat  $\varphi$  I  $\sigma$ }
proof -
  have fo_nmlz (act_edom  $\varphi$  I) (map  $\sigma$  (fv_fo_fmula_list  $\varphi$ )) = fo_nmlz (act_edom  $\varphi$  I) x
    x  $\in$  ( $\lambda$  $\tau$ . map  $\tau$  (fv_fo_fmula_list  $\varphi$ )) ' { $\sigma$ . esat  $\varphi$  I  $\sigma$  UNIV}  $\cap$  range (map Inl)
    if esat  $\varphi$  I  $\sigma$  UNIV esat  $\varphi$  I (Inl  $\circ$   $\tau$ ) UNIV x = map (Inl  $\circ$   $\tau$ ) (fv_fo_fmula_list  $\varphi$ )
      ad_agr_list (act_edom  $\varphi$  I) (map  $\sigma$  (fv_fo_fmula_list  $\varphi$ )) (map (Inl  $\circ$   $\tau$ ) (fv_fo_fmula_list  $\varphi$ ))
    for  $\sigma$   $\tau$  x
    using that
    by (auto intro!: fo_nmlz_eqI) (metis map_map range_eqI)
  then show ?thesis
    unfolding proj_fmula_esat_sat[OF assms, symmetric]
    using proj_out[OF assms]
    by (fastforce simp: image_iff proj_fmula_map)
qed

lemma proj_sat_fmula: proj_sat  $\varphi$  I = proj_fmula  $\varphi$  { $\sigma$ . sat  $\varphi$  I  $\sigma$ }
  by (auto simp: proj_sat_def proj_fmula_map)

fun fo_wf :: ('a, 'b) fo_fmula  $\implies$  ('b  $\times$  nat  $\implies$  'a list set)  $\implies$  ('a, nat) fo_t  $\implies$  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  $\implies$  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
proof (rule set_eqI, rule iffI)
  fix vs
  assume vs  $\in$  fo_rep (AD, n, X)

```

```

then obtain xs where xs_def: xs ∈ X ad_agr_list AD (map Inl vs) xs
  by auto
obtain zs where zs_def: xs = map Inl zs
  using xs_def(1) assms
  by auto (meson ex_map_conv isl_def)
have set zs ⊆ AD
  using assms(1) xs_def(1) zs_def
  by (force simp: vimage_def)
then have vs_zs: vs = zs
  using xs_def(2)
  unfolding zs_def
  by (fastforce simp: ad_agr_list_def ad_equiv_list_def set_zip ad_equiv_pair.simps
    intro!: nth_equalityI)
show vs ∈ map projl ' X
  using xs_def(1) zs_def
  by (auto simp: image_iff comp_def vs_zs intro!: beXI[of _ map Inl zs])
next
fix vs
assume vs ∈ map projl ' X
then obtain xs where xs_def: xs ∈ X vs = map projl xs
  by auto
have xs_map_Inl: xs = map Inl vs
  using assms xs_def
  by (auto simp: map_idI)
show vs ∈ fo_rep (AD, n, X)
  using xs_def(1)
  by (auto simp: xs_map_Inl intro!: beXI[of _ xs] ad_agr_list_refl)
qed

definition eval_abs :: ('a, 'b) fo_fmula ⇒ ('a table, 'b) fo_intp ⇒ ('a, nat) fo_t where
  eval_abs φ I = (act_edom φ I, nfv φ, fo_nmlz (act_edom φ I) ' proj_fmula φ {σ. esat φ I σ UNIV})

lemma map_projl_Inl: map projl (map Inl xs) = xs
  by (metis (mono_tags, lifting) length_map nth_equalityI nth_map sum.sel(1))

lemma fo_rep_eval_abs:
  fixes φ :: ('a :: infinite, 'b) fo_fmula
  assumes wf_fo_intp φ I
  shows fo_rep (eval_abs φ I) = proj_sat φ I
proof -
obtain AD n X where AD_X_def: eval_abs φ I = (AD, n, X) AD = act_edom φ I
  n = nfv φ X = fo_nmlz (act_edom φ I) ' proj_fmula φ {σ. esat φ I σ UNIV}
  by (cases eval_abs φ I) (auto simp: eval_abs_def)
have AD_sub: act_edom φ I ⊆ AD
  by (auto simp: AD_X_def)
have X_def: X = fo_nmlz AD ' map Inl ' proj_fmula φ {σ. sat φ I σ}
  using AD_X_def norm_proj_fmula_esat_sat[OF assms]
  by auto
have {ts. ∃ ts' ∈ X. ad_agr_list AD (map Inl ts) ts'} = proj_fmula φ {σ. sat φ I σ}
proof (rule set_eqI, rule iffI)
fix vs
assume vs ∈ {ts. ∃ ts' ∈ X. ad_agr_list AD (map Inl ts) ts'}
then obtain vs' where vs'_def: vs' ∈ proj_fmula φ {σ. sat φ I σ}
  ad_agr_list AD (map Inl vs) (fo_nmlz AD (map Inl vs'))
  using X_def
  by auto
have length vs = length (fv_fo_fmula_list φ)
  using vs'_def

```



```

    by (auto simp: proj_fmula_map ad_agr_list_def fo_nmlz_length)
  then obtain  $\sigma$  where  $\sigma\_def: vs = \text{map } \sigma (fv\_fo\_fmula\_list \varphi)$ 
    using exists_map[of fv_fmula_list  $\varphi$  vs] sorted_distinct_fv_list
    by fastforce
  obtain  $\tau$  where  $\tau\_def: fo\_nmlz AD (\text{map } Inl vs') = \text{map } \tau (fv\_fo\_fmula\_list \varphi)$ 
    using vs'_def fo_nmlz_map
    by (fastforce simp: proj_fmula_map)
  have ad_agr: ad_agr_list AD ( $\text{map } (Inl \circ \sigma) (fv\_fo\_fmula\_list \varphi)$ ) ( $\text{map } \tau (fv\_fo\_fmula\_list \varphi)$ )
    by (metis  $\sigma\_def \tau\_def \text{map\_map } vs'_def(2)$ )
  obtain  $\tau'$  where  $\tau'\_def: \text{map } Inl vs' = \text{map } (Inl \circ \tau') (fv\_fo\_fmula\_list \varphi)$ 
    sat  $\varphi I \tau'$ 
    using vs'_def(1)
    by (fastforce simp: proj_fmula_map)
  have ad_agr': ad_agr_list AD ( $\text{map } \tau (fv\_fo\_fmula\_list \varphi)$ )
    ( $\text{map } (Inl \circ \tau') (fv\_fo\_fmula\_list \varphi)$ )
    by (rule ad_agr_list_comm) (metis fo_nmlz_ad_agr  $\tau'\_def(1) \tau\_def \text{map\_map } \text{map\_projl\_Inl}$ )
  have esat: esat  $\varphi I \tau UNIV$ 
    using esat_UNIV_ad_agr_list[OF ad_agr' AD_sub, folded sat_esat_conv[OF assms]]  $\tau'\_def(2)$ 
    by auto
  show  $vs \in \text{proj\_fmula } \varphi \{\sigma. \text{sat } \varphi I \sigma\}$ 
    using esat_UNIV_ad_agr_list[OF ad_agr AD_sub, folded sat_esat_conv[OF assms]] esat
    unfolding  $\sigma\_def$ 
    by (auto simp: proj_fmula_map)
next
fix vs
assume  $vs \in \text{proj\_fmula } \varphi \{\sigma. \text{sat } \varphi I \sigma\}$ 
then have  $vs\_X: fo\_nmlz AD (\text{map } Inl vs) \in X$ 
  using X_def
  by auto
then show  $vs \in \{ts. \exists ts' \in X. \text{ad\_agr\_list } AD (\text{map } Inl ts) ts'\}$ 
  using fo_nmlz_ad_agr
  by auto
qed
then show ?thesis
  by (auto simp: AD_X_def proj_sat_fmula)
qed

lemma fo_wf_eval_abs:
  fixes  $\varphi :: ('a :: infinite, 'b) fo\_fmula$ 
  assumes wf_fo_intp  $\varphi I$ 
  shows fo_wf  $\varphi I (eval\_abs \varphi I)$ 
  using fo_nmlz_set[of act_edom  $\varphi I$ ] finite_act_edom[OF assms(1)]
    finite_subset[OF fo_nmlz_proj_sub, OF nall_tuples_finite]
    fo_rep_eval_abs[OF assms] assms
  by (auto simp: eval_abs_def fo_nmlz_sound fo_nmlz_length nfv_def proj_sat_def proj_fmula_map)
blast

lemma fo_fin:
  fixes  $t :: ('a :: infinite, nat) fo\_t$ 
  assumes fo_wf  $\varphi I t$ 
  shows fo_fin  $t = \text{finite } (fo\_rep t)$ 
proof -
  obtain AD n X where  $t\_def: t = (AD, n, X)$ 
    using assms
    by (cases t) auto
  have fin: finite AD finite X
    using assms
    by (auto simp:  $t\_def$ )

```

```

have len_in_X:  $\bigwedge vs. vs \in X \implies \text{length } vs = n$ 
  using assms
  by (auto simp: t_def)
have Inl_X_AD:  $\bigwedge x. \text{Inl } x \in \bigcup (\text{set } ' X) \implies x \in AD$ 
  using assms
  by (fastforce simp: t_def)
define Z where Z = {ts.  $\exists ts' \in X. \text{ad\_agr\_list } AD (\text{map } \text{Inl } ts) ts'$ }
have fin_Z_iff:  $\text{finite } Z = (\bigcup (\text{set } ' Z) \subseteq AD)$ 
  using assms fin_ad_agr_list_iff[OF fin(1) _ Z_def, of n]
  by (auto simp: Z_def t_def ad_agr_list_def)
moreover have  $(\bigcup (\text{set } ' Z) \subseteq AD) \longleftrightarrow (\forall x \in \bigcup (\text{set } ' X). \text{isl } x)$ 
proof (rule iffI, rule ccontr)
  fix x
  assume Z_sub_AD:  $\bigcup (\text{set } ' Z) \subseteq AD$ 
  assume  $\neg(\forall x \in \bigcup (\text{set } ' X). \text{isl } x)$ 
  then obtain vs i m where vs_def:  $vs \in X \ i < n \ vs ! \ i = \text{Inr } m$ 
    using len_in_X
    by (auto simp: in_set_conv_nth) (metis sum.collapse(2))
  obtain  $\sigma$  where  $\sigma$ _def:  $vs = \text{map } \sigma \ [0..<n]$ 
    using exists_map[of  $[0..<n]$  vs] len_in_X[OF vs_def(1)]
    by auto
  obtain  $\tau$  where  $\tau$ _def:  $\text{ad\_agr\_list } AD \ vs (\text{map } \text{Inl } (\text{map } \tau \ [0..<n]))$ 
    using proj_out_list[OF fin(1), of  $\sigma \ [0..<n]$ ]
    by (auto simp:  $\sigma$ _def)
  have  $\text{map } \tau \ \text{in } Z$ :  $\text{map } \tau \ [0..<n] \in Z$ 
    using vs_def(1) ad_agr_list_comm[OF  $\tau$ _def]
    by (auto simp: Z_def)
  moreover have  $\tau \ i \notin AD$ 
    using  $\tau$ _def vs_def(2,3)
    apply (auto simp: ad_agr_list_def ad_equiv_list_def set_zip_comp_def  $\sigma$ _def)
    unfolding ad_equiv_pair.simps
    by (metis (no_types, lifting) Inl_Inr_False diff_zero image_iff length_upt nth_map nth_upt
      plus_nat.add_0)
  ultimately show False
    using vs_def(2) Z_sub_AD
    by fastforce
next
  assume  $\forall x \in \bigcup (\text{set } ' X). \text{isl } x$ 
  then show  $\bigcup (\text{set } ' Z) \subseteq AD$ 
    using Inl_X_AD
    apply (auto simp: Z_def ad_agr_list_def ad_equiv_list_def set_zip in_set_conv_nth)
    unfolding ad_equiv_pair.simps
    by (metis image_eqI isl_def nth_map nth_mem)
qed
ultimately show ?thesis
  by (auto simp: t_def Z_def[symmetric])
qed

```

```

lemma eval_pred:
  fixes I :: 'b  $\times$  nat  $\Rightarrow$  'a :: infinite list set
  assumes finite (I (r, length ts))
  shows fo_wf (Pred r ts) I (eval_pred ts (I (r, length ts)))
proof -
  define  $\varphi$  where  $\varphi = \text{Pred } r \ ts$ 
  have nfv_len:  $\text{nfv } \varphi = \text{length } (\text{fv\_fo\_terms\_list } ts)$ 
    by (auto simp:  $\varphi$ _def nfv_def fv_fo_fmula_list_def fv_fo_fmula_list_Pred)
  have vimage_unfold:  $\text{Inl } -' (\bigcup x \in I (r, \text{length } ts). \text{Inl } ' \text{set } x) = \bigcup (\text{set } ' I (r, \text{length } ts))$ 
    by auto

```

```

have eval_table ts (map Inl ' I (r, length ts)) ⊆ nall_tuples (act_edom φ I) (nfv φ)
by (auto simp: φ_def proj_vals_def eval_table nfv_len[unfolded φ_def]
    fo_nmlz_length fo_nmlz_sound eval_eterms_def fv_fo_terms_set_list fv_fo_terms_set_def
    vimage_unfold intro!: nall_tuplesI fo_nmlzd_all_AD dest!: fv_fo_term_setD)
    (smt UN_I Un_iff eval_eterm.simps(2) imageE image_eqI list.set_map)
then have eval: eval_pred ts (I (r, length ts)) = eval_abs φ I
by (force simp: eval_abs_def φ_def proj_fmula_def eval_pred_def eval_table fv_fo_fmula_list_def
    fv_fo_fmula_list_Pred nall_tuples_set fo_nmlz_idem nfv_len[unfolded φ_def])
have fin: wf_fo_intp (Pred r ts) I
using assms
by auto
show ?thesis
using fo_wf_eval_abs[OF fin]
by (auto simp: eval φ_def)
qed

```

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

```

proof (induction ts arbitrary: zs)
case (Cons t ts)
obtain w ws where zs_split: zs = w # ws
using Cons(3)
by (cases zs) (auto simp: adAgrListDef eval_eterms_def)
obtain τ where τ_def: ws = τ ⊙ e ts
using Cons
by (fastforce simp: zs_split adAgrListDef adEquivListDef spEquivListDef pairwise_def
    eval_eterms_def)
show ?case
proof (cases t)
case (Const c)
then show ?thesis
using Cons(3)[unfolded zs_split] Cons(2)
unfolding Const
apply (auto simp: zs_split eval_eterms_def τ_def adAgrListDef adEquivListDef)
unfolding adEquivPair.simps
by blast
next
case (Var n)
show ?thesis
proof (cases n ∈ fv_fo_terms_set ts)
case True
obtain i where i_def: i < length ts ! i = Var n
using True
by (auto simp: fv_fo_terms_set_def in_set_conv_nth dest!: fv_fo_term_setD)
have w = τ n
using Cons(3)[unfolded zs_split τ_def] i_def
using pairwiseD[of spEquivPair _ (σ n, w) (σ · e (ts ! i), τ · e (ts ! i))]
by (force simp: Var eval_eterms_def adAgrListDef spEquivListDef set_zip)
then show ?thesis
by (auto simp: Var zs_split eval_eterms_def τ_def)
next
case False
then have ws = (τ(n := w)) ⊙ e ts
using eval_eterms_cong[of ts τ τ(n := w)] τ_def
by fastforce
then show ?thesis
by (auto simp: zs_split eval_eterms_def Var fun_upd_def intro: exI[of _ τ(n := w)])
qed

```

```

qed
qed (auto simp: adAgrListDef evalEtermsDef)

lemma spEquivListFvList:
  assumes spEquivList ( $\sigma \odot e$  ts) ( $\tau \odot e$  ts)
  shows spEquivList (map  $\sigma$  (fvFoTermsList ts)) (map  $\tau$  (fvFoTermsList ts))
proof -
  have spEquivList ( $\sigma \odot e$  (map Var (fvFoTermsList ts)))
    ( $\tau \odot e$  (map Var (fvFoTermsList ts)))
  unfolding evalEtermsDef
  by (rule spEquivListSubset[OF __ assms[unfolded evalEtermsDef]])
    (auto simp: fvFoTermsSetListDest: fvFoTermsSetD)
  then show ?thesis
  by (auto simp: evalEtermsDef compDef)
qed

lemma adAgrListFvList: adAgrList X ( $\sigma \odot e$  ts) ( $\tau \odot e$  ts)  $\implies$ 
  adAgrList X (map  $\sigma$  (fvFoTermsList ts)) (map  $\tau$  (fvFoTermsList ts))
  using spEquivListFvList
  by (auto simp: evalEtermsDef adAgrListDef adEquivListDef spEquivListDef setZip)
    (metis (noTypes, opaqueLifting) evalEterm.simps(2) fvFoTermsSetD fvFoTermsSetList
      inSetConvNth nthMap)

lemma evalBool: foWf (Bool b) I (evalBool b)
  by (auto simp: evalBoolDef foNmlzdDef natsDef LetDef List.mapFilterSimps
    projSatDef fvFoFmlaListDef adAgrListDef adEquivListDef spEquivListDef nfvDef)

lemma evalEq: fixes I :: 'a  $\times$  nat  $\implies$  'a :: infinite list set
  shows foWf (Eq t t') I (evalEq t t')
proof -
  define  $\varphi$  :: ('a, 'b) foFmla where  $\varphi = \text{Eq } t \ t'$ 
  obtain AD n X where AD_X_def: evalEq t t' = (AD, n, X)
  by (cases evalEq t t') auto
  have AD_def: AD = actEdom  $\varphi$  I
  using AD_X_def
  by (auto simp: evalEqDef  $\varphi$ _def split: foTerm.splits ifSplits)
  have n_def: n = nfv  $\varphi$ 
  using AD_X_def
  by (cases t; cases t')
    (auto simp:  $\varphi$ _def fvFoFmlaListDef evalEqDef nfvDef split: ifSplits)
  have foNmlzEmpty_x x: foNmlz {x} [x, x] = [Inr 0, Inr 0] for x :: 'a + nat
  by (cases x) (auto simp: foNmlzDef)
  have Inr_0_in_foNmlz_empty: [Inr 0, Inr 0]  $\in$  foNmlz {x} ' ( $\lambda x. [x \ n', x \ n']$ ) ' { $\sigma$  :: nat  $\implies$  'a +
    nat.  $\sigma \ n = \sigma \ n'$ } for n n'
  by (auto simp: imageDef foNmlzEmpty_x_x intro!: exI[of _ [Inr 0, Inr 0]])
  have X_def: X = foNmlz AD ' projFmla  $\varphi$  { $\sigma. \text{esat } \varphi \ I \ \sigma \ \text{UNIV}$ }
  proof (rule setEqI, rule iffI)
  fix vs
  assume assm: vs  $\in$  X
  define pes where pes = projFmla  $\varphi$  { $\sigma. \text{esat } \varphi \ I \ \sigma \ \text{UNIV}$ }
  have  $\bigwedge c \ c'. \ t = \text{Const } c \ \wedge \ t' = \text{Const } c' \implies$ 
    foNmlz AD ' pes = (if c = c' then {} else {})
  by (auto simp:  $\varphi$ _def pesDef projFmla_map foNmlzDef fvFoFmlaListDef)
  moreover have  $\bigwedge c \ n. \ (t = \text{Const } c \ \wedge \ t' = \text{Var } n) \vee (t' = \text{Const } c \ \wedge \ t = \text{Var } n) \implies$ 
    foNmlz AD ' pes = {[Inl c]}
  by (auto simp:  $\varphi$ _def AD_def pesDef projFmla_map foNmlz_Cons fvFoFmlaListDef im-
    ageDef
    split: sum.splits) (auto simp: foNmlzDef)

```

```

moreover have  $\bigwedge n. t = \text{Var } n \implies t' = \text{Var } n \implies \text{fo\_nmlz } AD \text{ ' pes} = \{[Inr 0]\}$ 
  by (auto simp:  $\varphi\_def$   $AD\_def$   $pes\_def$   $proj\_fmla\_map$   $fo\_nmlz\_Cons$   $fv\_fo\_fmla\_list\_def$   $image\_def$ 
    split: sum.splits)
moreover have  $\bigwedge n n'. t = \text{Var } n \implies t' = \text{Var } n' \implies n \neq n' \implies$ 
   $\text{fo\_nmlz } AD \text{ ' pes} = \{[Inr 0, Inr 0]\}$ 
  using  $Inr\_0$  in  $fo\_nmlz\_empty$ 
by (auto simp:  $\varphi\_def$   $AD\_def$   $pes\_def$   $proj\_fmla\_map$   $fo\_nmlz\_Cons$   $fv\_fo\_fmla\_list\_def$   $fo\_nmlz\_empty\_x\_x$ 
  split: sum.splits)
ultimately show  $vs \in \text{fo\_nmlz } AD \text{ ' pes}$ 
  using  $assm$   $AD\_X\_def$ 
  by (cases  $t$ ; cases  $t'$ ) (auto simp:  $eval\_eq\_def$  split: if_splits)
next
fix  $vs$ 
assume  $assm: vs \in \text{fo\_nmlz } AD \text{ ' proj\_fmla } \varphi \{\sigma. \text{esat } \varphi I \sigma UNIV\}$ 
obtain  $\sigma$  where  $\sigma\_def: vs = \text{fo\_nmlz } AD (\text{map } \sigma (fv\_fo\_fmla\_list \varphi))$ 
   $\text{esat } (Eq t t') I \sigma UNIV$ 
  using  $assm$ 
  by (auto simp:  $\varphi\_def$   $fv\_fo\_fmla\_list\_def$   $proj\_fmla\_map$ )
show  $vs \in X$ 
  using  $\sigma\_def$   $AD\_X\_def$ 
  by (cases  $t$ ; cases  $t'$ )
  (auto simp:  $\varphi\_def$   $eval\_eq\_def$   $fv\_fo\_fmla\_list\_def$   $fo\_nmlz\_Cons$   $fo\_nmlz\_Cons\_Cons$ 
  split: sum.splits)
qed
have  $eval: eval\_eq t t' = eval\_abs \varphi I$ 
  using  $X\_def$ [ $unfolded$   $AD\_def$ ]
  by (auto simp:  $eval\_abs\_def$   $AD\_X\_def$   $AD\_def$   $n\_def$ )
have  $fin: wf\_fo\_intp \varphi I$ 
  by (auto simp:  $\varphi\_def$ )
show  $?thesis$ 
  using  $fo\_wf\_eval\_abs$ [ $OF$   $fin$ ]
  by (auto simp:  $eval \varphi\_def$ )
qed

lemma  $fv\_fo\_terms\_list\_Var: fv\_fo\_terms\_list\_rec (\text{map } Var ns) = ns$ 
  by (induction  $ns$ ) auto

lemma  $eval\_eterms\_map\_Var: \sigma \odot e \text{ map } Var ns = \text{map } \sigma ns$ 
  by (auto simp:  $eval\_eterms\_def$ )

lemma  $fo\_wf\_eval\_table:$ 
  fixes  $AD :: 'a \text{ set}$ 
  assumes  $fo\_wf \varphi I (AD, n, X)$ 
  shows  $X = \text{fo\_nmlz } AD \text{ ' eval\_table } (\text{map } Var [0..<n]) X$ 
proof -
  have  $AD\_sup: Inl -' \bigcup (\text{set ' } X) \subseteq AD$ 
  using  $assms$ 
  by  $fastforce$ 
  have  $fv: fv\_fo\_terms\_list (\text{map } Var [0..<n]) = [0..<n]$ 
  by (auto simp:  $fv\_fo\_terms\_list\_def$   $fv\_fo\_terms\_list\_Var$   $remdups\_adj\_distinct$ )
  have  $\bigwedge vs. vs \in X \implies \text{length } vs = n$ 
  using  $assms$ 
  by  $auto$ 
  then have  $X\_map: \bigwedge vs. vs \in X \implies \exists \sigma. vs = \text{map } \sigma [0..<n]$ 
  using  $exists\_map$ [ $of$   $[0..<n]$ ]
  by  $auto$ 
  then have  $proj\_vals\_X: proj\_vals \{\sigma. \sigma \odot e \text{ map } Var [0..<n] \in X\} [0..<n] = X$ 

```

```

    by (auto simp: eval_eterms_map_Var proj_vals_def)
  then show  $X = fo\_nmlz\ AD \text{ ' } eval\_table\ (map\ Var\ [0..<n])\ X$ 
    unfolding eval_table fvs proj_vals_X
    using assms fo_nmlz_idem image_iff
    by fastforce
qed

lemma fo_rep_norm:
  fixes  $AD :: ('a :: infinite)\ set$ 
  assumes fo_wf  $\varphi\ I\ (AD,\ n,\ X)$ 
  shows  $X = fo\_nmlz\ AD \text{ ' } map\ Inl \text{ ' } fo\_rep\ (AD,\ n,\ X)$ 
proof (rule set_eqI, rule iffI)
  fix vs
  assume vs_in:  $vs \in X$ 
  have fin_AD: finite AD
    using assms(1)
    by auto
  have len_vs: length vs = n
    using vs_in assms(1)
    by auto
  obtain  $\tau\_def: ad\_agr\_list\ AD\ vs\ (map\ Inl\ (map\ \tau\ [0..<n]))$ 
    using proj_out_list[OF fin_AD, of (!) vs [0..<length vs], unfolded map_nth]
    by (auto simp: len_vs)
  have map_τ_in:  $map\ \tau\ [0..<n] \in fo\_rep\ (AD,\ n,\ X)$ 
    using vs_in ad_agr_list_comm[OF τ_def]
    by auto
  have vs = fo_nmlz AD (map Inl (map τ [0..<n]))
    using fo_nmlz_eqI[OF τ_def] fo_nmlz_idem vs_in assms(1)
    by fastforce
  then show  $vs \in fo\_nmlz\ AD \text{ ' } map\ Inl \text{ ' } fo\_rep\ (AD,\ n,\ X)$ 
    using map_τ_in
    by blast
next
  fix vs
  assume vs ∈ fo_nmlz AD ' map Inl ' fo_rep (AD, n, X)
  then obtain  $xs\ xs' \text{ where } vs\_def: xs' \in X\ ad\_agr\_list\ AD\ (map\ Inl\ xs)\ xs'$ 
     $vs = fo\_nmlz\ AD\ (map\ Inl\ xs)$ 
    by auto
  then have  $vs = fo\_nmlz\ AD\ xs'$ 
    using fo_nmlz_eqI[OF vs_def(2)]
    by auto
  then have  $vs = xs'$ 
    using vs_def(1) assms(1) fo_nmlz_idem
    by fastforce
  then show  $vs \in X$ 
    using vs_def(1)
    by auto
qed

lemma fo_wf_X:
  fixes  $\varphi :: ('a :: infinite,\ 'b)\ fo\_fmla$ 
  assumes wf: fo_wf  $\varphi\ I\ (AD,\ n,\ X)$ 
  shows  $X = fo\_nmlz\ AD \text{ ' } proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$ 
proof -
  have fin: wf_fo_intp  $\varphi\ I$ 
    using wf
    by auto
  have AD_def:  $AD = act\_edom\ \varphi\ I$ 

```

```

    using wf
    by auto
  have fo_wf: fo_wf  $\varphi$  I (AD, n, X)
    using wf
    by auto
  have fo_rep: fo_rep (AD, n, X) = proj_fmula  $\varphi$  { $\sigma$ . sat  $\varphi$  I  $\sigma$ }
    using wf
    by (auto simp: proj_sat_def proj_fmula_map)
  show ?thesis
    using fo_rep_norm[OF fo_wf] norm_proj_fmula_esat_sat[OF fin]
    unfolding fo_rep AD_def[symmetric]
    by auto
qed

lemma eval_neg:
  fixes  $\varphi$  :: ('a :: infinite, 'b) fo_fmula
  assumes wf: fo_wf  $\varphi$  I t
  shows fo_wf (Neg  $\varphi$ ) I (eval_neg (fv_fo_fmula_list  $\varphi$ ) t)
proof -
  obtain AD n X where t_def: t = (AD, n, X)
    by (cases t) auto
  have eval_neg: eval_neg (fv_fo_fmula_list  $\varphi$ ) t = (AD, nfv  $\varphi$ , nall_tuples AD (nfv  $\varphi$ ) - X)
    by (auto simp: t_def nfv_def)
  have fv_unfold: fv_fo_fmula_list (Neg  $\varphi$ ) = fv_fo_fmula_list  $\varphi$ 
    by (auto simp: fv_fo_fmula_list_def)
  then have nfv_unfold: nfv (Neg  $\varphi$ ) = nfv  $\varphi$ 
    by (auto simp: nfv_def)
  have AD_def: AD = act_edom (Neg  $\varphi$ ) I
    using wf
    by (auto simp: t_def)
  note X_def = fo_wf_X[OF wf[unfolded t_def]]
  have esat_iff:  $\bigwedge vs$ . vs  $\in$  nall_tuples AD (nfv  $\varphi$ )  $\implies$ 
    vs  $\in$  fo_nmlz AD 'proj_fmula  $\varphi$  { $\sigma$ . esat  $\varphi$  I  $\sigma$  UNIV}  $\longleftrightarrow$ 
    vs  $\notin$  fo_nmlz AD 'proj_fmula  $\varphi$  { $\sigma$ . esat (Neg  $\varphi$ ) I  $\sigma$  UNIV}
  proof (rule iffI; rule ccontr)
    fix vs
    assume vs  $\in$  fo_nmlz AD 'proj_fmula  $\varphi$  { $\sigma$ . esat  $\varphi$  I  $\sigma$  UNIV}
    then obtain  $\sigma$  where  $\sigma$ _def: vs = fo_nmlz AD (map  $\sigma$  (fv_fo_fmula_list  $\varphi$ ))
      esat  $\varphi$  I  $\sigma$  UNIV
      by (auto simp: proj_fmula_map)
    assume  $\neg$ vs  $\notin$  fo_nmlz AD 'proj_fmula  $\varphi$  { $\sigma$ . esat (Neg  $\varphi$ ) I  $\sigma$  UNIV}
    then obtain  $\sigma'$  where  $\sigma'$ _def: vs = fo_nmlz AD (map  $\sigma'$  (fv_fo_fmula_list  $\varphi$ ))
      esat (Neg  $\varphi$ ) I  $\sigma'$  UNIV
      by (auto simp: proj_fmula_map)
    have esat  $\varphi$  I  $\sigma$  UNIV = esat  $\varphi$  I  $\sigma'$  UNIV
      using esat_UNIV_cong[OF ad_agr_sets_restrict[OF iffD2[OF ad_agr_list_link],
        OF fo_nmlz_eqD[OF trans[OF  $\sigma$ _def(1)[symmetric]  $\sigma'$ _def(1)]]]]
      by (auto simp: AD_def)
    then show False
      using  $\sigma$ _def(2)  $\sigma'$ _def(2) by simp
  next
    fix vs
    assume asms: vs  $\notin$  fo_nmlz AD 'proj_fmula  $\varphi$  { $\sigma$ . esat (Neg  $\varphi$ ) I  $\sigma$  UNIV}
      vs  $\notin$  fo_nmlz AD 'proj_fmula  $\varphi$  { $\sigma$ . esat  $\varphi$  I  $\sigma$  UNIV}
    assume vs  $\in$  nall_tuples AD (nfv  $\varphi$ )
    then have l_vs: length vs = length (fv_fo_fmula_list  $\varphi$ ) fo_nmlzd AD vs
      by (auto simp: nfv_def dest: nall_tuplesD)
    obtain  $\sigma$  where vs = fo_nmlz AD (map  $\sigma$  (fv_fo_fmula_list  $\varphi$ ))

```

```

    using l_vs_sorted_distinct_fv_list_exists_fo_nmlzd by metis
  with assms show False
  by (auto simp: proj_fmula_map)
qed
moreover have  $\bigwedge R. \text{fo\_nmlz } AD \text{ ' proj\_fmula } \varphi R \subseteq \text{nall\_tuples } AD \text{ (nfv } \varphi)$ 
  by (auto simp: proj_fmula_map nfv_def nall_tuplesI fo_nmlz_length fo_nmlz_sound)
ultimately have eval:  $\text{eval\_neg (fv\_fo\_fmula\_list } \varphi) t = \text{eval\_abs (Neg } \varphi) I$ 
  unfolding eval_neg eval_abs_def AD_def[symmetric]
  by (auto simp: X_def proj_fmula_def fv_unfold nfv_unfold image_subset_iff)
have wf_neg:  $\text{wf\_fo\_intp (Neg } \varphi) I$ 
  using wf
  by (auto simp: t_def)
show ?thesis
  using fo_wf_eval_abs[OF wf_neg]
  by (auto simp: eval)
qed

definition cross_with f t t' =  $\bigcup ((\lambda xs. \bigcup (f xs \text{ ' } t')) \text{ ' } t)$ 

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  $\text{set\_of\_idx (mapping\_join (cross\_with } f) (\text{cluster (Some } \circ h) t) (\text{cluster (Some } \circ h') t')) =$ 
 $\text{cross\_with } f t t'$ 
proof -
  have sub:  $\text{cross\_with } f \{y \in t. h y = h x\} \{y \in t'. h' y = h x\} \subseteq \text{cross\_with } f t t' \text{ for } t t' x$ 
  by (auto simp: cross_with_def)
  have  $\exists a. a \in h \text{ ' } t \wedge a \in h' \text{ ' } t' \wedge z \in \text{cross\_with } f \{y \in t. h y = a\} \{y \in t'. h' y = a\}$  if  $z \in$ 
 $\text{cross\_with } f t t' \text{ for } z$ 
  proof -
    obtain xs ys where wit:  $xs \in t \text{ } ys \in t' \text{ } z \in f xs ys$ 
    using z
    by (auto simp: cross_with_def)
    have h:  $h xs = h' ys$ 
    using assms(1)[OF wit(1-2)] wit(3)
    by auto
    have hys:  $h' ys \in h \text{ ' } t$ 
    using wit(1)
    by (auto simp: h[symmetric])
    show ?thesis
    apply (rule exI[of _ h xs])
    using wit hys h
    by (auto simp: cross_with_def)
  qed
  then show ?thesis
  using sub
  apply (transfer fixing: f h h')
  apply (auto simp: ran_def)
  apply fastforce+
  done
qed

lemma fo_nmlzd_mono_sub:  $X \subseteq X' \implies \text{fo\_nmlzd } X xs \implies \text{fo\_nmlzd } X' xs$ 
  by (meson fo_nmlzd_def order_trans)

lemma idx_join:
  assumes  $X\varphi\_props: \bigwedge vs. vs \in X\varphi \implies \text{fo\_nmlzd } AD vs \wedge \text{length } vs = \text{length } ns\varphi$ 
  assumes  $X\psi\_props: \bigwedge vs. vs \in X\psi \implies \text{fo\_nmlzd } AD vs \wedge \text{length } vs = \text{length } ns\psi$ 
  assumes  $sd\_ns: \text{sorted\_distinct } ns\varphi \text{ sorted\_distinct } ns\psi$ 

```



**assumes**  $ns\_def: ns = filter (\lambda n. n \in set\ ns\ \psi) \ ns\ \varphi$   
**shows**  $idx\_join\ AD\ ns\ ns\ \varphi\ X\ \psi = eval\_conj\_set\ AD\ ns\ \varphi\ X\ \psi$   
**proof** –  
**have**  $ect\_empty: x \in X\ \varphi \implies x' \in X\ \psi \implies fo\_nmlz\ AD\ (proj\_tuple\ ns\ (zip\ ns\ \varphi\ x)) \neq fo\_nmlz\ AD\ (proj\_tuple\ ns\ (zip\ ns\ \psi\ x')) \implies$   
 $eval\_conj\_tuple\ AD\ ns\ \varphi\ ns\ \psi\ x\ x' = \{\}$   
**if**  $X\ \varphi' \subseteq X\ \varphi\ X\ \psi' \subseteq X\ \psi$  **for**  $X\ \varphi'\ X\ \psi'$  **and**  $x\ x'$   
**apply**  $(rule\ eval\_conj\_tuple\_empty[where\ ?ns=filter\ (\lambda n. n \in set\ ns\ \psi)\ ns\ \varphi])$   
**using**  $X\ \varphi\_props\ X\ \psi\_props\ that\ sd\_ns$   
**by**  $(auto\ simp: ns\_def\ ad\_agr\_close\_set\_def\ split: if\_splits)$   
**have**  $cross\_eval\_conj\_tuple: (\lambda X\ \varphi''. eval\_conj\_set\ AD\ ns\ \varphi''\ ns\ \psi) = cross\_with\ (eval\_conj\_tuple\ AD\ ns\ \varphi\ ns\ \psi)$  **for**  $AD :: 'a\ set$  **and**  $ns\ \varphi\ ns\ \psi$   
**by**  $(rule\ ext)+ (auto\ simp: eval\_conj\_set\_def\ cross\_with\_def)$   
**have**  $idx\_join\ AD\ ns\ ns\ \varphi\ X\ \psi = cross\_with\ (eval\_conj\_tuple\ AD\ ns\ \varphi\ ns\ \psi)\ X\ \varphi\ X\ \psi$   
**unfolding**  $idx\_join\_def\ Let\_def\ cross\_eval\_conj\_tuple$   
**by**  $(rule\ mapping\_join\_cross\_with[OF\ ect\_empty])\ auto$   
**moreover** **have**  $\dots = eval\_conj\_set\ AD\ ns\ \varphi\ X\ \psi\ X\ \psi$   
**by**  $(auto\ simp: cross\_with\_def\ eval\_conj\_set\_def)$   
**finally** **show**  $?thesis$  .  
**qed**

**lemma**  $proj\_fmla\_conj\_sub:$

**assumes**  $AD\_sub: act\_edom\ \psi\ I \subseteq AD$   
**shows**  $fo\_nmlz\ AD\ 'proj\_fmla\ (Conj\ \varphi\ \psi)\ \{\sigma. esat\ \varphi\ I\ \sigma\ UNIV\} \cap$   
 $fo\_nmlz\ AD\ 'proj\_fmla\ (Conj\ \varphi\ \psi)\ \{\sigma. esat\ \psi\ I\ \sigma\ UNIV\} \subseteq$   
 $fo\_nmlz\ AD\ 'proj\_fmla\ (Conj\ \varphi\ \psi)\ \{\sigma. esat\ (Conj\ \varphi\ \psi)\ I\ \sigma\ UNIV\}$   
**proof**  $(rule\ subsetI)$   
**fix**  $vs$   
**assume**  $vs \in fo\_nmlz\ AD\ 'proj\_fmla\ (Conj\ \varphi\ \psi)\ \{\sigma. esat\ \varphi\ I\ \sigma\ UNIV\} \cap$   
 $fo\_nmlz\ AD\ 'proj\_fmla\ (Conj\ \varphi\ \psi)\ \{\sigma. esat\ \psi\ I\ \sigma\ UNIV\}$   
**then** **obtain**  $\sigma\ \sigma'$  **where**  $\sigma\_def:$   
 $\sigma \in \{\sigma. esat\ \varphi\ I\ \sigma\ UNIV\}\ vs = fo\_nmlz\ AD\ (map\ \sigma\ (fv\_fo\_fmla\_list\ (Conj\ \varphi\ \psi)))$   
 $\sigma' \in \{\sigma. esat\ \psi\ I\ \sigma\ UNIV\}\ vs = fo\_nmlz\ AD\ (map\ \sigma'\ (fv\_fo\_fmla\_list\ (Conj\ \varphi\ \psi)))$   
**unfolding**  $proj\_fmla\_map$   
**by**  $blast$   
**have**  $ad\_sub: act\_edom\ \psi\ I \subseteq AD$   
**using**  $assms(1)$   
**by**  $auto$   
**have**  $ad\_agr: ad\_agr\_list\ AD\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \psi))\ (map\ \sigma'\ (fv\_fo\_fmla\_list\ \psi))$   
**by**  $(rule\ ad\_agr\_list\_subset[OF\ _\ fo\_nmlz\_eqD[OF\ trans[OF\ \sigma\_def(2)[symmetric]\ \sigma\_def(4)]]])$   
 $(auto\ simp: fv\_fo\_fmla\_list\_set)$   
**have**  $\sigma \in \{\sigma. esat\ \psi\ I\ \sigma\ UNIV\}$   
**using**  $esat\_UNIV\_cong[OF\ ad\_agr\_sets\_restrict[OF\ iffD2[OF\ ad\_agr\_list\_link]],$   
 $OF\ ad\_agr\ ad\_sub]\ \sigma\_def(3)$   
**by**  $blast$   
**then** **show**  $vs \in fo\_nmlz\ AD\ 'proj\_fmla\ (Conj\ \varphi\ \psi)\ \{\sigma. esat\ (Conj\ \varphi\ \psi)\ I\ \sigma\ UNIV\}$   
**using**  $\sigma\_def(1,2)$   
**by**  $(auto\ simp: proj\_fmla\_map)$   
**qed**

**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**  $fo\_wf\ (Conj\ \varphi\ \psi)\ I\ (eval\_conj\ (fv\_fo\_fmla\_list\ \varphi)\ t\ \varphi\ (fv\_fo\_fmla\_list\ \psi)\ t\ \psi)$   
**proof** –  
**obtain**  $AD\ \varphi\ n\ \varphi\ X\ \varphi\ AD\ \psi\ n\ \psi\ X\ \psi$  **where**  $ts\_def:$   
 $t\ \varphi = (AD\ \varphi, n\ \varphi, X\ \varphi)\ t\ \psi = (AD\ \psi, n\ \psi, X\ \psi)$   
 $AD\ \varphi = act\_edom\ \varphi\ I\ AD\ \psi = act\_edom\ \psi\ I$

```

using assms
by (cases  $t\varphi$ , cases  $t\psi$ ) auto
have  $AD\_sub: act\_edom \varphi I \subseteq AD\varphi \ act\_edom \psi I \subseteq AD\psi$ 
by (auto simp: ts_def(3,4))

obtain  $AD\ n\ X$  where  $AD\_X\_def:$ 
  eval_conj (fv_fo_fmula_list  $\varphi$ )  $t\varphi$  (fv_fo_fmula_list  $\psi$ )  $t\psi = (AD, n, X)$ 
by (cases eval_conj (fv_fo_fmula_list  $\varphi$ )  $t\varphi$  (fv_fo_fmula_list  $\psi$ )  $t\psi$ ) auto
have  $AD\_def: AD = act\_edom (Conj\ \varphi\ \psi) I \ act\_edom (Conj\ \varphi\ \psi) I \subseteq AD$ 
 $AD\varphi \subseteq AD\ AD\psi \subseteq AD\ AD = AD\varphi \cup AD\psi$ 
using  $AD\_X\_def$ 
by (auto simp: ts_def Let_def)
have  $n\_def: n = nfv (Conj\ \varphi\ \psi)$ 
using  $AD\_X\_def$ 
by (auto simp: ts_def Let_def nfv_card fv_fo_fmula_list_set)

define  $S\varphi$  where  $S\varphi \equiv \{\sigma. \text{esat } \varphi\ I\ \sigma\ UNIV\}$ 
define  $S\psi$  where  $S\psi \equiv \{\sigma. \text{esat } \psi\ I\ \sigma\ UNIV\}$ 
define  $AD\Delta\varphi$  where  $AD\Delta\varphi = AD - AD\varphi$ 
define  $AD\Delta\psi$  where  $AD\Delta\psi = AD - AD\psi$ 
define  $ns\varphi$  where  $ns\varphi = fv\_fo\_fmula\_list\ \varphi$ 
define  $ns\psi$  where  $ns\psi = fv\_fo\_fmula\_list\ \psi$ 
define  $ns$  where  $ns = filter\ (\lambda n. n \in fv\_fo\_fmula\ \varphi)\ (fv\_fo\_fmula\_list\ \psi)$ 
define  $ns\varphi'$  where  $ns\varphi' = filter\ (\lambda n. n \notin fv\_fo\_fmula\ \varphi)\ (fv\_fo\_fmula\_list\ \psi)$ 
define  $ns\psi'$  where  $ns\psi' = filter\ (\lambda n. n \notin fv\_fo\_fmula\ \psi)\ (fv\_fo\_fmula\_list\ \varphi)$ 

note  $X\varphi\_def = fo\_wf\_X[OF\ wf(1)[unfolded\ ts\_def(1)],\ unfolded\ proj\_fmula\_def,\ folded\ S\varphi\_def]$ 
note  $X\psi\_def = fo\_wf\_X[OF\ wf(2)[unfolded\ ts\_def(2)],\ unfolded\ proj\_fmula\_def,\ folded\ S\psi\_def]$ 

have  $sd\_ns: sorted\_distinct\ ns\varphi\ sorted\_distinct\ ns\psi$ 
by (auto simp: ns\varphi\_def\ ns\psi\_def\ sorted\_distinct\_fv\_list)
have  $ad\_agr\_X\varphi: ad\_agr\_close\_set\ AD\Delta\varphi\ X\varphi = fo\_nmlz\ AD\ 'proj\_vals\ S\varphi\ ns\varphi$ 
unfolding  $X\varphi\_def\ ad\_agr\_close\_set\_nmlz\_eq\ ns\varphi\_def[symmetric]\ AD\Delta\varphi\_def$ 
apply (rule ad\_agr\_close\_set\_correct[OF\ AD\_def(3)\ sd\_ns(1)])
using  $AD\_sub(1)\ \text{esat\_UNIV\_ad\_agr\_list}$ 
by (fastforce simp: ad\_agr\_list\_link\ S\varphi\_def\ ns\varphi\_def)
have  $ad\_agr\_X\psi: ad\_agr\_close\_set\ AD\Delta\psi\ X\psi = fo\_nmlz\ AD\ 'proj\_vals\ S\psi\ ns\psi$ 
unfolding  $X\psi\_def\ ad\_agr\_close\_set\_nmlz\_eq\ ns\psi\_def[symmetric]\ AD\Delta\psi\_def$ 
apply (rule ad\_agr\_close\_set\_correct[OF\ AD\_def(4)\ sd\_ns(2)])
using  $AD\_sub(2)\ \text{esat\_UNIV\_ad\_agr\_list}$ 
by (fastforce simp: ad\_agr\_list\_link\ S\psi\_def\ ns\psi\_def)

have  $idx\_join\_eval\_conj: idx\_join\ AD\ (filter\ (\lambda n. n \in set\ ns\psi)\ ns\varphi)\ ns\varphi\ (ad\_agr\_close\_set\ AD\Delta\varphi\ X\varphi)\ ns\psi\ (ad\_agr\_close\_set\ AD\Delta\psi\ X\psi) =$ 
 $eval\_conj\_set\ AD\ ns\varphi\ (ad\_agr\_close\_set\ AD\Delta\varphi\ X\varphi)\ ns\psi\ (ad\_agr\_close\_set\ AD\Delta\psi\ X\psi)$ 
apply (rule idx\_join[OF\ \_ \_ sd\_ns])
unfolding  $ad\_agr\_X\varphi\ ad\_agr\_X\psi$ 
by (auto simp: fo\_nmlz\_sound\ fo\_nmlz\_length\ proj\_vals\_def)

have  $fv\_sub: fv\_fo\_fmula\ (Conj\ \varphi\ \psi) = fv\_fo\_fmula\ \varphi \cup set\ (fv\_fo\_fmula\_list\ \psi)$ 
 $fv\_fo\_fmula\ (Conj\ \varphi\ \psi) = fv\_fo\_fmula\ \psi \cup set\ (fv\_fo\_fmula\_list\ \varphi)$ 
by (auto simp: fv\_fo\_fmula\_list\_set)
note  $res\_left\_alt = ext\_tuple\_ad\_agr\_close[OF\ S\varphi\_def\ AD\_sub(1)\ AD\_def(3)\ X\varphi\_def(1)[folded\ S\varphi\_def]\ ns\varphi'\_def\ sorted\_distinct\_fv\_list\ fv\_sub(1)]$ 
note  $res\_right\_alt = ext\_tuple\_ad\_agr\_close[OF\ S\psi\_def\ AD\_sub(2)\ AD\_def(4)\ X\psi\_def(1)[folded\ S\psi\_def]\ ns\psi'\_def\ sorted\_distinct\_fv\_list\ fv\_sub(2)]$ 

note  $eval\_conj\_set = eval\_conj\_set\_correct[OF\ ns\varphi'\_def[folded\ fv\_fo\_fmula\_list\_set]]$ 

```

```

    nsψ'_def[folded fv_fo_fmula_list_set] res_left_alt(2) res_right_alt(2)
    sorted_distinct_fv_list sorted_distinct_fv_list]
have X = fo_nmlz AD ' proj_fmula (Conj φ ψ) {σ. esat φ I σ UNIV} ∩
    fo_nmlz AD ' proj_fmula (Conj φ ψ) {σ. esat ψ I σ UNIV}
using AD_X_def
apply (simp add: ts_def(1,2) Let_def ts_def(3,4)[symmetric] AD_def(5)[symmetric] idx_join_eval_conj[unfolded
nsφ_def nsψ_def ADΔφ_def ADΔψ_def])
unfolding eval_conj_set proj_fmula_def
unfolding res_left_alt(1) res_right_alt(1) Sφ_def Sψ_def
by auto
then have eval: eval_conj (fv_fo_fmula_list φ) tφ (fv_fo_fmula_list ψ) tψ =
    eval_abs (Conj φ ψ) I
using proj_fmula_conj_sub[OF AD_def(4)[unfolded ts_def(4)], of φ]
unfolding AD_X_def AD_def(1)[symmetric] n_def eval_abs_def
by (auto simp: proj_fmula_map)
have wf_conj: wf_fo_intp (Conj φ ψ) I
using wf
by (auto simp: ts_def)
show ?thesis
using fo_wf_eval_abs[OF wf_conj]
by (auto simp: eval)
qed

```

```

lemma map_values_cluster: (∧ w z Z. Z ⊆ X ⇒ z ∈ Z ⇒ w ∈ f (h z) {z} ⇒ w ∈ f (h z) Z) ⇒
    (∧ w z Z. Z ⊆ X ⇒ z ∈ Z ⇒ w ∈ f (h z) Z ⇒ (∃ z' ∈ Z. w ∈ f (h z) {z'})) ⇒
    set_of_idx (Mapping.map_values f (cluster (Some ∘ h) X)) = ∪ ((λx. f (h x) {x}) ' X)
apply transfer
apply (auto simp: ran_def)
apply (smt (verit, del_insts) mem_Collect_eq subset_eq)
apply (smt (z3) imageI mem_Collect_eq subset_iff)
done

```

```

lemma fo_nmlz_twice:
assumes sorted_distinct ns sorted_distinct ns' set ns ⊆ set ns'
shows fo_nmlz AD (proj_tuple ns (zip ns' (fo_nmlz AD (map σ ns')))) = fo_nmlz AD (map σ ns)
proof -
obtain σ' where σ': fo_nmlz AD (map σ ns') = map σ' ns'
using exists_map[where ?ys=fo_nmlz AD (map σ ns') and ?xs=ns'] assms
by (auto simp: fo_nmlz_length)
have proj: proj_tuple ns (zip ns' (map σ' ns')) = map σ' ns
by (rule proj_tuple_map[OF assms])
show ?thesis
unfolding σ' proj
apply (rule fo_nmlz_eqI)
using σ'
by (metis ad_agr_list_comm ad_agr_list_subset assms(3) fo_nmlz_ad_agr)
qed

```

```

lemma map_values_cong:
assumes ∧ x y. Mapping.lookup t x = Some y ⇒ f x y = f' x y
shows Mapping.map_values f t = Mapping.map_values f' t
proof -
have map_option (f x) (Mapping.lookup t x) = map_option (f' x) (Mapping.lookup t x) for x
using assms
by (cases Mapping.lookup t x) auto
then show ?thesis
by (auto simp: lookup_map_values intro!: mapping_eqI)
qed

```

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

**by** (*auto simp*: *ad\_agr\_close\_set\_def ad\_agr\_close\_def split: if\_splits dest: ad\_agr\_close\_rec\_length*)

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

**using** *ad\_agr\_close\_sound*[**where**  $?X=AD'$  **and**  $?Y=AD - AD'$ ]

**by** (*auto simp*: *ad\_agr\_close\_set\_def Set.is\_empty\_def split: if\_splits*) (*metis Diff\_partition Un\_Diff\_cancel*)

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

**by** (*auto simp*: *ext\_tuple\_set\_def ext\_tuple\_def fo\_nmlz\_length merge\_length dest: nall\_tuples\_rec\_length split: if\_splits*)

**lemma** *eval\_ajoin*:

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

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

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

(*eval\_ajoin* (*fv\_fo\_fmla\_list*  $\varphi$ ) *t* $\varphi$  (*fv\_fo\_fmla\_list*  $\psi$ ) *t* $\psi$ )

**proof** -

**obtain** *AD* $\varphi$  *n* $\varphi$  *X* $\varphi$  *AD* $\psi$  *n* $\psi$  *X* $\psi$  **where** *ts\_def*:

*t* $\varphi = (AD\varphi, n\varphi, X\varphi) \ t\psi = (AD\psi, n\psi, X\psi)$

*AD* $\varphi = \text{act\_edom } \varphi \ I \ AD\psi = \text{act\_edom } \psi \ I$

**using** *assms*

**by** (*cases* *t* $\varphi$ , *cases* *t* $\psi$ ) *auto*

**have** *AD\_sub*: *act\_edom*  $\varphi \ I \subseteq AD\varphi \ \text{act_edom } \psi \ I \subseteq AD\psi$

**by** (*auto simp*: *ts\_def*(3,4))

**obtain** *AD* *n* *X* **where** *AD\_X\_def*:

*eval\_ajoin* (*fv\_fo\_fmla\_list*  $\varphi$ ) *t* $\varphi$  (*fv\_fo\_fmla\_list*  $\psi$ ) *t* $\psi = (AD, n, X)$

**by** (*cases* *eval\_ajoin* (*fv\_fo\_fmla\_list*  $\varphi$ ) *t* $\varphi$  (*fv\_fo\_fmla\_list*  $\psi$ ) *t* $\psi$ ) *auto*

**have** *AD\_def*: *AD* = *act\_edom* (*Conj*  $\varphi$  (*Neg*  $\psi$ )) *I*

*act\_edom* (*Conj*  $\varphi$  (*Neg*  $\psi$ )) *I*  $\subseteq AD \ AD\varphi \subseteq AD \ AD\psi \subseteq AD \ AD = AD\varphi \cup AD\psi$

**using** *AD\_X\_def*

**by** (*auto simp*: *ts\_def Let\_def*)

**have** *n\_def*: *n* = *nfv* (*Conj*  $\varphi$  (*Neg*  $\psi$ ))

**using** *AD\_X\_def*

**by** (*auto simp*: *ts\_def Let\_def nfv\_card fv\_fo\_fmla\_list\_set*)

**define** *S* $\varphi$  **where** *S* $\varphi \equiv \{\sigma. \text{esat } \varphi \ I \ \sigma \ UNIV\}$

**define** *S* $\psi$  **where** *S* $\psi \equiv \{\sigma. \text{esat } \psi \ I \ \sigma \ UNIV\}$

**define** *both* **where** *both* = *remdups\_adj* (*sort* (*fv\_fo\_fmla\_list*  $\varphi$  @ *fv\_fo\_fmla\_list*  $\psi$ ))

**define** *ns* $\varphi'$  **where** *ns* $\varphi' = \text{filter } (\lambda n. n \notin \text{fv\_fo\_fmla } \varphi) (\text{fv\_fo\_fmla\_list } \varphi)$

**define** *ns* $\psi'$  **where** *ns* $\psi' = \text{filter } (\lambda n. n \notin \text{fv\_fo\_fmla } \psi) (\text{fv\_fo\_fmla\_list } \psi)$

**define** *AD* $\Delta\varphi$  **where** *AD* $\Delta\varphi = AD - AD\varphi$

**define** *AD* $\Delta\psi$  **where** *AD* $\Delta\psi = AD - AD\psi$

**define** *ns* $\varphi$  **where** *ns* $\varphi = \text{fv\_fo\_fmla\_list } \varphi$

**define** *ns* $\psi$  **where** *ns* $\psi = \text{fv\_fo\_fmla\_list } \psi$

**define** *ns* **where** *ns* = *filter* ( $\lambda n. n \in \text{set } ns\psi$ ) *ns* $\varphi$

**define** *X* $\varphi'$  **where** *X* $\varphi' = \text{ext\_tuple\_set } AD \ ns\varphi \ ns\varphi' \ (\text{ad\_agr\_close\_set } AD\Delta\varphi \ X\varphi)$

**define** *idx* $\varphi$  **where** *idx* $\varphi = \text{cluster } (\text{Some } \circ (\lambda xs. \text{fo\_nmlz } AD\psi \ (\text{proj\_tuple } ns \ (\text{zip } ns\varphi \ xs)))) \ (\text{ad\_agr\_close\_set } AD\Delta\varphi \ X\varphi)$

**define** *idx* $\psi$  **where** *idx* $\psi = \text{cluster } (\text{Some } \circ (\lambda ys. \text{fo\_nmlz } AD\psi \ (\text{proj\_tuple } ns \ (\text{zip } ns\psi \ ys)))) \ X\psi$

**define** *res* **where** *res* = *Mapping.map\_values* ( $\lambda xs \ X. \text{case } \text{Mapping.lookup } \text{idx}\psi \ xs \ \text{of}$

*Some* *Y*  $\implies \text{eval\_conj\_set } AD \ ns\varphi \ X \ ns\psi \ (\text{ad\_agr\_close\_set } AD\Delta\psi \ (\text{ext\_tuple\_set } AD\psi \ ns \ ns\varphi' \ \{xs\} - Y))$

```

| _ ⇒ ext_tuple_set AD nsφ nsφ' X) idφ

note Xφ_def = fo_wf_X[OF wf(1)[unfolded ts_def(1)], unfolded proj_fmld_def, folded Sφ_def]
note Xψ_def = fo_wf_X[OF wf(2)[unfolded ts_def(2)], unfolded proj_fmld_def, folded Sψ_def]

have fv_sub: fv_fo_fmld (Conj φ (Neg ψ)) = fv_fo_fmld ψ ∪ set (fv_fo_fmld_list φ)
  by (auto simp: fv_fo_fmld_list_set)
have fv_sort: fv_fo_fmld_list (Conj φ (Neg ψ)) = both
  unfolding both_def
  apply (rule sorted_distinct_set_unique)
  using sorted_distinct_fv_list
  by (auto simp: fv_fo_fmld_list_def distinct_remdups_adj_sort)

have AD_disj: ADφ ∩ ADΔφ = {} ADψ ∩ ADΔψ = {}
  by (auto simp: ADΔφ_def ADΔψ_def)
have AD_delta: AD = ADφ ∪ ADΔφ AD = ADψ ∪ ADΔψ
  by (auto simp: ADΔφ_def ADΔψ_def AD_def ts_def)
have fo_nmlzd_X: Ball Xφ (fo_nmlzd ADφ) Ball Xψ (fo_nmlzd ADψ)
  using wf
  by (auto simp: ts_def)
have Ball_ad_agr: Ball (ad_agr_close_set ADΔφ Xφ) (fo_nmlzd AD)
  using ad_agr_close_sound[where ?X=ADφ and ?Y=ADΔφ] fo_nmlzd_X(1)
  by (auto simp: ad_agr_close_set_eq[OF fo_nmlzd_X(1)] AD_disj AD_delta)
have ad_agr_φ:
  ∧σ τ. ad_agr_sets (set (fv_fo_fmld_list φ)) (set (fv_fo_fmld_list φ)) ADφ σ τ ⇒ σ ∈ Sφ ⇔
τ ∈ Sφ
  ∧σ τ. ad_agr_sets (set (fv_fo_fmld_list φ)) (set (fv_fo_fmld_list φ)) AD σ τ ⇒ σ ∈ Sφ ⇔ τ
∈ Sφ
  using esat_UNIV_cong[OF ad_agr_sets_restrict, OF _ subset_refl] ad_agr_sets_mono AD_sub(1)
  subset_trans[OF AD_sub(1) AD_def(3)]
  unfolding Sφ_def
  by blast+
have ad_agr_Sφ: τ' ∈ Sφ ⇒ ad_agr_list ADφ (map τ' nsφ) (map τ'' nsφ) ⇒ τ'' ∈ Sφ for τ' τ''
  using ad_agr_φ
  by (auto simp: ad_agr_list_link nsφ_def)
have ad_agr_ψ:
  ∧σ τ. ad_agr_sets (set (fv_fo_fmld_list ψ)) (set (fv_fo_fmld_list ψ)) ADψ σ τ ⇒ σ ∈ Sψ ⇔
τ ∈ Sψ
  using esat_UNIV_cong[OF ad_agr_sets_restrict, OF _ subset_refl] ad_agr_sets_mono[OF AD_sub(2)]
  unfolding Sψ_def
  by blast+
have ad_agr_Sψ: τ' ∈ Sψ ⇒ ad_agr_list ADψ (map τ' nsψ) (map τ'' nsψ) ⇒ τ'' ∈ Sψ for τ' τ''
  using ad_agr_ψ
  by (auto simp: ad_agr_list_link nsψ_def)
have aux: sorted_distinct nsφ sorted_distinct nsφ' sorted_distinct both set nsφ ∩ set nsφ' = {} set
nsφ ∪ set nsφ' = set both
  by (auto simp: nsφ_def nsφ'_def fv_sort[symmetric] fv_fo_fmld_list_set sorted_distinct_fv_list
intro: sorted_filter[where ?f=id, simplified])
have aux2: nsφ' = filter (λn. n ∉ set nsφ) nsφ nsφ = filter (λn. n ∉ set nsφ') nsφ
  by (auto simp: nsφ_def nsφ'_def nsψ_def nsψ'_def fv_fo_fmld_list_set)
have aux3: set nsφ' ∩ set ns = {} set nsφ' ∪ set ns = set nsψ
  by (auto simp: nsφ_def nsφ'_def nsψ_def ns_def fv_fo_fmld_list_set)
have aux4: set ns ∩ set nsφ' = {} set ns ∪ set nsφ' = set nsψ
  by (auto simp: nsφ_def nsφ'_def nsψ_def ns_def fv_fo_fmld_list_set)
have aux5: nsφ' = filter (λn. n ∉ set nsφ) nsψ nsψ' = filter (λn. n ∉ set nsψ) nsφ
  by (auto simp: nsφ_def nsφ'_def nsψ_def nsψ'_def fv_fo_fmld_list_set)
have aux6: set nsψ ∩ set nsψ' = {} set nsψ ∪ set nsψ' = set both
  by (auto simp: nsφ_def nsφ'_def nsψ_def nsψ'_def both_def fv_fo_fmld_list_set)

```

```

have ns_sd: sorted_distinct ns sorted_distinct nsφ sorted_distinct nsψ set ns ⊆ set nsφ set ns ⊆ set
nsψ set ns ⊆ set both set nsφ' ⊆ set nsψ set nsψ ⊆ set both
by (auto simp: ns_def nsφ_def nsφ'_def nsψ_def both_def sorted_distinct_fv_list intro: sorted_filter[where
?f=id, simplified])
have ns_sd': sorted_distinct nsψ'
by (auto simp: nsψ'_def sorted_distinct_fv_list intro: sorted_filter[where ?f=id, simplified])
have ns: ns = filter (λn. n ∈ fv_fo_fmula φ) (fv_fo_fmula_list ψ)
by (rule sorted_distinct_set_unique)
(auto simp: ns_def nsφ_def nsψ_def fv_fo_fmula_list_set sorted_distinct_fv_list intro: sorted_filter[where
?f=id, simplified])
have len_nsψ: length ns + length nsφ' = length nsψ
using sum_length_filter_comp[where ?P=λn. n ∈ fv_fo_fmula φ and ?xs=fv_fo_fmula_list ψ]
by (auto simp: ns nsφ_def nsφ'_def nsψ_def fv_fo_fmula_list_set)

have res_eq: res = 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
proof -
have ad_agr_Xφ: ad_agr_close_set ADΔψ Xφ = fo_nmlz AD ' proj_vals Sφ nsφ
unfolding Xφ_def ad_agr_close_set_nmlz_eq nsφ_def[symmetric]
apply (rule ad_agr_close_set_correct[OF AD_def(3) aux(1), folded ADΔψ_def])
using ad_agr_Sφ ad_agr_list_comm
by (fastforce simp: ad_agr_list_link)
have idx_eval: idx_join AD ns nsφ y nsψ (ad_agr_close_set ADΔψ (ext_tuple_set ADψ ns nsφ'
{x} - x2)) =
eval_conj_set AD nsφ y nsψ (ad_agr_close_set ADΔψ (ext_tuple_set ADψ ns nsφ' {x} - x2))
if lup: Mapping.lookup idxφ x = Some y Mapping.lookup idxψ x = Some x2 for x y x2
proof -
have vs ∈ y ⇒ fo_nmlzd AD vs ∧ length vs = length nsφ for vs
using lup(1)
by (auto simp: idxφ_def lookup_cluster' ad_agr_Xφ fo_nmlz_sound fo_nmlz_length proj_vals_def
split: if_splits)
moreover have vs ∈ ad_agr_close_set ADΔψ (ext_tuple_set ADψ ns nsφ' {x} - x2) ⇒ fo_nmlzd
AD vs for vs
apply (rule ad_agr_close_set_sound[OF __ AD_def(4), folded ADΔψ_def, where ?X=ext_tuple_set
ADψ ns nsφ' {x} - x2])
using lup(1)
by (auto simp: idxφ_def lookup_cluster' ext_tuple_set_def fo_nmlz_sound split: if_splits)
moreover have vs ∈ ad_agr_close_set ADΔψ (ext_tuple_set ADψ ns nsφ' {x} - x2) ⇒ length
vs = length nsψ for vs
apply (erule ad_agr_close_set_length)
apply (rule ext_tuple_set_length[where ?AD=ADψ and ?ns=ns and ?ns'=nsφ' and ?X={x},
unfolded len_nsψ])
using lup(1) ns_sd(1,2,4)
by (auto simp: idxφ_def lookup_cluster' fo_nmlz_length ad_agr_Xφ proj_vals_def intro!:
proj_tuple_length split: if_splits)
ultimately show ?thesis
by (auto intro!: idx_join[OF __ ns_sd(2-3) ns_def])
qed
show ?thesis
unfolding res_def
by (rule map_values_cong) (auto simp: idx_eval split: option.splits)
qed

have eval_conj: eval_conj_set AD nsφ {x} nsψ (ad_agr_close_set ADΔψ (ext_tuple_set ADψ ns
nsφ' {fo_nmlz ADψ (proj_tuple ns (zip nsφ x))} - Y)) =
ext_tuple_set AD nsφ nsφ' {x} ∩ ext_tuple_set AD nsψ nsψ' (fo_nmlz AD ' proj_vals {σ ∈ - Sψ.

```

```

adAgrList ADψ (map σ ns) (map σ' ns) nsψ
  if x_ns: proj_tuple ns (zip nsφ x) = map σ' ns
    and x_proj_singleton: {x} = fo_nmlz AD ' proj_vals {σ} nsφ
    and Some: Mapping.lookup idxψ (fo_nmlz ADψ (proj_tuple ns (zip nsφ x))) = Some Y
  for x Y σ σ'
  proof -
    have Y = {ys ∈ fo_nmlz ADψ ' proj_vals Sψ nsψ. fo_nmlz ADψ (proj_tuple ns (zip nsψ ys)) =
fo_nmlz ADψ (map σ' ns)}
      using Some
      apply (auto simp: Xψ_def idxψ_def nsψ_def x_ns lookup_cluster' split: if_splits)
      done
    moreover have ... = fo_nmlz ADψ ' proj_vals {σ ∈ Sψ. fo_nmlz ADψ (map σ ns) = fo_nmlz
ADψ (map σ' ns)} nsψ
      by (auto simp: proj_vals_def fo_nmlz_twice[OF ns_sd(1,3,5)])+
    moreover have ... = fo_nmlz ADψ ' proj_vals {σ ∈ Sψ. adAgrList ADψ (map σ ns) (map σ'
ns)} nsψ
      by (auto simp: fo_nmlz_eq)
    ultimately have Y_def: Y = fo_nmlz ADψ ' proj_vals {σ ∈ Sψ. adAgrList ADψ (map σ ns)
(map σ' ns)} nsψ
      by auto
    have R_def: {fo_nmlz ADψ (map σ' ns)} = fo_nmlz ADψ ' proj_vals {σ. adAgrList ADψ (map
σ ns) (map σ' ns)} ns
      using adAgrList_refl
      by (auto simp: proj_vals_def intro: fo_nmlz_eqI)
    have ext_tuple_set ADψ ns nsφ' {fo_nmlz ADψ (map σ' ns)} = fo_nmlz ADψ ' proj_vals {σ.
adAgrList ADψ (map σ ns) (map σ' ns)} nsψ
      apply (rule ext_tuple_correct[OF ns_sd(1) aux(2) ns_sd(3) aux4 R_def])
      using adAgrList_trans adAgrList_comm
      apply (auto simp: adAgrList_link)
      by fast
    then have ext_tuple_set ADψ ns nsφ' {fo_nmlz ADψ (map σ' ns)} - Y = fo_nmlz ADψ ' proj_vals
{σ ∈ -Sψ. adAgrList ADψ (map σ ns) (map σ' ns)} nsψ
      apply (auto simp: Y_def proj_vals_def fo_nmlz_eq)
      using adAgrList_Sψ adAgrList_comm
      by blast+
    moreover have adAgrList_close_set ADΔψ (fo_nmlz ADψ ' proj_vals {σ ∈ -Sψ. adAgrList ADψ
(map σ ns) (map σ' ns)} nsψ) =
      fo_nmlz AD ' proj_vals {σ ∈ -Sψ. adAgrList ADψ (map σ ns) (map σ' ns)} nsψ
      unfolding adAgrList_close_set_eq[OF Ball_fo_nmlzd]
      apply (rule adAgrList_close_set_correct[OF AD_def(4) ns_sd(3), folded ADΔψ_def])
      apply (auto simp: adAgrList_link)
      using adAgrList_Sψ adAgrList_comm adAgrList_subset[OF ns_sd(5)] adAgrList_trans
      by blast+
    ultimately have comp_proj: adAgrList_close_set ADΔψ (ext_tuple_set ADψ ns nsφ' {fo_nmlz ADψ
(map σ' ns)} - Y) =
      fo_nmlz AD ' proj_vals {σ ∈ -Sψ. adAgrList ADψ (map σ ns) (map σ' ns)} nsψ
      by simp
    have ext_tuple_set AD nsψ nsφ' (fo_nmlz AD ' proj_vals {σ ∈ -Sψ. adAgrList ADψ (map σ
ns) (map σ' ns)} nsψ) = fo_nmlz AD ' proj_vals {σ ∈ -Sψ. adAgrList ADψ (map σ ns) (map σ'
ns)} both
      apply (rule ext_tuple_correct[OF ns_sd(3) ns_sd'(1) aux(3) aux6 refl])
      apply (auto simp: adAgrList_link)
      using adAgrList_Sψ adAgrList_comm adAgrList_subset[OF ns_sd(5)] adAgrList_trans adAgrList_mono[OF
AD_def(4)]
      by fast+
    show eval_conj_set AD nsφ {x} nsψ (adAgrList_close_set ADΔψ (ext_tuple_set ADψ ns nsφ'
{fo_nmlz ADψ (proj_tuple ns (zip nsφ x))} - Y)) =
      ext_tuple_set AD nsφ nsφ' {x} ∩ ext_tuple_set AD nsψ nsφ' (fo_nmlz AD ' proj_vals {σ ∈ -

```

```

Sψ. adAgrList ADψ (map  $\sigma$  ns) (map  $\sigma'$  ns) nsψ
  unfolding x_ns_comp_proj
  using eval_conj_set_correct[OF aux5 x_proj_singleton refl aux(1) ns_sd(3)]
  by auto
qed

have X = set_of_idx res
  using AD_X_def
  unfolding eval_ajoin.simps ts_def(1,2) Let_def AD_def(5)[symmetric] fv_fo_fmula_list_set
  nsφ'_def[symmetric] fv_sort[symmetric] proj_fmula_def Sφ_def[symmetric] Sψ_def[symmetric]
  ADΔφ_def[symmetric] ADΔψ_def[symmetric]
  nsφ_def[symmetric] nsφ'_def[symmetric, folded fv_fo_fmula_list_set[of φ, folded nsφ_def] nsψ_def]
nsψ_def[symmetric] ns_def[symmetric]
  Xφ'_def[symmetric] idxφ_def[symmetric] idxψ_def[symmetric] res_eq[symmetric]
  by auto
  moreover have ... = ( $\bigcup x \in \text{adAgrCloseSet } AD\Delta\varphi X\varphi$ .
    case Mapping.lookup idxψ (fo_nmlz ADψ (proj_tuple ns (zip nsφ x))) of None  $\Rightarrow$  ext_tuple_set AD
nsφ nsφ' {x}
    | Some Y  $\Rightarrow$  eval_conj_set AD nsφ {x} nsψ (adAgrCloseSet ADΔψ (ext_tuple_set ADψ ns
nsφ' {fo_nmlz ADψ (proj_tuple ns (zip nsφ x))} - Y)))
  unfolding res_def[unfolded idxφ_def]
  apply (rule map_values_cluster)
  apply (auto simp: eval_conj_set_def split: option.splits)
  apply (auto simp: ext_tuple_set_def split: if_splits)
  done
  moreover have ... = fo_nmlz AD 'proj_fmula (Conj φ (Neg ψ)) {σ. esat φ I σ UNIV} -
fo_nmlz AD 'proj_fmula (Conj φ (Neg ψ)) {σ. esat ψ I σ UNIV}
  unfolding Sφ_def[symmetric] Sψ_def[symmetric] proj_fmula_def fv_sort
  proof (rule set_eqI, rule iffI)
  fix t
  assume t  $\in$  ( $\bigcup x \in \text{adAgrCloseSet } AD\Delta\varphi X\varphi$ . case Mapping.lookup idxψ (fo_nmlz ADψ (proj_tuple
ns (zip nsφ x))) of
    None  $\Rightarrow$  ext_tuple_set AD nsφ nsφ' {x}
    | Some Y  $\Rightarrow$  eval_conj_set AD nsφ {x} nsψ (adAgrCloseSet ADΔψ (ext_tuple_set ADψ ns nsφ'
{fo_nmlz ADψ (proj_tuple ns (zip nsφ x))} - Y)))
  then obtain x where x: x  $\in$  adAgrCloseSet ADΔφ Xφ
  Mapping.lookup idxψ (fo_nmlz ADψ (proj_tuple ns (zip nsφ x))) = None  $\implies$  t  $\in$  ext_tuple_set
AD nsφ nsφ' {x}
   $\wedge$  Y. Mapping.lookup idxψ (fo_nmlz ADψ (proj_tuple ns (zip nsφ x))) = Some Y  $\implies$ 
t  $\in$  eval_conj_set AD nsφ {x} nsψ (adAgrCloseSet ADΔψ (ext_tuple_set ADψ ns nsφ' {fo_nmlz
ADψ (proj_tuple ns (zip nsφ x))} - Y))
  by (fastforce split: option.splits)
  obtain  $\sigma$  where val:  $\sigma \in S\varphi$  x = fo_nmlz AD (map  $\sigma$  nsφ)
  using adAgrCloseCorrect[OF AD_def(3) adAgrφ(1), folded ADΔφ_def] Xφ_def[folded
proj_fmula_def] adAgrCloseSetEq[OF fo_nmlzd_X(1)] x(1)
  apply (auto simp: proj_fmula_def proj_vals_def nsφ_def)
  apply fast
  done
  obtain  $\sigma'$  where σ': x = map σ' nsφ
  using exists_map[where ?ys=x and ?xs=nsφ] aux(1)
  by (auto simp: val(2) fo_nmlz_length)
  have x_proj_singleton: {x} = fo_nmlz AD 'proj_vals {σ} nsφ
  by (auto simp: val(2) proj_vals_def)
  have x_ns: proj_tuple ns (zip nsφ x) = map σ' ns
  unfolding  $\sigma'$ 
  by (rule proj_tuple_map[OF ns_sd(1-2,4)])
  have adAgrσ_σ': adAgrList AD (map σ nsφ) (map σ' nsφ)
  using  $\sigma'$ 

```



```

    by (auto simp: val(2)) (metis fo_nmlz_ad_agr)
  have x_proj_ad_agr: {x} = fo_nmlz AD ' proj_vals {σ. ad_agr_list AD (map σ nsφ) (map σ'
nsφ)} nsφ
    using ad_agr_σ_σ' ad_agr_list_comm ad_agr_list_trans
  by (auto simp: val(2) proj_vals_def fo_nmlz_eq) blast
  have t ∈ fo_nmlz AD ' ⋃ (ext_tuple AD nsφ nsφ' ' {x}) ⇒ fo_nmlz AD (proj_tuple nsφ (zip both
t)) ∈ {x}
    apply (rule ext_tuple_sound(1)[OF aux x_proj_ad_agr])
    apply (auto simp: ad_agr_list_link)
    using ad_agr_list_comm ad_agr_list_trans
  by blast+
  then have x_proj: t ∈ ext_tuple_set AD nsφ nsφ' {x} ⇒ x = fo_nmlz AD (proj_tuple nsφ (zip
both t))
    using ext_tuple_set_eq[where ?AD=AD] Ball_ad_agr x(1)
  by (auto simp: val(2) proj_vals_def)
  have x_Sφ: t ∈ ext_tuple_set AD nsφ nsφ' {x} ⇒ t ∈ fo_nmlz AD ' proj_vals Sφ both
    using ext_tuple_correct[OF aux refl ad_agr_φ(2)[folded nsφ_def]] ext_tuple_set_mono[of {x}
fo_nmlz AD ' proj_vals Sφ nsφ] val(1)
  by (fastforce simp: val(2) proj_vals_def)
  show t ∈ fo_nmlz AD ' proj_vals Sφ both - fo_nmlz AD ' proj_vals Sψ both
  proof (cases Mapping.lookup idxψ (fo_nmlz ADψ (proj_tuple ns (zip nsφ x))))
  case None
  have False if t_in_Sψ: t ∈ fo_nmlz AD ' proj_vals Sψ both
  proof -
    obtain τ where τ: τ ∈ Sψ t = fo_nmlz AD (map τ both)
      using t_in_Sψ
    by (auto simp: proj_vals_def)
    obtain τ' where t_τ': t = map τ' both
      using aux(3) exists_map[where ?ys=t and ?xs=both]
    by (auto simp: τ(2) fo_nmlz_length)
    obtain τ'' where τ'': fo_nmlz ADψ (map τ nsψ) = map τ'' nsψ
      using ns_sd exists_map[where ?ys=fo_nmlz ADψ (map τ nsψ) and xs=nsψ]
    by (auto simp: fo_nmlz_length)
    have proj_τ'': proj_tuple ns (zip nsψ (map τ'' nsψ)) = map τ'' ns
      apply (rule proj_tuple_map)
      using ns_sd
    by auto
    have proj_tuple nsφ (zip both t) = map τ' nsφ
      unfolding t_τ'
      apply (rule proj_tuple_map)
      using aux
    by auto
    then have x_τ': x = fo_nmlz AD (map τ' nsφ)
      by (auto simp: x_proj[OF x(2)][OF None])
    obtain τ''' where τ''': x = map τ''' nsφ
      using aux exists_map[where ?ys=x and ?xs=nsφ]
    by (auto simp: x_τ' fo_nmlz_length)
    have ad_τ_τ': ad_agr_list AD (map τ both) (map τ' both)
      using t_τ'
    by (auto simp: τ) (metis fo_nmlz_ad_agr)
    have ad_τ_τ'': ad_agr_list ADψ (map τ nsψ) (map τ'' nsψ)
      using τ''
    by (metis fo_nmlz_ad_agr)
    have ad_τ'_τ''': ad_agr_list AD (map τ' nsφ) (map τ''' nsφ)
      using τ'''
    by (auto simp: x_τ') (metis fo_nmlz_ad_agr)
    have proj_τ''': proj_tuple ns (zip nsφ (map τ''' nsφ)) = map τ''' ns
      apply (rule proj_tuple_map)

```

```

    using aux ns_sd
    by auto
  have fo_nmlz ADψ (proj_tuple ns (zip nsφ x)) = fo_nmlz ADψ (proj_tuple ns (zip nsψ (fo_nmlz
ADψ (map τ nsψ))))
    unfolding τ'' proj_τ'' τ''' proj_τ'''
    apply (rule fo_nmlz_eqI)
    using ad_agr_list_trans ad_agr_list_subset ns_sd(4-6) ad_agr_list_mono[OF AD_def(4)]
ad_agr_list_comm[OF ad_τ'_τ''] ad_agr_list_comm[OF ad_τ_τ'] ad_τ_τ''
    by metis
  then show ?thesis
    using None τ(1)
    by (auto simp: idxψ_def lookup_cluster' Xψ_def nsψ_def[symmetric] proj_vals_def split:
if_splits)
  qed
  then show ?thesis
    using x_Sφ[OF x(2)[OF None]]
    by auto
  next
  case (Some Y)
  have t_in: t ∈ ext_tuple_set AD nsφ nsφ' {x} t ∈ ext_tuple_set AD nsψ nsψ' (fo_nmlz AD '
proj_vals {σ ∈ - Sψ. ad_agr_list ADψ (map σ ns) (map σ' ns)} nsψ)
    using x(3)[OF Some] eval_conj[OF x_ns x_proj_singleton Some]
    by auto
  have ext_tuple_set AD nsψ nsψ' (fo_nmlz AD ' proj_vals {σ ∈ - Sψ. ad_agr_list ADψ (map σ
ns) (map σ' ns)} nsψ) = fo_nmlz AD ' proj_vals {σ ∈ - Sψ. ad_agr_list ADψ (map σ ns) (map σ'
ns)} both
    apply (rule ext_tuple_correct[OF ns_sd(3) ns_sd'(1) aux(3) aux6 refl])
    apply (auto simp: ad_agr_list_link)
    using ad_agr_Sψ ad_agr_list_comm ad_agr_list_subset[OF ns_sd(5)] ad_agr_list_trans
ad_agr_list_mono[OF AD_def(4)]
    by fast+
  then have t_both: t ∈ fo_nmlz AD ' proj_vals {σ ∈ - Sψ. ad_agr_list ADψ (map σ ns) (map σ'
ns)} both
    using t_in(2)
    by auto
  {
  assume t ∈ fo_nmlz AD ' proj_vals Sψ both
  then obtain τ where τ: τ ∈ Sψ t = fo_nmlz AD (map τ both)
    by (auto simp: proj_vals_def)
  obtain τ' where τ': τ' ∉ Sψ t = fo_nmlz AD (map τ' both)
    using t_both
    by (auto simp: proj_vals_def)
  have False
    using τ τ'
    apply (auto simp: fo_nmlz_eq)
    using ad_agr_Sψ ad_agr_list_comm ad_agr_list_subset[OF ns_sd(8)] ad_agr_list_mono[OF
AD_def(4)]
    by blast
  }
  then show ?thesis
    using x_Sφ[OF t_in(1)]
    by auto
  qed
  next
  fix t
  assume t_in_asm: t ∈ fo_nmlz AD ' proj_vals Sφ both - fo_nmlz AD ' proj_vals Sψ both
  then obtain σ where val: σ ∈ Sφ t = fo_nmlz AD (map σ both)
    by (auto simp: proj_vals_def)

```

```

define  $x$  where  $x = fo\_nmlz\ AD\ (map\ \sigma\ ns\varphi)$ 
obtain  $\sigma'$  where  $\sigma': x = map\ \sigma'\ ns\varphi$ 
  using  $exists\_map[where\ ?ys=x\ and\ ?xs=ns\varphi]\ aux(1)$ 
  by ( $auto\ simp: x\_def\ fo\_nmlz\_length$ )
have  $x\_proj\_singleton: \{x\} = fo\_nmlz\ AD\ 'proj\_vals\ \{\sigma\}\ ns\varphi$ 
  by ( $auto\ simp: x\_def\ proj\_vals\_def$ )
have  $x\_in\_ad\_agr\_close: x \in ad\_agr\_close\_set\ AD\Delta\varphi\ X\varphi$ 
  using  $ad\_agr\_close\_correct[OF\ AD\_def(3)\ ad\_agr\_varphi(1),\ folded\ AD\Delta\varphi\_def]\ val(1)$ 
  unfolding  $ad\_agr\_close\_set\_eq[OF\ fo\_nmlzd\_X(1)]\ x\_def$ 
  unfolding  $X\varphi\_def[folded\ proj\_fmla\_def]\ proj\_fmla\_map$ 
  by ( $fastforce\ simp: x\_def\ ns\varphi\_def$ )
have  $ad\_agr\_sigma\_sigma': ad\_agr\_list\ AD\ (map\ \sigma\ ns\varphi)\ (map\ \sigma'\ ns\varphi)$ 
  using  $\sigma'$ 
  by ( $auto\ simp: x\_def$ ) ( $metis\ fo\_nmlz\_ad\_agr$ )
  have  $x\_proj\_ad\_agr: \{x\} = fo\_nmlz\ AD\ 'proj\_vals\ \{\sigma.\ ad\_agr\_list\ AD\ (map\ \sigma\ ns\varphi)\ (map\ \sigma'\ ns\varphi)\}\ ns\varphi$ 
  using  $ad\_agr\_sigma\_sigma'\ ad\_agr\_list\_comm\ ad\_agr\_list\_trans$ 
  by ( $auto\ simp: x\_def\ proj\_vals\_def\ fo\_nmlz\_eq$ )  $blast+$ 
have  $x\_ns: proj\_tuple\ ns\ (zip\ ns\varphi\ x) = map\ \sigma'\ ns$ 
  unfolding  $\sigma'$ 
  by ( $rule\ proj\_tuple\_map[OF\ ns\_sd(1-2,4)]$ )
have  $ext\_tuple\_set\ AD\ ns\varphi\ ns\varphi'\ \{x\} = fo\_nmlz\ AD\ 'proj\_vals\ \{\sigma.\ ad\_agr\_list\ AD\ (map\ \sigma\ ns\varphi)\ (map\ \sigma'\ ns\varphi)\}\ both$ 
  apply ( $rule\ ext\_tuple\_correct[OF\ aux\ x\_proj\_ad\_agr]$ )
  using  $ad\_agr\_list\_comm\ ad\_agr\_list\_trans$ 
  by ( $auto\ simp: ad\_agr\_list\_link$ )  $blast+$ 
then have  $t\_in\_ext\_x: t \in ext\_tuple\_set\ AD\ ns\varphi\ ns\varphi'\ \{x\}$ 
  using  $ad\_agr\_sigma\_sigma'$ 
  by ( $auto\ simp: val(2)\ proj\_vals\_def$ )
  {
    fix  $Y$ 
    assume  $Some: Mapping.lookup\ id\ x\psi\ (fo\_nmlz\ AD\psi\ (map\ \sigma'\ ns)) = Some\ Y$ 
    have  $tmp: proj\_tuple\ ns\ (zip\ ns\varphi\ x) = map\ \sigma'\ ns$ 
      unfolding  $\sigma'$ 
      by ( $rule\ proj\_tuple\_map[OF\ ns\_sd(1)\ aux(1)\ ns\_sd(4)]$ )
    have  $unfold: ext\_tuple\_set\ AD\ ns\psi\ ns\psi'\ (fo\_nmlz\ AD\ 'proj\_vals\ \{\sigma \in -\ S\psi.\ ad\_agr\_list\ AD\psi\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns)\}\ ns\psi) =$ 
       $fo\_nmlz\ AD\ 'proj\_vals\ \{\sigma \in -\ S\psi.\ ad\_agr\_list\ AD\psi\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns)\}\ both$ 
      apply ( $rule\ ext\_tuple\_correct[OF\ ns\_sd(3)\ ns\_sd'(1)\ aux(3)\ aux6\ refl]$ )
      apply ( $auto\ simp: ad\_agr\_list\_link$ )
      using  $ad\_agr\_S\psi\ ad\_agr\_list\_mono[OF\ AD\_def(4)]\ ad\_agr\_list\_comm\ ad\_agr\_list\_trans$ 
       $ad\_agr\_list\_subset[OF\ ns\_sd(5)]$ 
      by  $blast+$ 
    have  $\sigma \notin S\psi$ 
      using  $t\_in\_asm$ 
      by ( $auto\ simp: val(2)\ proj\_vals\_def$ )
    moreover have  $ad\_agr\_list\ AD\psi\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns)$ 
      using  $ad\_agr\_sigma\_sigma'\ ad\_agr\_list\_mono[OF\ AD\_def(4)]\ ad\_agr\_list\_subset[OF\ ns\_sd(4)]$ 
      by  $blast$ 
    ultimately have  $t \in ext\_tuple\_set\ AD\ ns\psi\ ns\psi'\ (fo\_nmlz\ AD\ 'proj\_vals\ \{\sigma \in -\ S\psi.\ ad\_agr\_list\ AD\psi\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns)\}\ ns\psi)$ 
      unfolding  $unfold\ val(2)$ 
      by ( $auto\ simp: proj\_vals\_def$ )
    then have  $t \in eval\_conj\_set\ AD\ ns\varphi\ \{x\}\ ns\psi\ (ad\_agr\_close\_set\ AD\Delta\varphi\ (ext\_tuple\_set\ AD\psi\ ns\varphi'\ \{fo\_nmlz\ AD\psi\ (map\ \sigma'\ ns)\}\ -\ Y))$ 
      using  $eval\_conj[OF\ tmp\ x\_proj\_singleton\ Some[folded\ x\_ns]]\ t\_in\_ext\_x$ 
      by ( $auto\ simp: x\_ns$ )
  }
}

```

**then show**  $t \in (\bigcup_{x \in \text{ad\_agr\_close\_set } AD \Delta \varphi} X \varphi. \text{ case Mapping.lookup idx } \psi (fo\_nmlz AD \psi (proj\_tuple ns (zip ns \varphi x))) \text{ of}$   
 $\text{None} \Rightarrow \text{ext\_tuple\_set } AD ns \varphi ns \varphi' \{x\}$   
 $| \text{Some } Y \Rightarrow \text{eval\_conj\_set } AD ns \varphi \{x\} ns \psi (ad\_agr\_close\_set AD \Delta \psi (ext\_tuple\_set AD \psi ns ns \varphi' \{fo\_nmlz AD \psi (proj\_tuple ns (zip ns \varphi x))\} - Y)))$   
**using**  $t\_in\_ext\_x$   
**by** ( $\text{intro UN\_I}[OF x\_in\_ad\_agr\_close]$ ) ( $\text{auto simp: } x\_ns \text{ split: option.splits}$ )  
**qed**  
**ultimately have**  $X\_def: X = fo\_nmlz AD 'proj\_fmla (Conj \varphi (Neg \psi)) \{\sigma. \text{esat } \varphi I \sigma UNIV\} - fo\_nmlz AD 'proj\_fmla (Conj \varphi (Neg \psi)) \{\sigma. \text{esat } \psi I \sigma UNIV\}$   
**by**  $\text{simp}$

**have**  $AD\_Neg\_sub: \text{act\_edom } (Neg \psi) I \subseteq AD$   
**by** ( $\text{auto simp: } AD\_def(1)$ )  
**have**  $X = fo\_nmlz AD 'proj\_fmla (Conj \varphi (Neg \psi)) \{\sigma. \text{esat } \varphi I \sigma UNIV\} \cap fo\_nmlz AD 'proj\_fmla (Conj \varphi (Neg \psi)) \{\sigma. \text{esat } (Neg \psi) I \sigma UNIV\}$   
**unfolding**  $X\_def$   
**by** ( $\text{auto simp: } proj\_fmla\_map \text{ dest!: } fo\_nmlz\_eqD$ )  
 $(metis AD\_def(4) ad\_agr\_list\_subset \text{esat\_UNIV\_ad\_agr\_list } fv\_fo\_fmla\_list\_set fv\_sub \text{sup\_ge1 } ts\_def(4))$   
**then have**  $\text{eval: eval\_ajoin } (fv\_fo\_fmla\_list \varphi) t \varphi (fv\_fo\_fmla\_list \psi) t \psi = \text{eval\_abs } (Conj \varphi (Neg \psi)) I$   
**using**  $proj\_fmla\_conj\_sub[OF AD\_Neg\_sub, of \varphi]$   
**unfolding**  $AD\_X\_def AD\_def(1)[\text{symmetric}] n\_def \text{eval\_abs\_def}$   
**by** ( $\text{auto simp: } proj\_fmla\_map$ )  
**have**  $wf\_conj\_neg: wf\_fo\_intp (Conj \varphi (Neg \psi)) I$   
**using**  $wf$   
**by** ( $\text{auto simp: } ts\_def$ )  
**show**  $?thesis$   
**using**  $fo\_wf\_eval\_abs[OF wf\_conj\_neg]$   
**by** ( $\text{auto simp: } eval$ )  
**qed**

**lemma eval\_disj:**  
**fixes**  $\varphi :: ('a :: \text{infinite}, 'b) fo\_fmla$   
**assumes**  $wf: fo\_wf \varphi I t \varphi fo\_wf \psi I t \psi$   
**shows**  $fo\_wf (Disj \varphi \psi) I$   
 $(\text{eval\_disj } (fv\_fo\_fmla\_list \varphi) t \varphi (fv\_fo\_fmla\_list \psi) t \psi)$

**proof** –

**obtain**  $AD \varphi n \varphi X \varphi AD \psi n \psi X \psi$  **where**  $ts\_def:$   
 $t \varphi = (AD \varphi, n \varphi, X \varphi) t \psi = (AD \psi, n \psi, X \psi)$   
 $AD \varphi = \text{act\_edom } \varphi I AD \psi = \text{act\_edom } \psi I$   
**using**  $\text{assms}$   
**by** ( $\text{cases } t \varphi, \text{cases } t \psi$ )  $\text{auto}$   
**have**  $AD\_sub: \text{act\_edom } \varphi I \subseteq AD \varphi \text{act\_edom } \psi I \subseteq AD \psi$   
**by** ( $\text{auto simp: } ts\_def(3,4)$ )

**obtain**  $AD n X$  **where**  $AD\_X\_def:$   
 $\text{eval\_disj } (fv\_fo\_fmla\_list \varphi) t \varphi (fv\_fo\_fmla\_list \psi) t \psi = (AD, n, X)$   
**by** ( $\text{cases } \text{eval\_disj } (fv\_fo\_fmla\_list \varphi) t \varphi (fv\_fo\_fmla\_list \psi) t \psi$ )  $\text{auto}$   
**have**  $AD\_def: AD = \text{act\_edom } (Disj \varphi \psi) I \text{act\_edom } (Disj \varphi \psi) I \subseteq AD$   
 $AD \varphi \subseteq AD AD \psi \subseteq AD AD = AD \varphi \cup AD \psi$   
**using**  $AD\_X\_def$   
**by** ( $\text{auto simp: } ts\_def \text{Let\_def}$ )  
**have**  $n\_def: n = nfv (Disj \varphi \psi)$   
**using**  $AD\_X\_def$   
**by** ( $\text{auto simp: } ts\_def \text{Let\_def } nfv\_card fv\_fo\_fmla\_list\_set$ )

```

define  $S\varphi$  where  $S\varphi \equiv \{\sigma. \text{esat } \varphi \text{ I } \sigma \text{ UNIV}\}$ 
define  $S\psi$  where  $S\psi \equiv \{\sigma. \text{esat } \psi \text{ I } \sigma \text{ UNIV}\}$ 
define  $ns\varphi'$  where  $ns\varphi' = \text{filter } (\lambda n. n \notin \text{fv\_fo\_fmla } \varphi) (\text{fv\_fo\_fmla\_list } \varphi)$ 
define  $ns\psi'$  where  $ns\psi' = \text{filter } (\lambda n. n \notin \text{fv\_fo\_fmla } \psi) (\text{fv\_fo\_fmla\_list } \varphi)$ 

note  $X\varphi\_def = \text{fo\_wf\_X}[OF \text{wf}(1)[\text{unfolded } ts\_def(1)], \text{unfolded } \text{proj\_fmla\_def}, \text{folded } S\varphi\_def]$ 
note  $X\psi\_def = \text{fo\_wf\_X}[OF \text{wf}(2)[\text{unfolded } ts\_def(2)], \text{unfolded } \text{proj\_fmla\_def}, \text{folded } S\psi\_def]$ 
have  $\text{fv\_sub}: \text{fv\_fo\_fmla } (\text{Disj } \varphi \psi) = \text{fv\_fo\_fmla } \varphi \cup \text{set } (\text{fv\_fo\_fmla\_list } \psi)$ 
 $\text{fv\_fo\_fmla } (\text{Disj } \varphi \psi) = \text{fv\_fo\_fmla } \psi \cup \text{set } (\text{fv\_fo\_fmla\_list } \varphi)$ 
by (auto simp: fv\_fo\_fmla\_list\_set)
note  $\text{res\_left\_alt} = \text{ext\_tuple\_ad\_agr\_close}[OF S\varphi\_def AD\_sub(1) AD\_def(3)]$ 
 $X\varphi\_def(1)[\text{folded } S\varphi\_def] \text{ ns}\varphi'\_def \text{ sorted\_distinct\_fv\_list } \text{fv\_sub}(1)$ 
note  $\text{res\_right\_alt} = \text{ext\_tuple\_ad\_agr\_close}[OF S\psi\_def AD\_sub(2) AD\_def(4)]$ 
 $X\psi\_def(1)[\text{folded } S\psi\_def] \text{ ns}\psi'\_def \text{ sorted\_distinct\_fv\_list } \text{fv\_sub}(2)$ 

have  $X = \text{fo\_nmlz } AD \text{ 'proj\_fmla } (\text{Disj } \varphi \psi) \{\sigma. \text{esat } \varphi \text{ I } \sigma \text{ UNIV}\} \cup$ 
 $\text{fo\_nmlz } AD \text{ 'proj\_fmla } (\text{Disj } \varphi \psi) \{\sigma. \text{esat } \psi \text{ I } \sigma \text{ UNIV}\}$ 
using  $AD\_X\_def$ 
apply (simp add: ts\_def(1,2) Let\_def AD\_def(5)[symmetric])
unfolding  $\text{fv\_fo\_fmla\_list\_set } \text{proj\_fmla\_def } \text{ns}\varphi'\_def[\text{symmetric}] \text{ns}\psi'\_def[\text{symmetric}]$ 
 $S\varphi\_def[\text{symmetric}] S\psi\_def[\text{symmetric}]$ 
using  $\text{res\_left\_alt}(1) \text{res\_right\_alt}(1)$ 
by auto
then have  $\text{eval}: \text{eval\_disj } (\text{fv\_fo\_fmla\_list } \varphi) \text{ t}\varphi (\text{fv\_fo\_fmla\_list } \psi) \text{ t}\psi =$ 
 $\text{eval\_abs } (\text{Disj } \varphi \psi) \text{ I}$ 
unfolding  $AD\_X\_def AD\_def(1)[\text{symmetric}] \text{ n\_def } \text{eval\_abs\_def}$ 
by (auto simp: proj\_fmla\_map)
have  $\text{wf\_disj}: \text{wf\_fo\_intp } (\text{Disj } \varphi \psi) \text{ I}$ 
using  $\text{wf}$ 
by (auto simp: ts\_def)
show ?thesis
using  $\text{fo\_wf\_eval\_abs}[OF \text{wf\_disj}]$ 
by (auto simp: eval)
qed

lemma  $\text{fv\_ex\_all}$ :
assumes  $\text{pos } i (\text{fv\_fo\_fmla\_list } \varphi) = \text{None}$ 
shows  $\text{fv\_fo\_fmla\_list } (\text{Exists } i \varphi) = \text{fv\_fo\_fmla\_list } \varphi$ 
 $\text{fv\_fo\_fmla\_list } (\text{Forall } i \varphi) = \text{fv\_fo\_fmla\_list } \varphi$ 
using  $\text{pos\_complete}[of i \text{fv\_fo\_fmla\_list } \varphi] \text{fv\_fo\_fmla\_list\_eq}[of \text{Exists } i \varphi \varphi]$ 
 $\text{fv\_fo\_fmla\_list\_eq}[of \text{Forall } i \varphi \varphi] \text{assms}$ 
by (auto simp: fv\_fo\_fmla\_list\_set)

lemma  $\text{nfv\_ex\_all}$ :
assumes  $\text{Some: pos } i (\text{fv\_fo\_fmla\_list } \varphi) = \text{Some } j$ 
shows  $\text{nfv } \varphi = \text{Suc } (\text{nfv } (\text{Exists } i \varphi)) \text{ nfv } \varphi = \text{Suc } (\text{nfv } (\text{Forall } i \varphi))$ 
proof –
have  $i \in \text{fv\_fo\_fmla } \varphi \text{ j} < \text{nfv } \varphi \text{ i} \in \text{set } (\text{fv\_fo\_fmla\_list } \varphi)$ 
using  $\text{fv\_fo\_fmla\_list\_set } \text{pos\_set}[of i \text{fv\_fo\_fmla\_list } \varphi]$ 
 $\text{pos\_length}[of i \text{fv\_fo\_fmla\_list } \varphi] \text{Some}$ 
by (fastforce simp: nfv\_def)
then show  $\text{nfv } \varphi = \text{Suc } (\text{nfv } (\text{Exists } i \varphi)) \text{ nfv } \varphi = \text{Suc } (\text{nfv } (\text{Forall } i \varphi))$ 
using  $\text{nfv\_card}[of \varphi] \text{nfv\_card}[of \text{Exists } i \varphi] \text{nfv\_card}[of \text{Forall } i \varphi]$ 
by (auto simp: finite\_fv\_fo\_fmla)
qed

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

```

(metis (mono\_tags, lifting) distinct\_filter distinct\_remdups\_adj\_sort  
distinct\_remdups\_id filter\_set filter\_sort remdups\_adj\_set sorted\_list\_of\_set\_sort\_remdups  
sorted\_remdups\_adj sorted\_sort sorted\_sort\_id)

**lemma** *eval\_exists*:

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

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

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

**proof** –

**obtain**  $AD \ n \ X$  **where**  $t\_def: t = (AD, n, X)$

$AD = \text{act\_edom } \varphi \text{ I } AD = \text{act\_edom } (\text{Exists } i \varphi) \text{ I}$

**using** *assms*

**by** (*cases t*) *auto*

**note**  $X\_def = \text{fo\_wf\_X}[OF \ \text{wf}[\text{unfolded } t\_def], \text{folded } t\_def(2)]$

**have**  $\text{eval}: \text{eval\_exists } i (\text{fv\_fo\_fmla\_list } \varphi) t = \text{eval\_abs } (\text{Exists } i \varphi) \text{ I}$

**proof** (*cases pos i (fv\\_fo\\_fmla\\_list \varphi)*)

**case** *None*

**note**  $\text{fv\_eq} = \text{fv\_ex\_all}[OF \ \text{None}]$

**have**  $X = \text{fo\_nmlz } AD \ ' \ \text{proj\_fmla } (\text{Exists } i \varphi) \ \{\sigma. \text{esat } \varphi \text{ I } \sigma \text{ UNIV}\}$

**unfolding**  $X\_def$

**by** (*auto simp: proj\_fmla\_def fv\_eq*)

**also have**  $\dots = \text{fo\_nmlz } AD \ ' \ \text{proj\_fmla } (\text{Exists } i \varphi) \ \{\sigma. \text{esat } (\text{Exists } i \varphi) \text{ I } \sigma \text{ UNIV}\}$

**using** *esat\_exists\_not\_fv[of i \varphi UNIV I] pos\_complete[OF None]*

**by** (*simp add: fv\\_fo\\_fmla\\_list\\_set*)

**finally show** *?thesis*

**by** (*auto simp: t\\_def None eval\_abs\_def fv\_eq nfv\_def*)

**next**

**case** (*Some j*)

**have**  $\text{fo\_nmlz } AD \ ' \ \text{rem\_nth } j \ ' \ X =$

$\text{fo\_nmlz } AD \ ' \ \text{proj\_fmla } (\text{Exists } i \varphi) \ \{\sigma. \text{esat } (\text{Exists } i \varphi) \text{ I } \sigma \text{ UNIV}\}$

**proof** (*rule set\_eqI, rule iffI*)

**fix**  $vs$

**assume**  $vs \in \text{fo\_nmlz } AD \ ' \ \text{rem\_nth } j \ ' \ X$

**then obtain**  $ws$  **where**  $ws\_def: ws \in \text{fo\_nmlz } AD \ ' \ \text{proj\_fmla } \varphi \ \{\sigma. \text{esat } \varphi \text{ I } \sigma \text{ UNIV}\}$

$vs = \text{fo\_nmlz } AD \ (\text{rem\_nth } j \ ws)$

**unfolding**  $X\_def$

**by** *auto*

**then obtain**  $\sigma$  **where**  $\sigma\_def: \text{esat } \varphi \text{ I } \sigma \text{ UNIV}$

$ws = \text{fo\_nmlz } AD \ (\text{map } \sigma \ (\text{fv\_fo\_fmla\_list } \varphi))$

**by** (*auto simp: proj\_fmla\_map*)

**obtain**  $\tau$  **where**  $\tau\_def: ws = \text{map } \tau \ (\text{fv\_fo\_fmla\_list } \varphi)$

**using** *fo\_nmlz\_map \sigma\\_def(2)*

**by** *blast*

**have**  $\text{esat\_}\tau: \text{esat } (\text{Exists } i \varphi) \text{ I } \tau \text{ UNIV}$

**using** *esat\_UNIV\_ad\_agr\_list[OF fo\_nmlz\_ad\_agr[of AD map \sigma (fv\\_fo\\_fmla\\_list \varphi),  
folded \sigma\\_def(2), unfolded \tau\\_def]] \sigma\\_def(1)*

**by** (*auto simp: t\\_def intro!: exI[of \_ \tau i]*)

**have**  $\text{rem\_nth\_ws}: \text{rem\_nth } j \ ws = \text{map } \tau \ (\text{fv\_fo\_fmla\_list } (\text{Exists } i \varphi))$

**using** *rem\_nth\_sound[of fv\\_fo\\_fmla\\_list \varphi i j \tau] sorted\_distinct\_fv\_list Some*

**unfolding** *fv\\_fo\\_fmla\\_list\_exists \tau\\_def*

**by** *auto*

**have**  $vs \in \text{fo\_nmlz } AD \ ' \ \text{proj\_fmla } (\text{Exists } i \varphi) \ \{\sigma. \text{esat } (\text{Exists } i \varphi) \text{ I } \sigma \text{ UNIV}\}$

**using** *ws\\_def(2) esat\\_}\tau*

**unfolding** *rem\_nth\_ws*

**by** (*auto simp: proj\_fmla\_map*)

**then show**  $vs \in \text{fo\_nmlz } AD \ ' \ \text{proj\_fmla } (\text{Exists } i \varphi) \ \{\sigma. \text{esat } (\text{Exists } i \varphi) \text{ I } \sigma \text{ UNIV}\}$

**by** *auto*

**next**

```

fix vs
assume assm: vs ∈ fo_nmlz AD ‘ proj_fmula (Exists i φ) {σ. esat (Exists i φ) I σ UNIV}
from assm obtain σ where σ_def: vs = fo_nmlz AD (map σ (fv_fo_fmula_list (Exists i φ)))
  esat (Exists i φ) I σ UNIV
  by (auto simp: proj_fmula_map)
then obtain x where x_def: esat φ I (σ(i := x)) UNIV
  by auto
define ws where ws ≡ fo_nmlz AD (map (σ(i := x)) (fv_fo_fmula_list φ))
then have length ws = nfv φ
  using nfv_def fo_nmlz_length by (metis length_map)
then have ws_in: ws ∈ fo_nmlz AD ‘ proj_fmula φ {σ. esat φ I σ UNIV}
  using x_def ws_def
  by (auto simp: fo_nmlz_sound proj_fmula_map)
obtain τ where τ_def: ws = map τ (fv_fo_fmula_list φ)
  using fo_nmlz_map ws_def
  by blast
have rem_nth_ws: rem_nth j ws = map τ (fv_fo_fmula_list (Exists i φ))
  using rem_nth_sound[of fv_fo_fmula_list φ i j] sorted_distinct_fv_list Some
  unfolding fv_fo_fmula_list_exists τ_def
  by auto
have set (fv_fo_fmula_list (Exists i φ)) ⊆ set (fv_fo_fmula_list φ)
  by (auto simp: fv_fo_fmula_list_exists)
then have ad_agr: ad_agr_list AD (map (σ(i := x)) (fv_fo_fmula_list (Exists i φ)))
  (map τ (fv_fo_fmula_list (Exists i φ)))
  by (rule ad_agr_list_subset)
  (rule fo_nmlz_ad_agr[of AD map (σ(i := x)) (fv_fo_fmula_list φ), folded ws_def,
    unfolded τ_def])
have map_fv_cong: map (σ(i := x)) (fv_fo_fmula_list (Exists i φ)) =
  map σ (fv_fo_fmula_list (Exists i φ))
  by (auto simp: fv_fo_fmula_list_exists)
have vs_rem_nth: vs = fo_nmlz AD (rem_nth j ws)
  unfolding σ_def(1) rem_nth_ws
  apply (rule fo_nmlz_eqI)
  using ad_agr[unfolded map_fv_cong] .
show vs ∈ fo_nmlz AD ‘ rem_nth j ‘ X
  using Some ws_in
  unfolding vs_rem_nth X_def
  by auto
qed
then show ?thesis
  using nfv_ex_all[OF Some]
  by (auto simp: t_def Some eval_abs_def nfv_def)
qed
have wf_ex: wf_fo_intp (Exists i φ) I
  using wf
  by (auto simp: t_def)
show ?thesis
  using fo_wf_eval_abs[OF wf_ex]
  by (auto simp: eval)
qed

lemma fv_fo_fmula_list_forall: fv_fo_fmula_list (Forall n φ) = filter ((≠) n) (fv_fo_fmula_list φ)
  by (auto simp: fv_fo_fmula_list_def)
  (metis (mono_tags, lifting) distinct_filter distinct_remdups_adj_sort
    distinct_remdups_id filter_set filter_sort remdups_adj_set sorted_list_of_set_sort_remdups
    sorted_remdups_adj sorted_sort sorted_sort_id)

lemma pairwise_take_drop:

```

**assumes** *pairwise P (set (zip xs ys)) length xs = length ys*  
**shows** *pairwise P (set (zip (take i xs @ drop (Suc i) xs) (take i ys @ drop (Suc i) ys)))*  
**by** (*rule pairwise\_subset[OF assms(1)] (auto simp: set\_zip assms(2))*)

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

*fo\_nmlz AD xs = xs  $\implies$  set xs = set xs  $\cap$  Inl ' AD  $\cup$  Inr ' {..*card (Inr -' set xs)}}*  
**by** (*metis fo\_nmlz\_sound fo\_nmlzd\_set card\_Inr\_vimage\_le\_length min.absorb2*)*

**lemma** *adAgrListTakeDrop: adAgrList AD xs ys  $\implies$*

*adAgrList AD (take i xs @ drop (Suc i) xs) (take i ys @ drop (Suc i) ys)*  
**apply** (*auto simp: adAgrListDef adEquivListDef spEquivListDef*)  
**apply** (*metis take\_zip\_in\_set\_takeD*)  
**apply** (*metis drop\_zip\_in\_set\_dropD*)  
**using** *pairwise\_take\_drop*  
**by** *fastforce*

**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** -

**have** *adAgr: adAgrList AD (add\_nth i z zs) (fo\_nmlz AD (add\_nth i z zs))*  
**using** *fo\_nmlz\_adAgr*  
**by** *auto*

**have** *i\_lt\_add: i < length (add\_nth i z zs) i < length (fo\_nmlz AD (add\_nth i z zs))*  
**using** *add\_nth\_length assms(2)*  
**by** (*fastforce simp: fo\_nmlz\_length*)**+**

**show** *?thesis*

**using** *adAgrListTakeDrop[OF adAgr, of i, folded rem\_nth\_take\_drop[OF i\_lt\_add(1)]*  
*rem\_nth\_take\_drop[OF i\_lt\_add(2)], unfolded rem\_nth\_add\_nth[OF assms(2)]]*  
**apply** (*subst eq\_commute*)  
**apply** (*subst assms(1)[symmetric]*)  
**apply** (*auto intro: fo\_nmlz\_eqI*)  
**done**

**qed**

**lemma** *adAgrListAdd:*

**assumes** *adAgrList AD xs ys i  $\leq$  length xs*  
**shows**  $\exists z' \in \text{Inl ' AD} \cup \text{Inr ' \{..*Suc (card (Inr -' set ys))\} \cup set ys.*$   
*adAgrList AD (take i xs @ z # drop i xs) (take i ys @ z' # drop i ys)*

**proof** -

**define** *n where n = length xs*

**have** *len\_ys: n = length ys*

**using** *assms(1)*  
**by** (*auto simp: adAgrListDef n\_def*)

**obtain**  $\sigma$  **where**  $\sigma\_def: xs = \text{map } \sigma [0..<n]$

**unfolding** *n\_def*  
**by** (*metis map\_nth*)

**obtain**  $\tau$  **where**  $\tau\_def: ys = \text{map } \tau [0..<n]$

**unfolding** *len\_ys*  
**by** (*metis map\_nth*)

**have** *i\_le\_n: i  $\leq$  n*

**using** *assms(2)*  
**by** (*auto simp: n\_def*)

**have** *set\_n: set [0..<n] = \{..n\} - \{n\} set ([0..<i] @ n # [i..<n]) = \{..n\}*  
**using** *i\_le\_n*

**by** *auto*

**have** *adAgr: adAgrSets (\{..n\} - \{n\}) (\{..n\} - \{n\}) AD  $\sigma$   $\tau$*

**using** *iffD2[OF adAgrListLink, OF assms(1)[unfolded  $\sigma\_def$   $\tau\_def$ ]]*



```

unfolding set_n .
have set_ys:  $\tau \text{ ' } \{\dots n\} - \{n\} = \text{set } ys$ 
by (auto simp:  $\tau\_def$ )
obtain z' where z'_def:  $z' \in \text{Inl ' } AD \cup \text{Inr ' } \{\dots < \text{Suc } (\text{card } (\text{Inr - ' } \text{set } ys))\} \cup \text{set } ys$ 
  ad_agr_sets  $\{\dots n\} \{\dots n\} AD (\sigma(n := z)) (\tau(n := z'))$ 
using extend_ $\tau$ [OF ad_agr_subset_refl,
  of Inl '  $AD \cup \text{Inr ' } \{\dots < \text{Suc } (\text{card } (\text{Inr - ' } \text{set } ys))\} \cup \text{set } ys$ ]
by (auto simp: set_ys)
have map_take:  $\text{map } (\sigma(n := z)) ([0..<i] @ n \# [i..<n]) = \text{take } i \text{ xs } @ z \# \text{drop } i \text{ xs}$ 
   $\text{map } (\tau(n := z')) ([0..<i] @ n \# [i..<n]) = \text{take } i \text{ ys } @ z' \# \text{drop } i \text{ ys}$ 
using i_le_n
by (auto simp:  $\sigma\_def \tau\_def \text{take\_map drop\_map}$ )
show ?thesis
using iffD1[OF ad_agr_list_link, OF z'_def(2)[unfolded set_n[symmetric]]] z'_def(1)
unfolding map_take
by auto
qed

```

```

lemma add_nth_restrict:
assumes fo_nmlz AD zs = zs  $i \leq \text{length } zs$ 
shows  $\exists z' \in \text{Inl ' } AD \cup \text{Inr ' } \{\dots < \text{Suc } (\text{card } (\text{Inr - ' } \text{set } zs))\}.$ 
  fo_nmlz AD (add_nth i z zs) = fo_nmlz AD (add_nth i z' zs)
proof -
have set zs  $\subseteq \text{Inl ' } AD \cup \text{Inr ' } \{\dots < \text{Suc } (\text{card } (\text{Inr - ' } \text{set } zs))\}$ 
using fo_nmlz_set_card[OF assms(1)]
by auto
then obtain z' where z'_def:
   $z' \in \text{Inl ' } AD \cup \text{Inr ' } \{\dots < \text{Suc } (\text{card } (\text{Inr - ' } \text{set } zs))\}$ 
  ad_agr_list AD (take i zs @ z \# drop i zs) (take i zs @ z' \# drop i zs)
using ad_agr_list_add[OF ad_agr_list_refl assms(2), of AD z]
by auto blast
then show ?thesis
unfolding add_nth_take_drop[OF assms(2)]
by (auto intro: fo_nmlz_eqI)
qed

```

```

lemma fo_nmlz_add_rem:
assumes  $i \leq \text{length } zs$ 
shows  $\exists z'. \text{fo\_nmlz } AD (\text{add\_nth } i z zs) = \text{fo\_nmlz } AD (\text{add\_nth } i z' (\text{fo\_nmlz } AD zs))$ 
proof -
have ad_agr: ad_agr_list AD zs (fo_nmlz AD zs)
using fo_nmlz_ad_agr
by auto
have i_le_fo_nmlz:  $i \leq \text{length } (\text{fo\_nmlz } AD zs)$ 
using assms(1)
by (auto simp: fo_nmlz_length)
obtain x where x_def: ad_agr_list AD (add_nth i z zs) (add_nth i x (fo_nmlz AD zs))
using ad_agr_list_add[OF ad_agr assms(1)]
by (auto simp: add_nth_take_drop[OF assms(1)] add_nth_take_drop[OF i_le_fo_nmlz])
then show ?thesis
using fo_nmlz_eqI
by auto
qed

```

```

lemma fo_nmlz_add_rem':
assumes  $i \leq \text{length } zs$ 
shows  $\exists z'. \text{fo\_nmlz } AD (\text{add\_nth } i z (\text{fo\_nmlz } AD zs)) = \text{fo\_nmlz } AD (\text{add\_nth } i z' zs)$ 
proof -

```

```

have adAgr: adAgr_list AD (fo_nmlz AD zs) zs
  using adAgr_list_comm[OF fo_nmlz_adAgr]
  by auto
have i_le_fo_nmlz:  $i \leq \text{length } (\text{fo\_nmlz } AD \text{ } zs)$ 
  using assms(1)
  by (auto simp: fo_nmlz_length)
obtain x where x_def: adAgr_list AD (add_nth i z (fo_nmlz AD zs)) (add_nth i x zs)
  using adAgr_list_add[OF adAgr i_le_fo_nmlz]
  by (auto simp: add_nth_take_drop[OF assms(1)] add_nth_take_drop[OF i_le_fo_nmlz])
then show ?thesis
  using fo_nmlz_eqI
  by auto
qed

lemma fo_nmlz_add_nth_rem_nth:
  assumes fo_nmlz AD xs = xs  $i < \text{length } xs$ 
  shows  $\exists z. \text{fo\_nmlz } AD (\text{add\_nth } i \ z (\text{fo\_nmlz } AD (\text{rem\_nth } i \ xs))) = xs$ 
  using rem_nth_length[OF assms(2)] fo_nmlz_add_rem[of i rem_nth i xs AD xs ! i,
    unfolded assms(1) add_nth_rem_nth_self[OF assms(2)]] assms(2)
  by (subst eq_commute) auto

lemma sp_equiv_list_almost_same: sp_equiv_list (xs @ v # ys) (xs @ w # ys)  $\implies$ 
   $v \in \text{set } xs \cup \text{set } ys \vee w \in \text{set } xs \cup \text{set } ys \implies v = w$ 
  by (auto simp: sp_equiv_list_def pairwise_def) (metis UnCI sp_equiv_pair.simps zip_same)+

lemma adAgr_list_add_nth:
  assumes  $i \leq \text{length } zs$  adAgr_list AD (add_nth i v zs) (add_nth i w zs)  $v \neq w$ 
  shows  $\{v, w\} \cap (\text{Inl } 'AD \cup \text{set } zs) = \{\}$ 
  using assms(2)[unfolded add_nth_take_drop[OF assms(1)]] assms(1,3) sp_equiv_list_almost_same
  by (auto simp: adAgr_list_def ad_equiv_list_def ad_equiv_pair.simps)
  (smt append_take_drop_id set_append sp_equiv_list_almost_same)+

lemma Inr_in_tuple:
  assumes fo_nmlz AD zs = zs  $n < \text{card } (\text{Inr } -' \text{set } zs)$ 
  shows Inr n  $\in \text{set } zs$ 
  using assms fo_nmlz_set_card[OF assms(1)]
  by (auto simp: fo_nmlzd_code[symmetric])

lemma card_wit_sub:
  assumes finite Z  $\text{card } Z \leq \text{card } \{ts \in X. \exists z \in Z. ts = fz\}$ 
  shows  $f' \ 'Z \subseteq X$ 
proof -
  have set_unfold:  $\{ts \in X. \exists z \in Z. ts = fz\} = f' \ 'Z \cap X$ 
  by auto
  show ?thesis
  using assms
  unfolding set_unfold
  by (metis Int_lower1 card_image_le card_seteq finite_imageI inf.absorb_iff1 le_antisym
    surj_card_le)
qed

lemma add_nth_iff_card:
  assumes  $(\bigwedge xs. xs \in X \implies \text{fo\_nmlz } AD \ xs = xs)$   $(\bigwedge xs. xs \in X \implies i < \text{length } xs)$ 
  fo_nmlz AD zs = zs  $i \leq \text{length } zs$  finite AD finite X
  shows  $(\forall z. \text{fo\_nmlz } AD (\text{add\_nth } i \ z \ zs) \in X) \longleftrightarrow$ 
   $\text{Suc } (\text{card } AD + \text{card } (\text{Inr } -' \text{set } zs)) \leq \text{card } \{ts \in X. \exists z. ts = \text{fo\_nmlz } AD (\text{add\_nth } i \ z \ zs)\}$ 
proof -
  have inj: inj_on  $(\lambda z. \text{fo\_nmlz } AD (\text{add\_nth } i \ z \ zs))$ 

```

```

(Inl ' AD  $\cup$  Inr ' {.. $\text{Suc}$  (card (Inr -' set zs))})
using adAgrList_add_nth[OF assms(4)] Inr_in_tuple[OF assms(3)] less_Suc_eq
by (fastforce simp: inj_on_def dest!: fo_nmlz_eqD)
have card_Un: card (Inl ' AD  $\cup$  Inr ' {.. $\text{Suc}$  (card (Inr -' set zs))}) =
  Suc (card AD + card (Inr -' set zs))
using card_Un_disjoint[of Inl ' AD Inr ' {.. $\text{Suc}$  (card (Inr -' set zs))}] assms(5)
by (auto simp add: card_image disjoint_iff_not_equal)
have restrict_z: ( $\forall z. \text{fo\_nmlz AD (add\_nth } i \ z \ zs) \in X$ )  $\longleftrightarrow$ 
  ( $\forall z \in \text{Inl ' AD} \cup \text{Inr ' } \{.. $\text{Suc}$  (card (Inr -' set zs))\}. \text{fo\_nmlz AD (add\_nth } i \ z \ zs) \in X$ )
using add_nth_restrict[OF assms(3,4)]
by metis
have restrict_z': {ts  $\in$  X.  $\exists z. ts = \text{fo\_nmlz AD (add\_nth } i \ z \ zs)$ } =
  {ts  $\in$  X.  $\exists z \in \text{Inl ' AD} \cup \text{Inr ' } \{.. $\text{Suc}$  (card (Inr -' set zs))\}.
  ts = \text{fo\_nmlz AD (add\_nth } i \ z \ zs)\}
using add_nth_restrict[OF assms(3,4)]
by auto
{
assume  $\bigwedge z. \text{fo\_nmlz AD (add\_nth } i \ z \ zs) \in X$ 
then have image_sub: ( $\lambda z. \text{fo\_nmlz AD (add\_nth } i \ z \ zs)$ ) '
  (Inl ' AD  $\cup$  Inr ' {.. $\text{Suc}$  (card (Inr -' set zs))})  $\subseteq$ 
  {ts  $\in$  X.  $\exists z. ts = \text{fo\_nmlz AD (add\_nth } i \ z \ zs)$ }
by auto
have Suc (card AD + card (Inr -' set zs))  $\leq$ 
  card {ts  $\in$  X.  $\exists z. ts = \text{fo\_nmlz AD (add\_nth } i \ z \ zs)$ }
unfolding card_Un[symmetric]
using card_inj_on_le[OF inj image_sub] assms(6)
by auto
then have Suc (card AD + card (Inr -' set zs))  $\leq$ 
  card {ts  $\in$  X.  $\exists z. ts = \text{fo\_nmlz AD (add\_nth } i \ z \ zs)$ }
by (auto simp: card_image)
}
moreover
{
assume assm: card (Inl ' AD  $\cup$  Inr ' {.. $\text{Suc}$  (card (Inr -' set zs))})  $\leq$ 
  card {ts  $\in$  X.  $\exists z \in \text{Inl ' AD} \cup \text{Inr ' } \{.. $\text{Suc}$  (card (Inr -' set zs))\}.
  ts = \text{fo\_nmlz AD (add\_nth } i \ z \ zs)\}
have  $\forall z \in \text{Inl ' AD} \cup \text{Inr ' } \{.. $\text{Suc}$  (card (Inr -' set zs))\}. \text{fo\_nmlz AD (add\_nth } i \ z \ zs) \in X$ 
using card_wit_sub[OF _ assm] assms(5)
by auto
}
ultimately show ?thesis
unfolding restrict_z[symmetric] restrict_z'[symmetric] card_Un
by auto
qed$$ 
```

```

lemma set_fo_nmlz_add_nth_rem_nth:
assumes  $j < \text{length } xs \wedge x. x \in X \implies \text{fo\_nmlz AD } x = x$ 
   $\wedge x. x \in X \implies j < \text{length } x$ 
shows {ts  $\in$  X.  $\exists z. ts = \text{fo\_nmlz AD (add\_nth } j \ z \ (\text{fo\_nmlz AD (rem\_nth } j \ xs)))$ } =
  {y  $\in$  X.  $\text{fo\_nmlz AD (rem\_nth } j \ y) = \text{fo\_nmlz AD (rem\_nth } j \ xs)$ }
using fo_nmlz_rem_nth_add_nth[where ?zs= $\text{fo\_nmlz AD (rem\_nth } j \ xs)$ ] rem_nth_length[OF assms(1)]
  fo_nmlz_add_nth_rem_nth assms
by (fastforce simp: fo_nmlz_idem[OF fo_nmlz_sound] fo_nmlz_length)

```

```

lemma eval_forall:
fixes  $\varphi :: ('a :: \text{infinite}, 'b) \text{fo\_fmla}$ 
assumes wf: fo_wf  $\varphi$  I t
shows fo_wf (Forall i  $\varphi$ ) I (eval_forall i (fv_fo_fmla_list  $\varphi$ ) t)

```

**proof** –

```

obtain  $AD\ n\ X$  where  $t\_def: t = (AD, n, X)$   $AD = act\_edom\ \varphi\ I$ 
 $AD = act\_edom\ (Forall\ i\ \varphi)\ I$ 
using  $assms$ 
by  $(cases\ t)\ auto$ 
have  $AD\_sub: act\_edom\ \varphi\ I \subseteq AD$ 
by  $(auto\ simp: t\_def(2))$ 
have  $fin\_AD: finite\ AD$ 
using  $finite\_act\_edom\ wf$ 
by  $(auto\ simp: t\_def)$ 
have  $fin\_X: finite\ X$ 
using  $wf$ 
by  $(auto\ simp: t\_def)$ 
note  $X\_def = fo\_wf\_X[OF\ wf[unfolded\ t\_def],\ folded\ t\_def(2)]$ 
have  $eval: eval\_forall\ i\ (fv\_fo\_fmla\_list\ \varphi)\ t = eval\_abs\ (Forall\ i\ \varphi)\ I$ 
proof  $(cases\ pos\ i\ (fv\_fo\_fmla\_list\ \varphi))$ 
case  $None$ 
note  $fv\_eq = fv\_ex\_all[OF\ None]$ 
have  $X = fo\_nmlz\ AD\ 'proj\_fmla\ (Forall\ i\ \varphi)\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$ 
unfolding  $X\_def$ 
by  $(auto\ simp: proj\_fmla\_def\ fv\_eq)$ 
also have  $\dots = fo\_nmlz\ AD\ 'proj\_fmla\ (Forall\ i\ \varphi)\ \{\sigma.\ esat\ (Forall\ i\ \varphi)\ I\ \sigma\ UNIV\}$ 
using  $esat\_forall\_not\_fv[of\ i\ \varphi\ UNIV\ I]\ pos\_complete[OF\ None]$ 
by  $(auto\ simp: fv\_fo\_fmla\_list\_set)$ 
finally show  $?thesis$ 
by  $(auto\ simp: t\_def\ None\ eval\_abs\_def\ fv\_eq\ nfv\_def)$ 
next
case  $(Some\ j)$ 
have  $i\_in\_fv: i \in fv\_fo\_fmla\ \varphi$ 
by  $(rule\ pos\_set[OF\ Some,\ unfolded\ fv\_fo\_fmla\_list\_set])$ 
have  $fo\_nmlz\_X: \bigwedge xs. xs \in X \implies fo\_nmlz\ AD\ xs = xs$ 
by  $(auto\ simp: X\_def\ proj\_fmla\_map\ fo\_nmlz\_idem[OF\ fo\_nmlz\_sound])$ 
have  $j\_lt\_len: \bigwedge xs. xs \in X \implies j < length\ xs$ 
using  $pos\_sound[OF\ Some]$ 
by  $(auto\ simp: X\_def\ proj\_fmla\_map\ fo\_nmlz\_length)$ 
have  $rem\_nth\_j\_le\_len: \bigwedge xs. xs \in X \implies j \leq length\ (fo\_nmlz\ AD\ (rem\_nth\ j\ xs))$ 
using  $rem\_nth\_length\ j\_lt\_len$ 
by  $(fastforce\ simp: fo\_nmlz\_length)$ 
have  $img\_proj\_fmla: Mapping.keys\ (Mapping.filter\ (\lambda t\ Z.\ Suc\ (card\ AD + card\ (Inr\ -' set\ t)) \leq$ 
 $card\ Z)$ 
 $(cluster\ (Some\ \circ\ (\lambda ts.\ fo\_nmlz\ AD\ (rem\_nth\ j\ ts)))\ X)) =$ 
 $fo\_nmlz\ AD\ 'proj\_fmla\ (Forall\ i\ \varphi)\ \{\sigma.\ esat\ (Forall\ i\ \varphi)\ I\ \sigma\ UNIV\}$ 
proof  $(rule\ set\_eqI,\ rule\ iffI)$ 
fix  $vs$ 
assume  $vs \in Mapping.keys\ (Mapping.filter\ (\lambda t\ Z.\ Suc\ (card\ AD + card\ (Inr\ -' set\ t)) \leq card\ Z)$ 
 $(cluster\ (Some\ \circ\ (\lambda ts.\ fo\_nmlz\ AD\ (rem\_nth\ j\ ts)))\ X))$ 
then obtain  $ws$  where  $ws\_def: ws \in X\ vs = fo\_nmlz\ AD\ (rem\_nth\ j\ ws)$ 
 $\bigwedge a.\ fo\_nmlz\ AD\ (add\_nth\ j\ a\ (fo\_nmlz\ AD\ (rem\_nth\ j\ ws))) \in X$ 
using  $add\_nth\_iff\_card[OF\ fo\_nmlz\_X\ j\_lt\_len\ fo\_nmlz\_idem[OF\ fo\_nmlz\_sound]]$ 
 $rem\_nth\_j\_le\_len\ fin\_AD\ fin\_X]$   $set\_fo\_nmlz\_add\_nth\_rem\_nth[OF\ j\_lt\_len\ fo\_nmlz\_X$ 
 $j\_lt\_len]$ 
by  $transfer\ (fastforce\ split: option.splits\ if\_splits)$ 
then obtain  $\sigma$  where  $\sigma\_def:$ 
 $esat\ \varphi\ I\ \sigma\ UNIV\ ws = fo\_nmlz\ AD\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \varphi))$ 
unfolding  $X\_def$ 
by  $(auto\ simp: proj\_fmla\_map)$ 
obtain  $\tau$  where  $\tau\_def: ws = map\ \tau\ (fv\_fo\_fmla\_list\ \varphi)$ 
using  $fo\_nmlz\_map\ \sigma\_def(2)$ 

```

```

    by blast
  have fo_nmlzd_τ: fo_nmlzd AD (map τ (fv_fo_fmula_list φ))
    unfolding τ_def[symmetric] σ_def(2)
    by (rule fo_nmlz_sound)
  have rem_nth_j_ws: rem_nth j ws = map τ (filter ((≠) i) (fv_fo_fmula_list φ))
    using rem_nth_sound[OF _ Some] sorted_distinct_fv_list
    by (auto simp: τ_def)
  have esat_τ: esat (Forall i φ) I τ UNIV
    unfolding esat.simps
  proof (rule ballI)
    fix x
    have fo_nmlz AD (add_nth j x (rem_nth j ws)) ∈ X
      using fo_nmlz_add_rem[of j rem_nth j ws AD x] rem_nth_length
      j_lt_len[OF ws_def(1)] ws_def(3)
      by fastforce
    then have fo_nmlz AD (map (τ(i := x)) (fv_fo_fmula_list φ)) ∈ X
      using add_nth_rem_nth_map[OF _ Some, of x] sorted_distinct_fv_list
      unfolding τ_def
      by fastforce
    then show esat φ I (τ(i := x)) UNIV
      by (auto simp: X_def proj_fmula_map esat_UNIV_ad_agr_list[OF _ AD_sub]
        dest!: fo_nmlz_eqD)
  qed
  have rem_nth_ws: rem_nth j ws = map τ (fv_fo_fmula_list (Forall i φ))
    using rem_nth_sound[OF _ Some] sorted_distinct_fv_list
    by (auto simp: fv_fo_fmula_list_forall τ_def)
  then show vs ∈ fo_nmlz AD ‘ proj_fmula (Forall i φ) {σ. esat (Forall i φ) I σ UNIV}
    using ws_def(2) esat_τ
    by (auto simp: proj_fmula_map rem_nth_ws)
next
  fix vs
  assume assm: vs ∈ fo_nmlz AD ‘ proj_fmula (Forall i φ) {σ. esat (Forall i φ) I σ UNIV}
  from assm obtain σ where σ_def: vs = fo_nmlz AD (map σ (fv_fo_fmula_list (Forall i φ)))
    esat (Forall i φ) I σ UNIV
    by (auto simp: proj_fmula_map)
  then have all_esat:  $\bigwedge x. esat φ I (σ(i := x)) UNIV$ 
    by auto
  define ws where ws ≡ fo_nmlz AD (map σ (fv_fo_fmula_list φ))
  then have length_ws = nfv φ
    using nfv_def fo_nmlz_length by (metis length_map)
  then have ws_in: ws ∈ fo_nmlz AD ‘ proj_fmula φ {σ. esat φ I σ UNIV}
    using all_esat[of σ i] ws_def
    by (auto simp: fo_nmlz_sound proj_fmula_map)
  then have ws_in_X: ws ∈ X
    by (auto simp: X_def)
  obtain τ where τ_def: ws = map τ (fv_fo_fmula_list φ)
    using fo_nmlz_map ws_def
    by blast
  have rem_nth_ws: rem_nth j ws = map τ (fv_fo_fmula_list (Forall i φ))
    using rem_nth_sound[of fv_fo_fmula_list φ i j] sorted_distinct_fv_list Some
    unfolding fv_fo_fmula_list_forall τ_def
    by auto
  have set (fv_fo_fmula_list (Forall i φ)) ⊆ set (fv_fo_fmula_list φ)
    by (auto simp: fv_fo_fmula_list_forall)
  then have ad_agr: ad_agr_list AD (map σ (fv_fo_fmula_list (Forall i φ)))
    (map τ (fv_fo_fmula_list (Forall i φ)))
    apply (rule ad_agr_list_subset)
    using fo_nmlz_ad_agr[of AD] ws_def τ_def

```

```

    by metis
  have map_fv_cong:  $\bigwedge x. \text{map } (\sigma(i := x)) \text{ (fv\_fo\_fmla\_list (Forall } i \ \varphi)) =$ 
    map  $\sigma$  (fv\_fo\_fmla\_list (Forall  $i \ \varphi$ ))
    by (auto simp: fv\_fo\_fmla\_list_forall)
  have vs_rem_nth:  $vs = \text{fo\_nmlz } AD \text{ (rem\_nth } j \ ws)$ 
    unfolding  $\sigma\_def(1)$  rem_nth_ws
    apply (rule fo_nmlz_eqI)
    using ad_agr[unfolded map_fv_cong] .
  have  $\bigwedge a. \text{fo\_nmlz } AD \text{ (add\_nth } j \ a \ (\text{fo\_nmlz } AD \text{ (rem\_nth } j \ ws))) \in$ 
    fo_nmlz  $AD \text{ 'proj\_fmla } \varphi \ \{\sigma. \text{esat } \varphi \ I \ \sigma \ UNIV\}$ 
  proof -
    fix a
    obtain  $x$  where add_rem:  $\text{fo\_nmlz } AD \text{ (add\_nth } j \ a \ (\text{fo\_nmlz } AD \text{ (rem\_nth } j \ ws))) =$ 
      fo_nmlz  $AD \text{ (map } (\tau(i := x)) \text{ (fv\_fo\_fmla\_list } \varphi))$ 
      using add_nth_rem_nth_map[OF Some, of  $\tau$ ] sorted_distinct_fv_list
        fo_nmlz_add_rem[of  $j$  rem_nth  $j$  ws] rem_nth_length[of  $j$  ws]
        j_lt_len[OF ws_in_X]
      by (fastforce simp:  $\tau\_def$ )
    have esat (Forall  $i \ \varphi$ )  $I \ \tau \ UNIV$ 
      apply (rule iffD1[OF esat_UNIV_ad_agr_list  $\sigma\_def(2)$ , OF subset_refl, folded t_def])
      using fo_nmlz_ad_agr[of  $AD$  map  $\sigma$  (fv\_fo\_fmla\_list  $\varphi$ ), folded ws_def, unfolded  $\tau\_def$ ]
      unfolding ad_agr_list_link[symmetric]
      by (auto simp: fv\_fo\_fmla\_list_set ad_agr_sets_def sp_equiv_def pairwise_def)
    then have esat  $\varphi \ I \ (\tau(i := x)) \ UNIV$ 
      by auto
    then show  $\text{fo\_nmlz } AD \text{ (add\_nth } j \ a \ (\text{fo\_nmlz } AD \text{ (rem\_nth } j \ ws))) \in$ 
      fo_nmlz  $AD \text{ 'proj\_fmla } \varphi \ \{\sigma. \text{esat } \varphi \ I \ \sigma \ UNIV\}$ 
      by (auto simp: add_rem proj_fmla_map)
  qed
  then show  $vs \in \text{Mapping.keys } (\text{Mapping.filter } (\lambda t \ Z. \text{Suc } (\text{card } AD + \text{card } (\text{Inr } - \text{'set } t)) \leq \text{card}$ 
 $Z)$ 
    (cluster (Some  $\circ (\lambda ts. \text{fo\_nmlz } AD \text{ (rem\_nth } j \ ts))$ )  $X$ )
    unfolding vs_rem_nth_X_def[symmetric]
    using add_nth_iff_card[OF fo_nmlz_X j_lt_len fo_nmlz_idem[OF fo_nmlz_sound]
      rem_nth_j_le_len fin_AD fin_X] set_fo_nmlz_add_nth_rem_nth[OF j_lt_len fo_nmlz_X
 $j\_lt\_len$ ] ws_in_X
    by transfer (fastforce split: option.splits if_splits)
  qed
  show ?thesis
    using nfv_ex_all[OF Some]
    by (simp add: t_def Some eval_abs_def nfv_def img_proj_fmla[unfolded t_def(2)]
      split: option.splits)
  qed
  have wf_all:  $wf\_fo\_intp$  (Forall  $i \ \varphi$ )  $I$ 
    using wf
    by (auto simp: t_def)
  show ?thesis
    using fo_wf_eval_abs[OF wf_all]
    by (auto simp: eval)
  qed

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 'X) else Infin)

lemma fo_res_fin:
  fixes  $t :: ('a :: infinite, nat) fo_t$ 
  assumes fo_wf  $\varphi \ I \ t$  finite (fo_rep  $t$ )
  shows fo_res  $t = \text{Fin } (\text{fo\_rep } t)$ 

```

```

proof –
  obtain  $AD\ n\ X$  where  $t\_def: t = (AD, n, X)$ 
    using  $assms(I)$ 
    by  $(cases\ t)\ auto$ 
  show  $?thesis$ 
    using  $fo\_fin\ assms$ 
    by  $(fastforce\ simp\ only: t\_def\ fo\_res.simps\ fo\_rep\_fin\ split: if\_splits)$ 
qed

```

```

lemma  $fo\_res\_infin$ :
  fixes  $t :: ('a :: infinite, nat)\ fo\_t$ 
  assumes  $fo\_wf\ \varphi\ I\ t\ \neg finite\ (fo\_rep\ t)$ 
  shows  $fo\_res\ t = Infin$ 
proof –
  obtain  $AD\ n\ X$  where  $t\_def: t = (AD, n, X)$ 
    using  $assms(I)$ 
    by  $(cases\ t)\ auto$ 
  show  $?thesis$ 
    using  $fo\_fin\ assms$ 
    by  $(fastforce\ simp\ only: t\_def\ fo\_res.simps\ split: if\_splits)$ 
qed

```

```

lemma  $fo\_rep$ :  $fo\_wf\ \varphi\ I\ t \implies fo\_rep\ t = proj\_sat\ \varphi\ I$ 
  by  $(cases\ t)\ auto$ 

```

```

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  $eval = Ailamazyan.eval$ 
  apply  $standard$ 
    apply  $(rule\ fo\_rep, assumption+)$ 
    apply  $(rule\ fo\_res\_fin, assumption+)$ 
    apply  $(rule\ fo\_res\_infin, assumption+)$ 
    apply  $(rule\ eval\_pred, assumption+)$ 
    apply  $(rule\ eval\_bool)$ 
    apply  $(rule\ eval\_eq)$ 
    apply  $(rule\ eval\_neg, assumption+)$ 
    apply  $(rule\ eval\_conj, assumption+)$ 
    apply  $(rule\ eval\_ajoin, assumption+)$ 
    apply  $(rule\ eval\_disj, assumption+)$ 
    apply  $(rule\ eval\_exists, assumption+)$ 
    apply  $(rule\ eval\_forall, assumption+)$ 
  done

```

```

definition  $esat\_UNIV :: ('a :: infinite, 'b)\ fo\_fmla \implies ('a\ table, 'b)\ fo\_intp \implies ('a + nat)\ val \implies bool$ 
where
   $esat\_UNIV\ \varphi\ I\ \sigma = esat\ \varphi\ I\ \sigma\ UNIV$ 

```

```

lemma  $esat\_UNIV\_code[code]: esat\_UNIV\ \varphi\ I\ \sigma \iff (if\ wf\_fo\_intp\ \varphi\ I\ then$ 
   $(case\ eval\_fmla\ \varphi\ I\ of\ (AD, n, X) \implies$ 
     $fo\_nmlz\ (act\_edom\ \varphi\ I)\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \varphi)) \in X)$ 
   $else\ esat\_UNIV\ \varphi\ I\ \sigma)$ 

```

```

proof –
  obtain  $AD\ n\ T$  where  $t\_def: Ailamazyan.eval\_fmla\ \varphi\ I = (AD, n, T)$ 
    by  $(cases\ Ailamazyan.eval\_fmla\ \varphi\ I)\ auto$ 
  {
    assume  $wf\_fo\_intp: wf\_fo\_intp\ \varphi\ I$ 

```

```

note fo_wf = Ailamazyan.eval_fmla_correct[OF wf_fo_intp, unfolded t_def]
note T_def = fo_wf_X[OF fo_wf]
have AD_def: AD = act_edom  $\varphi$  I
  using fo_wf
  by auto
have esat_UNIV  $\varphi$  I  $\sigma \longleftrightarrow$ 
  fo_nmlz (act_edom  $\varphi$  I) (map  $\sigma$  (fv_fo_fmla_list  $\varphi$ ))  $\in$  T
  using esat_UNIV_ad_agr_list[OF _ subset_refl]
  by (force simp add: esat_UNIV_def T_def AD_def proj_fmla_map
    dest!: fo_nmlz_eqD)
}
then show ?thesis
  by (auto simp: t_def)
qed

```

```

lemma sat_code[code]:
  fixes  $\varphi :: ('a :: infinite, 'b) \text{fo\_fmla}$ 
  shows sat  $\varphi$  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 (Inl  $\circ$   $\sigma$ ) (fv_fo_fmla_list  $\varphi$ ))  $\in$  X)
    else sat  $\varphi$  I  $\sigma$ )
  using esat_UNIV_code sat_esat_conv[folded esat_UNIV_def]
  by metis

```

**end**

**theory** Ailamazyan\_Code

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

```

definition insert_db :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ('a, 'b set) mapping  $\Rightarrow$  ('a, 'b set) mapping where
  insert_db k v m = (case Mapping.lookup m k of None  $\Rightarrow$ 
    Mapping.update k ({v}) m
  | Some vs  $\Rightarrow$  Mapping.update k ({v}  $\cup$  vs) m)

```

```

fun convert_db_rec :: ('a  $\times$  'c list) list  $\Rightarrow$  (('a  $\times$  nat), 'c list set) mapping  $\Rightarrow$ 
  (('a  $\times$  nat), 'c list set) mapping where
  convert_db_rec [] m = m
  | convert_db_rec ((r, ts) # ktss) m = convert_db_rec ktss (insert_db (r, length ts) ts m)

```

```

lemma convert_db_rec_mono: Mapping.lookup m (r, n) = Some tss  $\Longrightarrow$ 
   $\exists$  tss'. Mapping.lookup (convert_db_rec ktss m) (r, n) = Some tss'  $\wedge$  tss  $\subseteq$  tss'
apply (induction ktss m arbitrary: tss rule: convert_db_rec.induct)
apply (auto simp: insert_db_def fun_upd_def Mapping.lookup_update' split: option.splits if_splits)
apply (metis option.discI)
apply (smt option.inject order_trans subset_insertI)
done

```

```

lemma convert_db_rec_sound: (r, ts)  $\in$  set ktss  $\Longrightarrow$ 
   $\exists$  tss. Mapping.lookup (convert_db_rec ktss m) (r, length ts) = Some tss  $\wedge$  ts  $\in$  tss
proof (induction ktss m rule: convert_db_rec.induct)

```

**case** (2 r ts ktss m)

**obtain** tss **where**

Mapping.lookup (convert\_db\_rec ktss (insert\_db (r, length ts) ts m)) (r, length ts) = Some tss  
 ts  $\in$  tss

**using** convert\_db\_rec\_mono[of insert\_db (r, length ts) ts m r length ts \_ ktss]

**by** atomize\_elim (auto simp: insert\_db\_def Mapping.lookup\_update' split: option.splits)+



**then show** *?case*  
**using** 2  
**by** *auto*  
**qed** *auto*

**lemma** *convert\_db\_rec\_complete*:  $\text{Mapping.lookup } (\text{convert\_db\_rec } ktss \ m) \ (r, n) = \text{Some } tss' \implies$   
 $ts \in tss' \implies$   
 $(\text{length } ts = n \wedge (r, ts) \in \text{set } ktss) \vee (\exists tss. \text{Mapping.lookup } m \ (r, n) = \text{Some } tss \wedge ts \in tss)$   
**by** (*induction ktss m rule: convert\_db\_rec.induct*)  
*(auto simp: insert\_db\_def Mapping.lookup\_update' split: option.splits if\_splits)*

**definition** *convert\_db* ::  $('a \times 'c \text{ list}) \text{ list} \Rightarrow ('c \text{ table}, 'a) \text{ fo\_intp}$  **where**  
 $\text{convert\_db } ktss = (\text{let } m = \text{convert\_db\_rec } ktss \ \text{Mapping.empty in}$   
 $(\lambda x. \text{case Mapping.lookup } m \ x \text{ of None} \Rightarrow \{\} \mid \text{Some } v \Rightarrow v))$

**lemma** *convert\_db\_correct*:  $(ts \in \text{convert\_db } ktss \ (r, n) \longrightarrow n = \text{length } ts) \wedge$   
 $((r, ts) \in \text{set } ktss \longleftrightarrow ts \in \text{convert\_db } ktss \ (r, \text{length } ts))$   
**by** (*auto simp: convert\_db\_def dest!: convert\_db\_rec\_sound[of \_ \_ \_ Mapping.empty]*)  
*split: option.splits*  
*(metis Mapping.lookup\_empty convert\_db\_rec\_complete option.distinct(1))+*

**lemma** *Inl\_vimage\_set\_code[code\_unfold]*:  $\text{Inl } -' \text{ set } as = \text{set } (\text{List.map\_filter } (\text{case\_sum } \text{Some } \text{Map.empty})$   
 $as)$   
**by** (*induction as*) (*auto simp: List.map\_filter\_simps split: option.splits sum.splits*)

**lemma** *Inr\_vimage\_set\_code[code\_unfold]*:  $\text{Inr } -' \text{ set } as = \text{set } (\text{List.map\_filter } (\text{case\_sum } \text{Map.empty}$   
 $\text{Some}) \ as)$   
**by** (*induction as*) (*auto simp: List.map\_filter\_simps split: option.splits sum.splits*)

**lemma** *Inl\_vimage\_code*:  $\text{Inl } -' \text{ as} = \text{projl } \{x \in as. \text{isl } x\}$   
**by** (*force simp: vimage\_def*)

**lemmas** *ad\_pred\_code[code]* = *ad\_terms.simps[unfolded Inl\_vimage\_code]*

**lemmas** *fo\_wf\_code[code]* = *fo\_wf.simps[unfolded Inl\_vimage\_code]*

**definition** *empty\_J* ::  $((\text{nat}, \text{nat}) \text{ fo\_t}, \text{String.literal}) \text{ fo\_intp}$  **where**  
 $\text{empty\_J} = (\lambda(\_, n). \text{eval\_pred } (\text{map } \text{Var } [0..<n]) \ \{\})$

**definition** *eval\_fin\_nat* ::  $(\text{nat}, \text{String.literal}) \text{ fo\_fmla} \Rightarrow (\text{nat table}, \text{String.literal}) \text{ fo\_intp} \Rightarrow \text{nat}$   
 $\text{eval\_res}$  **where**  
 $\text{eval\_fin\_nat } \varphi \ I = \text{eval } \varphi \ I$

**definition** *sat\_fin\_nat* ::  $(\text{nat}, \text{String.literal}) \text{ fo\_fmla} \Rightarrow (\text{nat table}, \text{String.literal}) \text{ fo\_intp} \Rightarrow \text{nat val} \Rightarrow$   
 $\text{bool}$  **where**  
 $\text{sat\_fin\_nat } \varphi \ I = \text{sat } \varphi \ I$

**definition** *convert\_nat\_db* ::  $(\text{String.literal} \times \text{nat list}) \text{ list} \Rightarrow$   
 $(\text{nat table}, \text{String.literal}) \text{ fo\_intp}$  **where**  
 $\text{convert\_nat\_db} = \text{convert\_db}$

**definition** *rbt\_nat\_fold* ::  $\_ \Rightarrow \text{nat set\_rbt} \Rightarrow \_ \Rightarrow \_$  **where**  
 $\text{rbt\_nat\_fold} = \text{RBT\_Set2.fold}$

**definition** *rbt\_nat\_list\_fold* ::  $\_ \Rightarrow (\text{nat list}) \text{ set\_rbt} \Rightarrow \_ \Rightarrow \_$  **where**

`rbt_nat_list_fold = RBT_Set2.fold`

**definition** `rbt_sum_list_fold` :: `_ => ((nat + nat) list) set_rbt => _ => _` **where**  
`rbt_sum_list_fold = RBT_Set2.fold`

**export\_code** `eval_fin_nat sat_fin_nat fv_fo_fmula_list convert_nat_db rbt_nat_fold rbt_nat_list_fold`  
`rbt_sum_list_fold Const Conj Inl Fin nat_of_integer integer_of_nat RBT_set`  
**in** `OCaml module_name Eval_FO file_prefix verified`

**definition** `φ` :: `(nat, String.literal) fo_fmula` **where**  
`φ ≡ Exists 0 (Conj (FO.Eqa (Var 0) (Const 2)) (FO.Eqa (Var 0) (Var 1)))`

**value** `eval_fin_nat φ (convert_nat_db [])`

**value** `sat_fin_nat φ (convert_nat_db []) (λ_. 0)`  
**value** `sat_fin_nat φ (convert_nat_db []) (λ_. 2)`

**definition** `ψ` :: `(nat, String.literal) fo_fmula` **where**  
`ψ ≡ Forall 2 (Disj (FO.Eqa (Var 2) (Const 42))`  
`(Exists 1 (Conj (FO.Pred (String.implode "P") [Var 0, Var 1])`  
`(Neg (FO.Pred (String.implode "Q") [Var 1, Var 2])))`

**value** `eval_fin_nat ψ (convert_nat_db`  
`[(String.implode "P", [1, 20]),`  
`(String.implode "P", [9, 20]),`  
`(String.implode "P", [2, 30]),`  
`(String.implode "P", [3, 31]),`  
`(String.implode "P", [4, 32]),`  
`(String.implode "P", [5, 30]),`  
`(String.implode "P", [6, 30]),`  
`(String.implode "P", [7, 30]),`  
`(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.