First-Order Query Evaluation

Martin Raszyk

December 19, 2021

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 occurring in the query and the database. 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 intermediate relation for a negated subquery in a conjunction. Finally, we export code for the infinite domain of natural numbers.

Contents

theory *Infinite*
  imports *Main*
begin

class *infinite* =
  assumes *infinite_UNIV*: *infinite* $(UNIV :: 'a\ set)$
begin

lemma *arb_element*: $finite\ Y \Longrightarrow \exists x :: 'a.\ x \notin Y$
  using *ex_new_if_finite infinite_UNIV*
  by *blast*

lemma *arb_finite_subset*: $finite\ Y \Longrightarrow \exists X :: 'a\ set.\ Y \cap X = \{\} \wedge finite\ X \wedge n \leq card\ X$
proof −
  assume *fin*: *finite Y*
  then obtain *X* **where** $X \subseteq UNIV - Y$ *finite X* $n \leq card\ X$
    using *infinite_UNIV*
    by (*metis Compl_eq_Diff_UNIV finite_compl infinite_arbitrarily_large order_refl*)
  then show *?thesis*
    by *auto*
qed

lemma *arb_countable_map*: $finite\ Y \Longrightarrow \exists f :: (nat \Rightarrow 'a).\ inj\ f \wedge range\ f \subseteq UNIV - Y$
  using *infinite_UNIV*

by (*auto simp*: *infinite_countable_subset*)

end

instance *nat* :: *infinite*
by *standard auto*

end
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** $\cdot$ *60*) **where**
eval_term σ (*Const c*) = *c*
| *eval_term* σ (*Var n*) = σ *n*

definition *eval_terms* :: *'a val* $\Rightarrow$ (*'a fo_term*) *list* $\Rightarrow$ *'a list* (**infix** $\odot$ *60*) **where**
eval_terms σ *ts* = *map* (*eval_term* σ) *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* ∈ *set ts* ⟹ *m* ∈ *fv_fo_terms_set ts*
  by (*induction ts*) (*auto simp*: *fv_fo_terms_set_def*)

lemma *fv_fo_terms_setD*: *m* ∈ *fv_fo_terms_set ts* ⟹ *Var m* ∈ *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* ⟹ *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* ∈ *fv_fo_term_set t* ⟹ σ *n* = σ' *n*) ⟹
  eval_term σ *t* = *eval_term* σ' *t*
  by (*cases t*) *auto*

lemma *eval_terms_fv_fo_terms_set*: $\sigma \odot$ *ts* = $\sigma' \odot$ *ts* ⟹ *n* ∈ *fv_fo_terms_set ts* ⟹ σ *n* = σ' *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* ∈ *fv_fo_terms_set ts* ⟹ σ *n* = σ' *n*) ⟹
  eval_terms σ *ts* = *eval_terms* σ' *ts*
  by (*auto simp*: *eval_terms_def fv_fo_terms_set_def intro*: *eval_term_cong*)

datatype ('*a*, '*b*) *fo_fmla* =
  Pred '*b* ('*a fo_term*) *list*
| *Bool bool*
| *Eqa* '*a fo_term* '*a fo_term*
| *Neg* ('*a*, '*b*) *fo_fmla*
| *Conj* ('*a*, '*b*) *fo_fmla* ('*a*, '*b*) *fo_fmla*

```
| Disj ('a, 'b) fo_fmla ('a, 'b) fo_fmla
| Exists nat ('a, 'b) fo_fmla
| Forall nat ('a, 'b) fo_fmla

fun fv_fo_fmla_list_rec :: ('a, 'b) fo_fmla ⇒ nat list where
  fv_fo_fmla_list_rec (Pred _ ts) = fv_fo_terms_list ts
| fv_fo_fmla_list_rec (Bool b) = []
| fv_fo_fmla_list_rec (Eqa t t') = fv_fo_term_list t @ fv_fo_term_list t'
| fv_fo_fmla_list_rec (Neg φ) = fv_fo_fmla_list_rec φ
| fv_fo_fmla_list_rec (Conj φ ψ) = fv_fo_fmla_list_rec φ @ fv_fo_fmla_list_rec ψ
| fv_fo_fmla_list_rec (Disj φ ψ) = fv_fo_fmla_list_rec φ @ fv_fo_fmla_list_rec ψ
| fv_fo_fmla_list_rec (Exists n φ) = filter (λm. n ≠ m) (fv_fo_fmla_list_rec φ)
| fv_fo_fmla_list_rec (Forall n φ) = filter (λm. n ≠ m) (fv_fo_fmla_list_rec φ)

definition fv_fo_fmla_list :: ('a, 'b) fo_fmla ⇒ nat list where
  fv_fo_fmla_list φ = remdups_adj (sort (fv_fo_fmla_list_rec φ))

fun fv_fo_fmla :: ('a, 'b) fo_fmla ⇒ nat set where
  fv_fo_fmla (Pred _ ts) = fv_fo_terms_set ts
| fv_fo_fmla (Bool b) = {}
| fv_fo_fmla (Eqa t t') = fv_fo_term_set t ∪ fv_fo_term_set t'
| fv_fo_fmla (Neg φ) = fv_fo_fmla φ
| fv_fo_fmla (Conj φ ψ) = fv_fo_fmla φ ∪ fv_fo_fmla ψ
| fv_fo_fmla (Disj φ ψ) = fv_fo_fmla φ ∪ fv_fo_fmla ψ
| fv_fo_fmla (Exists n φ) = fv_fo_fmla φ − {n}
| fv_fo_fmla (Forall n φ) = fv_fo_fmla φ − {n}

lemma finite_fv_fo_fmla: finite (fv_fo_fmla φ)
  by (induction φ 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 φ) = fv_fo_fmla φ
  unfolding fv_fo_fmla_list_def
  by (induction φ 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 φ)
  by (auto simp: fv_fo_fmla_list_def intro: distinct_remdups_adj_sort)

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

lemma fv_fo_fmla_list_eq: fv_fo_fmla φ = fv_fo_fmla ψ ⟹ fv_fo_fmla_list φ = fv_fo_fmla_list
ψ
  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 φ ψ) = fv_fo_fmla_list (Conj ψ φ)
  using fv_fo_fmla_list_eq[of Conj φ ψ Conj ψ φ]
  by auto

type_synonym 'a table = ('a list) set

type_synonym ('t, 'b) fo_intp = 'b × nat ⇒ 't

fun wf_fo_intp :: ('a, 'b) fo_fmla ⇒ ('a table, 'b) fo_intp ⇒ bool where
  wf_fo_intp (Pred r ts) I ⟷ finite (I (r, length ts))
```

| $\mathit{wf\_fo\_intp}\ (\mathit{Bool}\ b)\ I \longleftrightarrow \mathit{True}$
| $\mathit{wf\_fo\_intp}\ (\mathit{Eqa}\ t\ t')\ I \longleftrightarrow \mathit{True}$
| $\mathit{wf\_fo\_intp}\ (\mathit{Neg}\ \varphi)\ I \longleftrightarrow \mathit{wf\_fo\_intp}\ \varphi\ I$
| $\mathit{wf\_fo\_intp}\ (\mathit{Conj}\ \varphi\ \psi)\ I \longleftrightarrow \mathit{wf\_fo\_intp}\ \varphi\ I \wedge \mathit{wf\_fo\_intp}\ \psi\ I$
| $\mathit{wf\_fo\_intp}\ (\mathit{Disj}\ \varphi\ \psi)\ I \longleftrightarrow \mathit{wf\_fo\_intp}\ \varphi\ I \wedge \mathit{wf\_fo\_intp}\ \psi\ I$
| $\mathit{wf\_fo\_intp}\ (\mathit{Exists}\ n\ \varphi)\ I \longleftrightarrow \mathit{wf\_fo\_intp}\ \varphi\ I$
| $\mathit{wf\_fo\_intp}\ (\mathit{Forall}\ n\ \varphi)\ I \longleftrightarrow \mathit{wf\_fo\_intp}\ \varphi\ I$

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

lemma $\mathit{sat\_fv\_cong}$: $(\bigwedge n.\ n \in \mathit{fv\_fo\_fmla}\ \varphi \Longrightarrow \sigma\ n = \sigma'\ n) \Longrightarrow$
  $\mathit{sat}\ \varphi\ I\ \sigma \longleftrightarrow \mathit{sat}\ \varphi\ I\ \sigma'$
proof $(\mathit{induction}\ \varphi\ \mathit{arbitrary}{:}\ \sigma\ \sigma')$
  case $(\mathit{Neg}\ \varphi)$
  show *?case*
    using $\mathit{Neg}(1)[\mathit{of}\ \sigma\ \sigma']\ \mathit{Neg}(2)$
    by *auto*
next
  case $(\mathit{Conj}\ \varphi\ \psi)$
  show *?case*
    using $\mathit{Conj}(1,2)[\mathit{of}\ \sigma\ \sigma']\ \mathit{Conj}(3)$
    by *auto*
next
  case $(\mathit{Disj}\ \varphi\ \psi)$
  show *?case*
    using $\mathit{Disj}(1,2)[\mathit{of}\ \sigma\ \sigma']\ \mathit{Disj}(3)$
    by *auto*
next
  case $(\mathit{Exists}\ n\ \varphi)$
  have $\bigwedge x.\ \mathit{sat}\ \varphi\ I\ (\sigma(n := x)) = \mathit{sat}\ \varphi\ I\ (\sigma'(n := x))$
    using $\mathit{Exists}(2)$
    by $(\mathit{auto}\ \mathit{intro}!{:}\ \mathit{Exists}(1))$
  then show *?case*
    by *simp*
next
  case $(\mathit{Forall}\ n\ \varphi)$
  have $\bigwedge x.\ \mathit{sat}\ \varphi\ I\ (\sigma(n := x)) = \mathit{sat}\ \varphi\ I\ (\sigma'(n := x))$
    using $\mathit{Forall}(2)$
    by $(\mathit{auto}\ \mathit{intro}!{:}\ \mathit{Forall}(1))$
  then show *?case*
    by *simp*
qed $(\mathit{auto}\ \mathit{cong}{:}\ \mathit{eval\_terms\_cong}\ \mathit{eval\_term\_cong})$

definition $\mathit{proj\_sat} :: ('a,\ 'b)\ \mathit{fo\_fmla} \Rightarrow ('a\ \mathit{table},\ 'b)\ \mathit{fo\_intp} \Rightarrow 'a\ \mathit{table}$ **where**
  $\mathit{proj\_sat}\ \varphi\ I = (\lambda\sigma.\ \mathit{map}\ \sigma\ (\mathit{fv\_fo\_fmla\_list}\ \varphi))\ `\ \{\sigma.\ \mathit{sat}\ \varphi\ I\ \sigma\}$

end
theory *Eval_FO*
  imports *Infinite FO*
begin

```
datatype 'a eval_res = Fin 'a table | Infin | Wf_error

locale eval_fo =
  fixes wf :: ('a :: infinite, 'b) fo_fmla ⇒ ('b × nat ⇒ 'a list set) ⇒ 't ⇒ bool
    and abs :: ('a fo_term) list ⇒ 'a table ⇒ 't
    and rep :: 't ⇒ 'a table
    and res :: 't ⇒ 'a eval_res
    and eval_bool :: bool ⇒ 't
    and eval_eq :: 'a fo_term ⇒ 'a fo_term ⇒ 't
    and eval_neg :: nat list ⇒ 't ⇒ 't
    and eval_conj :: nat list ⇒ 't ⇒ nat list ⇒ 't ⇒ 't
    and eval_ajoin :: nat list ⇒ 't ⇒ nat list ⇒ 't ⇒ 't
    and eval_disj :: nat list ⇒ 't ⇒ nat list ⇒ 't ⇒ 't
    and eval_exists :: nat ⇒ nat list ⇒ 't ⇒ 't
    and eval_forall :: nat ⇒ nat list ⇒ 't ⇒ 't
  assumes fo_rep: wf φ I t ⟹ rep t = proj_sat φ I
  and fo_res_fin: wf φ I t ⟹ finite (rep t) ⟹ res t = Fin (rep t)
  and fo_res_infin: wf φ I t ⟹ ¬finite (rep t) ⟹ res t = Infin
  and fo_abs: finite (I (r, length ts)) ⟹ wf (Pred r ts) I (abs ts (I (r, length ts)))
  and fo_bool: wf (Bool b) I (eval_bool b)
  and fo_eq: wf (Eqa trm trm') I (eval_eq trm trm')
  and fo_neg: wf φ I t ⟹ wf (Neg φ) I (eval_neg (fv_fo_fmla_list φ) t)
  and fo_conj: wf φ I tφ ⟹ wf ψ I tψ ⟹ (case ψ of Neg ψ' ⇒ False | _ ⇒ True) ⟹
    wf (Conj φ ψ) I (eval_conj (fv_fo_fmla_list φ) tφ (fv_fo_fmla_list ψ) tψ)
  and fo_ajoin: wf φ I tφ ⟹ wf ψ' I tψ' ⟹
    wf (Conj φ (Neg ψ')) I (eval_ajoin (fv_fo_fmla_list φ) tφ (fv_fo_fmla_list ψ') tψ')
  and fo_disj: wf φ I tφ ⟹ wf ψ I tψ ⟹
    wf (Disj φ ψ) I (eval_disj (fv_fo_fmla_list φ) tφ (fv_fo_fmla_list ψ) tψ)
  and fo_exists: wf φ I t ⟹ wf (Exists i φ) I (eval_exists i (fv_fo_fmla_list φ) t)
  and fo_forall: wf φ I t ⟹ wf (Forall i φ) I (eval_forall i (fv_fo_fmla_list φ) t)
begin

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

lemma eval_fmla_correct:
  fixes φ :: ('a :: infinite, 'b) fo_fmla
  assumes wf_fo_intp φ I
  shows wf φ I (eval_fmla φ I)
  using assms
proof (induction φ I rule: eval_fmla.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 φ I)
  then show ?case
    using fo_neg
    by auto
next
  case (5 φ ψ I)
  have fins: wf_fo_intp φ I wf_fo_intp ψ I
    using 5(10)
    by auto
  have evalφ: wf φ I (eval_fmla φ I)
    using 5(1)[OF _ _ fins(1)]
    by auto
  show ?case
  proof (cases ∃ψ'. ψ = Neg ψ')
    case True
    then obtain ψ' where ψ_def: ψ = Neg ψ'
      by auto
    have fin: wf_fo_intp ψ' I
      using fins(2)
      by (auto simp: ψ_def)
    have evalψ': wf ψ' I (eval_fmla ψ' I)
      using 5(5)[OF _ _ _ ψ_def fin]
      by auto
    show ?thesis
      unfolding ψ_def
      using fo_ajoin[OF evalφ evalψ']
      by auto
  next
    case False
    then have evalψ: wf ψ I (eval_fmla ψ I)
      using 5 fins(2)
      by (cases ψ) auto
    have eval: eval_fmla (Conj φ ψ) I = eval_conj (fv_fo_fmla_list φ) (eval_fmla φ I)
      (fv_fo_fmla_list ψ) (eval_fmla ψ I)
      using False
      by (auto simp: Let_def split: fo_fmla.splits)
    show wf (Conj φ ψ) I (eval_fmla (Conj φ ψ) I)
      using fo_conj[OF evalφ evalψ, folded eval] False
      by (auto split: fo_fmla.splits)
  qed
next
  case (6 φ ψ I)
  then show ?case
    using fo_disj
    by auto
next
  case (7 i φ I)
```

```
  then show ?case
    using fo_exists
    by auto
next
  case (8 i φ I)
  then show ?case
    using fo_forall
    by auto
qed

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

lemma eval_fmla_proj_sat:
  fixes φ :: ('a :: infinite, 'b) fo_fmla
  assumes wf_fo_intp φ I
  shows rep (eval_fmla φ I) = proj_sat φ I
  using eval_fmla_correct[OF assms]
  by (auto simp: fo_rep)

lemma eval_sound:
  fixes φ :: ('a :: infinite, 'b) fo_fmla
  assumes eval φ I = Fin Z
  shows Z = proj_sat φ I
proof −
  have wf φ I (eval_fmla φ I)
    using eval_fmla_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 φ :: ('a :: infinite, 'b) fo_fmla
  assumes eval φ I = Infin
  shows infinite (proj_sat φ I)
proof −
  have wf φ I (eval_fmla φ I)
    using eval_fmla_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 ⇒ 'b set is
  λm. ⋃(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* ⇒ *Code.abort* (*STR ''set_of_idx RBT_Mapping*: *ccompare* = *None''*) (λ_. *set_of_idx* (*RBT_Mapping t*))
| *Some* _ ⇒ ⋃(*snd ‘ 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* ⇒ *Code.abort* (*STR ''combine RBT_Mapping*: *ccompare* = *None''*) (λ_. *Mapping.combine f* (*RBT_Mapping t*) (*RBT_Mapping u*))
| *Some* _ ⇒ *RBT_Mapping* (*RBT_Mapping2.join* (λ_. *f*) *t u*))
by (*auto simp add*: *Mapping.combine.abs_eq Mapping_inject lookup_join split*: *option.split*)

lift_definition *mapping_join* :: (*'b* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping* **is**
λ*f m m' x*. *case m x of None* ⇒ *None* | *Some y* ⇒ (*case m' x of None* ⇒ *None* | *Some y'* ⇒ *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* ⇒ *Code.abort* (*STR ''mapping_join RBT_Mapping*: *ccompare* = *None''*) (λ_. *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*) =
(λ*k*. *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**
λ*m 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**
λ*f 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* ∘ *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* ≡ (λ*b* _ *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 ⟹ rbt_lookup (add_to_rbt (a, b) t) x = (if a = x then Some (case rbt_lookup t x of None ⇒ {b} | Some Y ⇒ 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′ ⟹ rbt_lookup (RBT_Impl.fold (add_option_to_rbt f) t t′) x =
    (if x ∈ Option.these (f ‘ set (RBT_Impl.keys t)) ∪ set (RBT_Impl.keys t′) then Some ({y ∈ set (RBT_Impl.keys t). f y = Some x}
    ∪ (case rbt_lookup t′ x of None ⇒ {} | Some Y ⇒ 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* $\Rightarrow$ $\{k\}$ | *Some Y* $\Rightarrow$ *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* $\Rightarrow$ $\{\}$ | *Some Y* $\Rightarrow$ *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* $\times$ *'b* $\Rightarrow$ (*'a*, *'b set*) *rbt* $\Rightarrow$ (*'a*, *'b set*) *rbt* **where**
add_to_rbt_comp = (λ(*a*, *b*) *t*. *case rbt_comp_lookup c t a of None* $\Rightarrow$ *rbt_comp_insert c a* $\{b\}$ *t*
| *Some X* $\Rightarrow$ *rbt_comp_insert c a* (*insert b X*) *t*)

abbreviation *add_option_to_rbt_comp f* $\equiv$ (λb _ *t*. *case f b of Some a* $\Rightarrow$ *add_to_rbt_comp* (*a*, *b*) *t* | *None* $\Rightarrow$ *t*)

definition *cluster_rbt_comp* :: (*'b* $\Rightarrow$ *'a option*) $\Rightarrow$ (*'b*, *unit*) *rbt* $\Rightarrow$ (*'a*, *'b set*) *rbt* **where**
cluster_rbt_comp f t = *RBT_Impl.fold* (*add_option_to_rbt_comp f*) *t RBT_Impl.Empty*

context
assumes *c*: *comparator c*
begin

lemma *add_to_rbt_comp*: *add_to_rbt_comp* = *ord.add_to_rbt* (*lt_of_comp c*)
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_ccompare'*[*OF assms*(*1*)]
by (*auto simp*: *c_def*)
have *c'*: *comparator* (*ccomp* :: '*b comparator*)
using *ID_ccompare'*[*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*) =
(λ*x*. *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_fmla ⇒ nat set where
  SP (Eqa (Var n) (Var n')) = (if n ≠ n' then {n, n'} else {})
| SP (Neg φ) = SP φ
| SP (Conj φ ψ) = SP φ ∪ SP ψ
| SP (Disj φ ψ) = SP φ ∪ SP ψ
| SP (Exists n φ) = SP φ − {n}
| SP (Forall n φ) = SP φ − {n}
| SP _ = {}

lemma SP_fv: SP φ ⊆ fv_fo_fmla φ
  by (induction φ rule: SP.induct) auto

lemma finite_SP: finite (SP φ)
  using SP_fv finite_fv_fo_fmla finite_subset by fastforce

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

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

lemma SP_list_set: set (SP_list φ) = SP φ
  unfolding SP_list_def
  by (induction φ rule: SP.induct) (auto simp: fv_fo_terms_set_list)

lemma sorted_distinct_SP_list: sorted_distinct (SP_list φ)
  unfolding SP_list_def
  by (auto intro: distinct_remdups_adj_sort)

fun d :: ('a, 'b) fo_fmla ⇒ nat where
  d (Eqa (Var n) (Var n')) = (if n ≠ n' then 2 else 1)
| d (Neg φ) = d φ
| d (Conj φ ψ) = max (d φ) (max (d ψ) (card (SP (Conj φ ψ))))
| d (Disj φ ψ) = max (d φ) (max (d ψ) (card (SP (Disj φ ψ))))
| d (Exists n φ) = d φ
```

| d (*Forall* n φ) = d φ
| d _ = *1*

lemma *d_pos*: *1* ≤ d φ
  by (*induction* φ *rule*: *d.induct*) *auto*

lemma *card_SP_d*: *card* (*SP* φ) ≤ d φ
  using *dual_order.trans*
  by (*induction* φ *rule*: *SP.induct*) (*fastforce simp*: *card_Diff1_le finite_SP*)+

fun *eval_eterm* :: ('*a* + '*c*) *val* ⇒ '*a fo_term* ⇒ '*a* + '*c* (**infix** ·*e 60*) **where**
  eval_eterm σ (*Const c*) = *Inl c*
| *eval_eterm* σ (*Var n*) = σ *n*

definition *eval_eterms* :: ('*a* + '*c*) *val* ⇒ ('*a fo_term*) *list* ⇒
  ('*a* + '*c*) *list* (**infix** ⊙*e 60*) **where**
  eval_eterms σ *ts* = *map* (*eval_eterm* σ) *ts*

lemma *eval_eterm_cong*: (⋀*n*. *n* ∈ *fv_fo_term_set t* ⟹ σ *n* = σ' *n*) ⟹
  eval_eterm σ *t* = *eval_eterm* σ' *t*
  by (*cases t*) *auto*

lemma *eval_eterms_fv_fo_terms_set*: σ ⊙*e ts* = σ' ⊙*e ts* ⟹ *n* ∈ *fv_fo_terms_set ts* ⟹ σ *n* = σ' *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*: (⋀*n*. *n* ∈ *fv_fo_terms_set ts* ⟹ σ *n* = σ' *n*) ⟹
  eval_eterms σ *ts* = *eval_eterms* σ' *ts*
  by (*auto simp*: *eval_eterms_def fv_fo_terms_set_def intro*: *eval_eterm_cong*)

lemma *eval_terms_eterms*: *map Inl* (σ ⊙ *ts*) = (*Inl* ∘ σ) ⊙*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* ⇒ ('*a* + '*c*) × ('*a* + '*c*) ⇒ *bool* **where**
  ad_equiv_pair X (*a*, *a*') ⟷ (*a* ∈ *Inl* ‘ *X* ⟶ *a* = *a*') ∧ (*a*' ∈ *Inl* ‘ *X* ⟶ *a* = *a*')

fun *sp_equiv_pair* :: '*a* × '*b* ⇒ '*a* × '*b* ⇒ *bool* **where**
  sp_equiv_pair (*a*, *b*) (*a*', *b*') ⟷ (*a* = *a*' ⟷ *b* = *b*')

definition *ad_equiv_list* :: '*a set* ⇒ ('*a* + '*c*) *list* ⇒ ('*a* + '*c*) *list* ⇒ *bool* **where**
  ad_equiv_list X xs ys ⟷ *length xs* = *length ys* ∧ (∀ *x* ∈ *set* (*zip xs ys*). *ad_equiv_pair X x*)

definition *sp_equiv_list* :: ('*a* + '*c*) *list* ⇒ ('*a* + '*c*) *list* ⇒ *bool* **where**
  sp_equiv_list xs ys ⟷ *length xs* = *length ys* ∧ *pairwise sp_equiv_pair* (*set* (*zip xs ys*))

definition *ad_agr_list* :: '*a set* ⇒ ('*a* + '*c*) *list* ⇒ ('*a* + '*c*) *list* ⇒ *bool* **where**
  ad_agr_list X xs ys ⟷ *length xs* = *length ys* ∧ *ad_equiv_list X xs ys* ∧ *sp_equiv_list xs ys*

lemma *ad_equiv_pair_refl*[*simp*]: *ad_equiv_pair X* (*a*, *a*)
  by *auto*

declare *ad_equiv_pair.simps*[*simp del*]

lemma *ad_equiv_pair_comm*: *ad_equiv_pair* X $(a,\ a') \longleftrightarrow$ *ad_equiv_pair* X $(a',\ a)$
  by (*auto simp*: *ad_equiv_pair.simps*)

lemma *ad_equiv_pair_mono*: $X \subseteq Y \Longrightarrow$ *ad_equiv_pair* Y $(a,\ a') \Longrightarrow$ *ad_equiv_pair* X $(a,\ a')$
  unfolding *ad_equiv_pair.simps*
  by *fastforce*

lemma *sp_equiv_pair_comm*: *sp_equiv_pair* x $y \longleftrightarrow$ *sp_equiv_pair* y x
  by (*cases* x; *cases* y) *auto*

definition *sp_equiv* :: $('a + 'c)$ *val* $\Rightarrow$ $('a + 'c)$ *val* $\Rightarrow$ *nat set* $\Rightarrow$ *bool* **where**
  sp_equiv σ τ $I \longleftrightarrow$ *pairwise sp_equiv_pair* $((\lambda n.\ (\sigma\ n,\ \tau\ n))$ ‘ $I)$

lemma *sp_equiv_mono*: $I \subseteq J \Longrightarrow$ *sp_equiv* σ τ $J \Longrightarrow$ *sp_equiv* σ τ I
  by (*auto simp*: *sp_equiv_def pairwise_def*)

definition *ad_agr_sets* :: *nat set* $\Rightarrow$ *nat set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $('a + 'c)$ *val* $\Rightarrow$
  $('a + 'c)$ *val* $\Rightarrow$ *bool* **where**
  ad_agr_sets FV S X σ $\tau \longleftrightarrow (\forall i \in FV.$ *ad_equiv_pair* X $(\sigma\ i,\ \tau\ i)) \wedge$ *sp_equiv* σ τ S

lemma *ad_agr_sets_comm*: *ad_agr_sets* FV S X σ $\tau \Longrightarrow$ *ad_agr_sets* FV S X τ σ
  unfolding *ad_agr_sets_def sp_equiv_def pairwise_def*
  by (*subst ad_equiv_pair_comm*) *auto*

lemma *ad_agr_sets_mono*: $X \subseteq Y \Longrightarrow$ *ad_agr_sets* FV S Y σ $\tau \Longrightarrow$ *ad_agr_sets* FV S X σ τ
  using *ad_equiv_pair_mono*
  by (*fastforce simp*: *ad_agr_sets_def*)

lemma *ad_agr_sets_mono'*: $S \subseteq S' \Longrightarrow$ *ad_agr_sets* FV S' X σ $\tau \Longrightarrow$ *ad_agr_sets* FV S X σ τ
  by (*auto simp*: *ad_agr_sets_def sp_equiv_def pairwise_def*)

lemma *ad_equiv_list_comm*: *ad_equiv_list* X *xs ys* $\Longrightarrow$ *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*: $X \subseteq Y \Longrightarrow$ *ad_equiv_list* Y *xs ys* $\Longrightarrow$ *ad_equiv_list* X *xs ys*
  using *ad_equiv_pair_mono*
  by (*fastforce simp*: *ad_equiv_list_def*)

lemma *ad_equiv_list_trans*:
  assumes *ad_equiv_list* X *xs ys ad_equiv_list* X *ys zs*
  shows *ad_equiv_list* X *xs zs*
proof −
  have *lens*: *length xs* = *length ys length xs* = *length zs length ys* = *length zs*
    using *assms*
    by (*auto simp*: *ad_equiv_list_def*)
  have $\bigwedge x\ z.\ (x,\ z) \in$ *set* (*zip xs zs*) $\Longrightarrow$ *ad_equiv_pair* X $(x,\ z)$
  proof −
    fix x z
    assume $(x,\ z) \in$ *set* (*zip xs zs*)
    then obtain i **where** *i_def*: $i <$ *length xs xs* ! $i = x$ *zs* ! $i = z$
      by (*auto simp*: *set_zip*)
    define y **where** $y =$ *ys* ! i
    have *ad_equiv_pair* X $(x,\ y)$ *ad_equiv_pair* X $(y,\ z)$
      using *assms lens i_def*
      by (*fastforce simp*: *set_zip y_def ad_equiv_list_def*)+

```
    then show ad_equiv_pair X (x, z)
      unfolding 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: (∀ i ∈ set ns. ad_equiv_pair X (σ i, τ i)) ⟷
  ad_equiv_list X (map σ ns) (map τ ns)
  by (auto simp: ad_equiv_list_def set_zip) (metis in_set_conv_nth nth_map)

lemma set_zip_comm: (x, y) ∈ set (zip xs ys) ⟹ (y, x) ∈ set (zip ys xs)
  by (metis in_set_zip prod.sel(1) prod.sel(2))

lemma set_zip_map: set (zip (map σ ns) (map τ ns)) = (λn. (σ n, τ n)) ` set ns
  by (induction ns) auto

lemma sp_equiv_list_comm: sp_equiv_list xs ys ⟹ 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 sp_equiv_list xs ys 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: Y ⊆ X ⟹ ad_agr_list Y ys xs ⟹
  Inl -` set xs ⊆ Y ⟹ Inl -` set ys ⊆ Y ⟹ ad_agr_list X ys xs
  apply (auto simp: ad_agr_list_def ad_equiv_list_def)
  subgoal for a b
    apply (drule bspec[of _ _ (a, b)])
     apply assumption
    apply (cases a; cases b)
       apply (auto simp: 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
  done

lemma ad_agr_list_trans: ad_agr_list X xs ys ⟹ ad_agr_list X ys zs ⟹ 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 ⟹ y ∈ X ⟹ Inl y ∈ set ys ⟹ Inl y ∈ 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 ⟹ length xs = length ys
  by (auto simp: ad_agr_list_def)

lemma ad_agr_list_eq: set ys ⊆ AD ⟹ ad_agr_list AD (map Inl xs) (map Inl ys) ⟹ 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 ⊆ set ns sp_equiv_list (map σ ns) (map σ' ns)
  shows sp_equiv_list (map σ ms) (map σ' ms)
  unfolding sp_equiv_list_def length_map pairwise_def
proof (rule conjI, rule refl, (rule ballI)+, rule impI)
  fix x y
  assume x ∈ set (zip (map σ ms) (map σ' ms)) y ∈ set (zip (map σ ms) (map σ' ms)) x ≠ y
  then have x ∈ set (zip (map σ ns) (map σ' ns)) y ∈ set (zip (map σ ns) (map σ' ns)) x ≠ 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 ⊆ set ns ⟹ ad_agr_list X (map σ ns) (map σ' ns) ⟹
  ad_agr_list X (map σ ms) (map σ' 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 σ τ ⟷
  ad_agr_list AD (map σ ns) (map τ 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 ⇒ 'a set ⇒ ('a + 'c) val ⇒ ('a + 'c) val ⇒ bool where
  ad_agr φ X σ τ ⟷ ad_agr_sets (fv_fo_fmla φ) (SP φ) X σ τ

lemma ad_agr_sets_restrict:
  ad_agr_sets (set (fv_fo_fmla_list φ)) (set (fv_fo_fmla_list φ)) AD σ τ ⟹ ad_agr φ AD σ τ
  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 ⟹ finite (Inl -` X)
  using finite_vimageI[of X Inl]
  by (auto simp: vimage_def)

lemma ex_out:
  assumes finite X
  shows ∃k. k ∉ X ∧ k < Suc (card X)
  using card_mono[OF assms, of {..<Suc (card X)}]
  by auto

lemma extend_τ:
  assumes ad_agr_sets (FV - {n}) (S - {n}) X σ τ S ⊆ FV finite S τ ` (FV - {n}) ⊆ Z
    Inl ` X ∪ Inr ` {..<max 1 (card (Inr -` τ ` (S - {n})) + (if n ∈ S then 1 else 0))} ⊆ Z
  shows ∃k ∈ Z. ad_agr_sets FV S X (σ(n := x)) (τ(n := k))
proof (cases n ∈ S)
  case True
  note n_in_S = True
  show ?thesis
  proof (cases x ∈ Inl ` X)
    case True
    show ?thesis
      apply (rule bexI[of _ x])
      using assms n_in_S True
       apply (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def)
      unfolding ad_equiv_pair.simps
          apply (metis True insert_Diff insert_iff subsetD)+
      done
  next
    case False
    note σ_n_not_Inl = False
    show ?thesis
    proof (cases ∃m ∈ S - {n}. x = σ m)
      case True
      obtain m where m_def: m ∈ S - {n} x = σ m
        using True
        by auto
      have τ_m_in: τ m ∈ Z
        using assms m_def
```

```
      by auto
    show ?thesis
      apply (rule bexI[of _ τ m])
      using assms n_in_S σ_n_not_Inl True m_def
      by (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def)
  next
    case False
    have out: x ∉ σ ` (S − {n})
      using False
      by auto
    have fin: finite (Inr −` τ ` (S − {n}))
      using assms(3)
      by (simp add: finite_vimageI)
    obtain k where k_def: Inr k ∉ τ ` (S − {n}) k < Suc (card (Inr −` τ ` (S − {n})))
      using ex_out[OF fin] True
      by auto
    show ?thesis
      apply (rule bexI[of _ Inr k])
      using assms n_in_S σ_n_not_Inl out k_def assms(5)
       apply (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def)
      unfolding ad_equiv_pair.simps
       apply fastforce
      apply (metis image_eqI insertE insert_Diff)
      done
  qed
 qed
next
 case False
 show ?thesis
  apply (cases x ∈ Inl ` X)
  subgoal
    apply (rule bexI[of _ x])
    using assms False
     apply (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def)
    done
  subgoal
    apply (rule bexI[of _ Inr 0])
    using assms False
     apply (auto simp: ad_agr_sets_def sp_equiv_def pairwise_def)
    unfolding ad_equiv_pair.simps
    apply fastforce
    done
  done
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'. t' ∈ set ts ⟹ sp_equiv_pair (σ ·e t, τ ·e t) (σ ·e t', τ ·e t')
      subgoal for t'
        apply (cases t')
        using c_X Const Cons(2)
        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)+
      done
    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'. t' ∈ set ts ⟹ sp_equiv_pair (σ ·e t, τ ·e t) (σ ·e t', τ ·e t')
      subgoal for t'
        apply (cases t')
        using Cons(2,3) sp_equiv_Var
         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
    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
    ad_agr_sets FV S AD σ τ ad_agr_list AD (σ ⊙e ts) vs
    ⋃(set_fo_term ‘ set ts) ⊆ AD
  shows ad_agr_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 ad_agr_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: ad_agr_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
  then have ad_agr_list:
    ad_agr_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)
    apply (auto simp: ad_agr_sets_def)
    subgoal for t
      by (cases t) (auto simp: ad_equiv_pair.simps intro!: fv_fo_terms_setI)
    done
  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_fmla ⇒ ('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_fmla :: ('a, 'b) fo_fmla ⇒ nat where
  sz_fmla (Neg φ) = Suc (sz_fmla φ)
| sz_fmla (Conj φ ψ) = Suc (sz_fmla φ + sz_fmla ψ)
| sz_fmla (Disj φ ψ) = Suc (sz_fmla φ + sz_fmla ψ)
| sz_fmla (Exists n φ) = Suc (sz_fmla φ)
| sz_fmla (Forall n φ) = Suc (Suc (Suc (Suc (sz_fmla φ))))
| sz_fmla _ = 0

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

$(\bigwedge r\ ts.\ P\ (Pred\ r\ ts)) \Longrightarrow (\bigwedge b.\ P\ (Bool\ b)) \Longrightarrow$
$(\bigwedge t\ t'.\ P\ (Eqa\ t\ t')) \Longrightarrow (\bigwedge \varphi.\ P\ \varphi \Longrightarrow P\ (Neg\ \varphi)) \Longrightarrow$
$(\bigwedge \varphi\ \psi.\ P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Conj\ \varphi\ \psi)) \Longrightarrow (\bigwedge \varphi\ \psi.\ P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Disj\ \varphi\ \psi)) \Longrightarrow$
$(\bigwedge n\ \varphi.\ P\ \varphi \Longrightarrow P\ (Exists\ n\ \varphi)) \Longrightarrow (\bigwedge n\ \varphi.\ P\ (Exists\ n\ (Neg\ \varphi)) \Longrightarrow P\ (Forall\ n\ \varphi)) \Longrightarrow P\ \varphi$
proof (*induction sz_fmla φ arbitrary: φ rule: nat_less_induct*)
  case *1*
  have *IH*: $\bigwedge\psi.\ sz\_fmla\ \psi < sz\_fmla\ \varphi \Longrightarrow P\ \psi$
    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*: $(\bigwedge n.\ n \in fv\_fo\_fmla\ \varphi \Longrightarrow \sigma\ n = \sigma'\ n) \Longrightarrow esat\ \varphi\ I\ \sigma\ X \longleftrightarrow esat\ \varphi\ I\ \sigma'\ X$
proof (*induction φ arbitrary: σ σ′ rule: sz_fmla_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 φ1 φ2*)
  show *?case*
    using *Conj*(*1*,*2*)[*of σ σ′*] *Conj*(*3*) **by** *auto*
next
  case (*Disj φ1 φ2*)
  show *?case*
    using *Disj*(*1*,*2*)[*of σ σ′*] *Disj*(*3*) **by** *auto*
next
  case (*Exists n φ*)
  show *?case*
  proof (*rule iffI*)
    assume *esat* (*Exists n φ*) *I σ X*
    then obtain *x* **where** *x_def*: $x \in X$ *esat φ I* $(\sigma(n := x))$ *X*
      by *auto*
    from *x_def*(*2*) **have** *esat φ 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 φ*) *I σ′ X*
      by *auto*
  next
    assume *esat* (*Exists n φ*) *I σ′ X*
    then obtain *x* **where** *x_def*: $x \in X$ *esat φ I* $(\sigma'(n := x))$ *X*
      by *auto*
    from *x_def*(*2*) **have** *esat φ 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 φ*) *I σ X*
      by *auto*
  qed
next

```
  case (Forall n φ)
  then show ?case
    by auto
qed auto

fun ad_terms :: ('a fo_term) list ⇒ 'a set where
  ad_terms ts = ⋃(set (map set_fo_term ts))

fun act_edom :: ('a, 'b) fo_fmla ⇒ ('a table, 'b) fo_intp ⇒ 'a set where
  act_edom (Pred r ts) I = ad_terms ts ∪ ⋃(set ‘ I (r, length ts))
| act_edom (Bool b) I = {}
| act_edom (Eqa t t') I = set_fo_term t ∪ set_fo_term t'
| act_edom (Neg φ) I = act_edom φ I
| act_edom (Conj φ ψ) I = act_edom φ I ∪ act_edom ψ I
| act_edom (Disj φ ψ) I = act_edom φ I ∪ act_edom ψ I
| act_edom (Exists n φ) I = act_edom φ I
| act_edom (Forall n φ) I = act_edom φ I

lemma finite_act_edom: wf_fo_intp φ I ⟹ finite (act_edom φ I)
  using finite_Inl
  by (induction φ I rule: wf_fo_intp.induct)
     (auto simp: finite_set_fo_term vimage_def)

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

theorem main: ad_agr φ AD σ τ ⟹ act_edom φ I ⊆ AD ⟹
  Inl ‘ AD ∪ Inr ‘ {..<d φ} ⊆ X ⟹ τ ‘ fv_fo_fmla φ ⊆ X ⟹
  esat φ I σ UNIV ⟷ esat φ I τ X
proof (induction φ arbitrary: σ τ rule: sz_fmla_induct)
  case (Pred r ts)
  have fv_sub: fv_fo_terms_set ts ⊆ fv_fo_fmla (Pred r ts)
    by auto
  have sub_AD: ⋃(set ‘ I (r, length ts)) ⊆ AD
    using Pred(2)
    by auto
  show ?case
    unfolding esat.simps
  proof (rule iffI)
    assume assm: σ ⊙e ts ∈ map Inl ‘ I (r, length ts)
    have σ ⊙e ts = τ ⊙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 τ ⊙e ts ∈ map Inl ‘ I (r, length ts)
      by auto
  next
    assume assm: τ ⊙e ts ∈ map Inl ‘ I (r, length ts)
    have τ ⊙e ts = σ ⊙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 σ ⊙e ts ∈ 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 σ m' = Inl c ⟷ τ 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 σ m = Inl c' ⟷ τ 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 σ m = σ m' ⟷ τ m = τ 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 φ)
  from Neg(2) have ad_agr φ AD σ τ
    by (auto simp: ad_agr_def)
  with Neg show ?case
    by auto
next
  case (Conj φ1 φ2)
  have aux: ad_agr φ1 AD σ τ ad_agr φ2 AD σ τ
    Inl ` AD ∪ Inr ` {..<d φ1} ⊆ X Inl ` AD ∪ Inr ` {..<d φ2} ⊆ X
    τ ` fv_fo_fmla φ1 ⊆ X τ ` fv_fo_fmla φ2 ⊆ 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 φ1 φ2)
  have aux: ad_agr φ1 AD σ τ ad_agr φ2 AD σ τ
    Inl ` AD ∪ Inr ` {..<d φ1} ⊆ X Inl ` AD ∪ Inr ` {..<d φ2} ⊆ X
    τ ` fv_fo_fmla φ1 ⊆ X τ ` fv_fo_fmla φ2 ⊆ 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* φ)
show *?case*
proof (*rule iffI*)
assume *esat* (*Exists m* φ) *I* σ *UNIV*
then obtain *x* **where** *assm*: *esat* φ *I* (σ(*m* := *x*)) *UNIV*
by *auto*
have *m* $\in$ *SP* φ $\Longrightarrow$ *Suc* (*card* (*Inr* −‘ τ ‘ (*SP* φ − {*m*}))) $\leq$ *card* (*SP* φ)
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* −‘ τ ‘ *SP* φ) $\leq$ *card* (*SP* φ)
by (*metis card_image finite_SP image_vimage_subset inj_Inr surj_card_le*)
ultimately have *max 1* (*card* (*Inr* −‘ τ ‘ (*SP* φ − {*m*})) + (*if m* $\in$ *SP* φ *then 1 else 0*)) $\leq$ *d* φ
using *d_pos card_SP_d*[*of* φ]
by *auto*
then have $\exists$ *x'* $\in$ *X*. *ad_agr* φ *AD* (σ(*m* := *x*)) (τ(*m* := *x'*))
using *extend_*τ[*OF Exists*(*2*)[*unfolded ad_agr_def fv_fo_fmla.simps SP.simps*]
SP_fv[*of* φ] *finite_SP Exists*(*5*)[*unfolded fv_fo_fmla.simps*]]
Exists(*4*)
by (*force simp*: *ad_agr_def*)
then obtain *x'* **where** *x'_def*: *x'* $\in$ *X ad_agr* φ *AD* (σ(*m* := *x*)) (τ(*m* := *x'*))
by *auto*
from *Exists*(*5*) **have** τ(*m* := *x'*) ‘ *fv_fo_fmla* φ $\subseteq$ *X*
using *x'_def*(*1*) **by** *fastforce*
then have *esat* φ *I* (τ(*m* := *x'*)) *X*
using *Exists x'_def*(*1*,*2*) *assm*
by *fastforce*
with *x'_def* **show** *esat* (*Exists m* φ) *I* τ *X*
by *auto*
next
assume *esat* (*Exists m* φ) *I* τ *X*
then obtain *z* **where** *assm*: *z* $\in$ *X esat* φ *I* (τ(*m* := *z*)) *X*
by *auto*
have *ad_agr*: *ad_agr_sets* (*fv_fo_fmla* φ − {*m*}) (*SP* φ − {*m*}) *AD* τ σ
using *Exists*(*2*)[*unfolded ad_agr_def fv_fo_fmla.simps SP.simps*]
by (*rule ad_agr_sets_comm*)
have $\exists$ *x*. *ad_agr* φ *AD* (σ(*m* := *x*)) (τ(*m* := *z*))
using *extend_*τ[*OF ad_agr SP_fv*[*of* φ] *finite_SP subset_UNIV subset_UNIV*] *ad_agr_sets_comm*
unfolding *ad_agr_def*
by *fastforce*
then obtain *x* **where** *x_def*: *ad_agr* φ *AD* (σ(*m* := *x*)) (τ(*m* := *z*))
by *auto*
have τ(*m* := *z*) ‘ *fv_fo_fmla* (*Exists m* φ) $\subseteq$ *X*
using *Exists*
by *fastforce*
with *x_def* **have** *esat* φ *I* (σ(*m* := *x*)) *UNIV*
using *Exists assm*
by *fastforce*
then show *esat* (*Exists m* φ) *I* σ *UNIV*
by *auto*
qed
next
case (*Forall n* φ)
have *unfold*: *act_edom* (*Forall n* φ) *I* = *act_edom* (*Exists n* (*Neg* φ)) *I*
Inl ‘ *AD* $\cup$ *Inr* ‘ {..<*d* (*Forall n* φ)} = *Inl* ‘ *AD* $\cup$ *Inr* ‘ {..<*d* (*Exists n* (*Neg* φ))}
fv_fo_fmla (*Forall n* φ) = *fv_fo_fmla* (*Exists n* (*Neg* φ))
by *auto*
have *pred*: *ad_agr* (*Exists n* (*Neg* φ)) *AD* σ τ
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 φ AD σ τ act_edom φ I ⊆ AD d φ ≤ n
    τ ` fv_fo_fmla φ ⊆ Inl ` AD ∪ Inr ` {..<n}
  shows esat φ I σ UNIV ⟷ esat φ I τ (Inl ` AD ∪ Inr ` {..<n})
proof -
  show ?thesis
    using main[OF assms(1,2) _ assms(4)] assms(3)
    by fastforce
qed

lemma esat_UNIV_cong:
  fixes σ :: nat ⇒ 'a + nat
  assumes ad_agr φ AD σ τ act_edom φ I ⊆ AD
  shows esat φ I σ UNIV ⟷ esat φ I τ UNIV
proof -
  show ?thesis
    using main[OF assms(1,2) subset_UNIV subset_UNIV]
    by auto
qed

lemma esat_UNIV_ad_agr_list:
  fixes σ :: nat ⇒ 'a + nat
  assumes ad_agr_list AD (map σ (fv_fo_fmla_list φ)) (map τ (fv_fo_fmla_list φ))
    act_edom φ I ⊆ AD
  shows esat φ I σ UNIV ⟷ esat φ I τ 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_fmla_list_set]] assms(2)] .

fun fo_rep :: ('a, 'c) fo_t ⇒ 'a table where
  fo_rep (AD, n, X) = {ts. ∃ts' ∈ X. ad_agr_list AD (map Inl ts) ts'}

lemma sat_esat_conv:
  fixes φ :: ('a :: infinite, 'b) fo_fmla
  assumes fin: wf_fo_intp φ I
  shows sat φ I σ ⟷ esat φ I (Inl ∘ σ :: nat ⇒ 'a + nat) UNIV
  using assms
proof (induction φ arbitrary: I σ rule: sz_fmla_induct)
  case (Pred r ts)
  show ?case
    unfolding sat.simps esat.simps comp_def[symmetric] eval_terms_eterms[symmetric]
    by auto
next
  case (Eqa t t')
  show ?case
    by (cases t; cases t') auto
next
  case (Exists n φ)
  show ?case
  proof (rule iffI)
    assume sat (Exists n φ) I σ
    then obtain x where x_def: esat φ I (Inl ∘ σ(n := x)) UNIV
      using Exists
```

```
      by fastforce
    have Inl_unfold: Inl ∘ σ(n := x) = (Inl ∘ σ)(n := Inl x)
      by auto
    show esat (Exists n φ) I (Inl ∘ σ) UNIV
      using x_def
      unfolding Inl_unfold
      by auto
  next
    assume esat (Exists n φ) I (Inl ∘ σ) UNIV
    then obtain x where x_def: esat φ I ((Inl ∘ σ)(n := x)) UNIV
      by auto
    show sat (Exists n φ) I σ
    proof (cases x)
      case (Inl a)
      have Inl_unfold: (Inl ∘ σ)(n := x) = Inl ∘ σ(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 ∉ act_edom φ I ∪ σ ` fv_fo_fmla φ
        using arb_element finite_act_edom[OF Exists(2), simplified] finite_fv_fo_fmla
        by (metis finite_Un finite_imageI)
      have wf_local: wf_fo_intp φ I
        using Exists(2)
        by auto
      have sat φ I (σ(n := c))
        apply (rule iffD2[OF Exists(1)[OF wf_local]
              iffD1[OF esat_UNIV_ad_agr_list[OF _ subset_refl] x_def[unfolded Inr]]])
        apply (auto simp: ad_agr_list_def ad_equiv_list_def fun_upd_def)
        subgoal for k l
          using c_def
          by (cases k; cases l) (auto simp: set_zip ad_equiv_pair.simps split: if_splits)
        using c_def[unfolded fv_fo_fmla_list_set[symmetric]]
        apply (auto simp: sp_equiv_list_def pairwise_def set_zip split: if_splits)
        done
      then show ?thesis
        by auto
    qed
  qed
next
  case (Forall n φ)
  show ?case
    using Forall(1)[of I σ] Forall(2)
    by auto
qed auto

lemma sat_ad_agr_list:
  fixes φ :: ('a :: infinite, 'b) fo_fmla
    and J :: (('a, nat) fo_t, 'b) fo_intp
  assumes wf_fo_intp φ I
    ad_agr_list AD (map (Inl ∘ σ :: nat ⇒ 'a + nat) (fv_fo_fmla_list φ))
      (map (Inl ∘ τ) (fv_fo_fmla_list φ)) act_edom φ I ⊆ AD
  shows sat φ I σ ⟷ sat φ I τ
  using esat_UNIV_ad_agr_list[OF assms(2,3)] sat_esat_conv[OF assms(1)]
  by auto
```

```
definition nfv :: ('a, 'b) fo_fmla ⇒ nat where
  nfv φ = length (fv_fo_fmla_list φ)

lemma nfv_card: nfv φ = card (fv_fo_fmla φ)
proof –
  have distinct (fv_fo_fmla_list φ)
    using sorted_distinct_fv_list
    by auto
  then have length (fv_fo_fmla_list φ) = card (set (fv_fo_fmla_list φ))
    using distinct_card by fastforce
  then show ?thesis
    unfolding fv_fo_fmla_list_set by (auto simp: nfv_def)
qed

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

lemma filter_rremdups_filter: filter P (rremdups (filter Q xs)) =
  rremdups (filter (λx. P x ∧ Q x) xs)
  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=λ_. True]
  by auto

lemma filter_take: ∃ 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: ∃ 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 ((≠) x) (take j (rremdups xs)) =
      take j' (filter ((≠) 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 ∈ 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* $\Longrightarrow$ *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* ⇒ *bool* **where**
  nats ns = (*ns* = [*0*..<*length ns*])

definition *fo_nmlzd* :: *'a set* ⇒ (*'a* + *nat*) *list* ⇒ *bool* **where**
  fo_nmlzd AD xs ⟷ *Inl* −' *set xs* ⊆ *AD* ∧
    (*let ns* = *List.map_filter* (*case_sum Map.empty Some*) *xs in nats* (*rremdups ns*))

lemma *fo_nmlzd_all_AD*:
  assumes *set xs* ⊆ *Inl* ' *AD*
  shows *fo_nmlzd AD xs*
proof −
  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*) ≤ *length xs*
proof −
  have *card* (*Inr* −' *set xs*) ≤ *card* (*set xs*)
    by (*meson List.finite_set card_inj_on_le image_vimage_subset inj_Inr*)
  moreover have . . . ≤ *length xs*
    by (*rule card_length*)
  finally show *?thesis* **.**
qed

lemma *fo_nmlzd_set*:
  assumes *fo_nmlzd AD xs*
  shows *set xs* = *set xs* ∩ *Inl* ' *AD* ∪ *Inr* ' {..<*min* (*length xs*) (*card* (*Inr* −' *set xs*))}
proof −
  have *Inl* −' *set xs* ⊆ *AD*
    using *assms*

```
    by (auto simp: fo_nmlzd_def)
  moreover have Inr -` set xs = {..<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 ∩ Inl ` AD ∪ Inr ` {..<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: ∃j. List.map_filter f (take i xs) = take j (List.map_filter f 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: fo_nmlzd AD xs ⟹ fo_nmlzd AD (take i xs)
  apply (auto simp: fo_nmlzd_def vimage_def nats_def Let_def)
  using set_take_subset apply fastforce
  using map_filter_take[of case_sum Map.empty Some i xs]
  apply auto
  subgoal for j
    using rremdups_take[of j List.map_filter (case_sum Map.empty Some) xs]
    by auto (metis (no_types, lifting) add.left_neutral min.cobounded1 min_def take_all take_upt)
  done

lemma map_filter_app: List.map_filter f (xs @ [x]) = List.map_filter f xs @
  (case f x of Some y ⇒ [y] | _ ⇒ [])
  by (induction xs) (auto simp: List.map_filter_simps split: option.splits)

lemma fo_nmlzd_app_Inr: Inr n ∉ set xs ⟹ Inr n' ∉ set xs ⟹ fo_nmlzd AD (xs @ [Inr n]) ⟹
  fo_nmlzd AD (xs @ [Inr n']) ⟹ 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 ⇒ nat ⇒ 'c table where
  all_tuples xs 0 = {[]}
| all_tuples xs (Suc n) = ⋃((λas. (λx. x # as) ` xs) ` (all_tuples xs n))

definition nall_tuples :: 'a set ⇒ nat ⇒ ('a + nat) table where
  nall_tuples AD n = {zs ∈ all_tuples (Inl ` AD ∪ Inr ` {..<n}) n. fo_nmlzd AD zs}

lemma all_tuples_finite: finite xs ⟹ finite (all_tuples xs n)
  by (induction xs n rule: all_tuples.induct) auto

lemma nall_tuples_finite: finite AD ⟹ finite (nall_tuples AD n)
  by (auto simp: nall_tuples_def all_tuples_finite)

lemma all_tuplesI: length vs = n ⟹ set vs ⊆ xs ⟹ vs ∈ 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 ⊆ xs w ∈ 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 ⟹ fo_nmlzd AD vs ⟹ vs ∈ nall_tuples AD n
  using fo_nmlzd_set[of AD vs]
  by (auto simp: nall_tuples_def intro!: all_tuplesI)

lemma all_tuplesD: vs ∈ all_tuples xs n ⟹ length vs = n ∧ set vs ⊆ xs
  by (induction xs n arbitrary: vs rule: all_tuples.induct) auto+

lemma all_tuples_setD: vs ∈ all_tuples xs n ⟹ set vs ⊆ xs
  by (auto dest: all_tuplesD)

lemma nall_tuplesD: vs ∈ nall_tuples AD n ⟹
  length vs = n ∧ set vs ⊆ Inl ‘ AD ∪ Inr ‘ {..<n} ∧ 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 ∧ set ys ⊆ 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 ∈ all_tuples xs (Suc n)
    then show ys ∈ {ys. length ys = Suc n ∧ set ys ⊆ 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 zs* = (*case zs of x* # *xs* ⇒ *x* # *add_nth n a xs*)

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

lemma *add_nth_take_drop*: *i* ≤ *length zs* ⟹ *add_nth i v zs* = *take i zs* @ *v* # *drop i zs*
by (*induction i v zs rule*: *add_nth.induct*) (*auto split*: *list.splits*)

lemma *add_nth_rem_nth_map*: *distinct xs* ⟹ *pos n xs* = *Some i* ⟹
add_nth i a (*rem_nth i* (*map* σ *xs*)) = *map* (σ(*n* := *a*)) *xs*
by (*induction xs arbitrary*: *i*) (*auto simp*: *pos_set split*: *option.splits*)

lemma *add_nth_rem_nth_self*: *i* < *length xs* ⟹ *add_nth i* (*xs* ! *i*) (*rem_nth i xs*) = *xs*
by (*induction i xs rule*: *rem_nth.induct*) *auto*

lemma *rem_nth_add_nth*: *i* ≤ *length zs* ⟹ *rem_nth i* (*add_nth i z zs*) = *zs*
by (*induction i z zs rule*: *add_nth.induct*) (*auto split*: *list.splits*)

fun *merge* :: (*nat* × *'a*) *list* ⇒ (*nat* × *'a*) *list* ⇒ (*nat* × *'a*) *list* **where**
merge [] *mys* = *mys*
| *merge nxs* [] = *nxs*
| *merge* ((*n*, *x*) # *nxs*) ((*m*, *y*) # *mys*) =
(*if n* ≤ *m then* (*n*, *x*) # *merge nxs* ((*m*, *y*) # *mys*)
else (*m*, *y*) # *merge* ((*n*, *x*) # *nxs*) *mys*)

lemma *merge_Nil2*[*simp*]: *merge nxs* [] = *nxs*
by (*cases nxs*) *auto*

lemma *merge_length*: *length* (*merge nxs mys*) = *length* (*map fst nxs* @ *map fst mys*)
by (*induction nxs mys rule*: *merge.induct*) *auto*

lemma *insort_aux_le*: ∀ *x*∈*set nxs*. *n* ≤ *fst x* ⟹ ∀ *x*∈*set mys*. *m* ≤ *fst x* ⟹ *n* ≤ *m* ⟹
insort n (*sort* (*map fst nxs* @ *m* # *map fst mys*)) = *n* # *sort* (*map fst nxs* @ *m* # *map fst mys*)
by (*induction nxs*) (*auto simp*: *insort_is_Cons insort_left_comm*)

lemma *insort_aux_gt*: ∀ *x*∈*set nxs*. *n* ≤ *fst x* ⟹ ∀ *x*∈*set mys*. *m* ≤ *fst x* ⟹ ¬ *n* ≤ *m* ⟹
insort n (*sort* (*map fst nxs* @ *m* # *map fst mys*)) =

```
    m # insort n (sort (map fst nxs @ map 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: sorted_distinct (map fst nxs) ⟹ sorted_distinct (map fst mys) ⟹
  map fst (merge nxs mys) = sort (map fst nxs @ map 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': sorted_distinct (map fst nxs) ⟹ sorted_distinct (map fst mys) ⟹
  fst ` set nxs ∩ fst ` set mys = {} ⟹
  map snd nxs = map σ (map fst nxs) ⟹ map snd mys = map σ (map fst mys) ⟹
  map snd (merge nxs mys) = map σ (sort (map fst nxs @ map 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: sorted_distinct ns ⟹ sorted_distinct ms ⟹ set ns ∩ set ms = {} ⟹
  map snd (merge (zip ns (map σ ns)) (zip ms (map σ ms))) = map σ (sort (ns @ ms))
  using merge_map'[of zip ns (map σ ns) zip ms (map σ ms) σ]
  by auto (metis length_map list.set_map map_fst_zip)

fun fo_nmlz_rec :: nat ⇒ ('a + nat ⇀ nat) ⇒ 'a set ⇒
  ('a + nat) list ⇒ ('a + nat) list where
  fo_nmlz_rec i m AD [] = []
| fo_nmlz_rec i m AD (Inl x # xs) = (if x ∈ AD then Inl x # fo_nmlz_rec i m AD xs 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 ⊆ {..<i} ⟹ filter ((≤) i) (rremdups
  (List.map_filter (case_sum Map.empty Some) (fo_nmlz_rec i m AD xs))) = ns ⟹
  ns = [i..<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)) ⊆ {..<Suc i}
        using 2(4) None
        by (auto simp: inj_on_def dom_def ran_def)
      have ns = i # filter ((≤) (Suc i)) (rremdups
        (List.map_filter (case_sum Map.empty Some) (fo_nmlz_rec (Suc i) (m(Inl x ↦ i)) AD xs)))
        using 2(5) False None
        by (auto simp: List.map_filter_simps filter_rremdups)
           (metis Suc_leD antisym not_less_eq_eq)
      then show ?thesis
        by (auto simp: 2(2)[OF False None pred, OF refl])
           (smt Suc_le_eq Suc_pred le_add1 le_zero_eq less_add_same_cancel1 not_less_eq_eq
            upt_Suc_append upt_rec)
    next
      case (Some j)
      then have j_lt_i: j < i
```

```
        using 2(4)
        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 2(5) False Some j_lt_i
        by (auto simp: List.map_filter_simps filter_rremdups) (metis leD)
      show ?thesis
        by (rule 2(3)[OF False Some 2(4) ns_def[symmetric]])
    qed
  qed (auto simp: List.map_filter_simps split: option.splits)
next
  case (3 i m AD n xs)
  show ?case
  proof (cases m (Inr n))
    case None
    have pred: ran (m(Inr n ↦ i)) ⊆ {..<Suc i}
      using 3(3) None
      by (auto simp: inj_on_def dom_def ran_def)
    have ns = i # filter ((≤) (Suc i)) (rremdups
      (List.map_filter (case_sum Map.empty Some) (fo_nmlz_rec (Suc i) (m(Inr n ↦ i)) AD xs)))
      using 3(4) None
      by (auto simp: List.map_filter_simps filter_rremdups) (metis Suc_leD antisym not_less_eq_eq)
    then show ?thesis
      by (auto simp add: 3(1)[OF None pred, OF refl])
        (smt Suc_le_eq Suc_pred le_add1 le_zero_eq less_add_same_cancel1 not_less_eq_eq
         upt_Suc_append upt_rec)
  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) ⊆ {..<i} ⟹ filter ((≤) i) ns = [i..<i + k] ⟹
  fo_nmlz_rec i (id_map i) AD ys = ys
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') ⊆ {..<Suc i}
        using Cons(2,3)
        by auto
      moreover have filter ((≤) (Suc i)) ns' = [Suc i..<i + k]
        using Cons(3,5)
        by (auto simp: Inr List.map_filter_simps j_i filter_rremdups[symmetric] ns'_def[symmetric])
           (smt One_nat_def Suc_eq_plus1 Suc_le_eq add_diff_cancel_left' diff_is_0_eq'
            dual_order.order_iff_strict filter_cong n_not_Suc_n upt_eq_Cons_conv)
      moreover have Inl -` set ys ⊆ AD
        using Cons(2)
        by (auto simp: vimage_def)
      ultimately have fo_nmlz_rec (Suc i) ((id_map i)(Inr i ↦ i)) AD ys = ys
        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)) ⊆ {..<Suc i}
        using 3(3)
        by (auto simp: ran_def)
      have set (Inr k # xs) − Inl ‘ AD − dom m =
        {Inr k} ∪ (set xs − Inl ‘ AD − dom (m(Inr k ↦ i)))
        using None
        by (auto simp: dom_def)
      then have Suc: Suc i + card (set xs − Inl ‘ AD − dom (m(Inr k ↦ i))) =
        i + card (set (Inr k # xs) − Inl ‘ AD − dom m)
```

using *None*
by *auto*
show *?thesis*
using *None 3(1)[OF None preds]*
unfolding *Suc*
by (*auto simp*: *fun_upd_def[symmetric] insert_Inr*)
next
case (*Some j*)
have *fin*: *finite* (*set* (*Inr k # xs*) − *Inl ‘ AD* − *dom m*)
by *auto*
have *card_eq*: *card* (*set xs* − *Inl ‘ AD* − *dom m*) = *card* (*set* (*Inr k # xs*) − *Inl ‘ AD* − *dom m*)
by (*auto simp*: *Some intro*!: *arg_cong[of _ _ card]*)
have *j_lt_i*: $j < i$
using *3(3) Some*
by (*auto simp*: *ran_def*)
show *?thesis*
using *3(2)[OF Some 3(3)] j_lt_i*
unfolding *card_eq*
by (*auto simp*: *ran_def insert_Inr Some*)
qed
qed *auto*

lemma *fo_nmlz_rec_set_rev*: *set* (*fo_nmlz_rec i m AD xs*) ⊆ *Inl ‘ AD* ⟹ *set xs* ⊆ *Inl ‘ AD*
by (*induction i m AD xs rule*: *fo_nmlz_rec.induct*) (*auto split*: *if_splits option.splits*)

lemma *fo_nmlz_rec_map*: *inj_on m* (*dom m*) ⟹ *ran m* ⊆ {..<*i*} ⟹ ∃ *m'*. *inj_on m'* (*dom m'*) ∧
(∀ *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 *auto*
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 *auto*
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'
        apply (auto simp: fun_upd_def intro!: exI[of _ m'] split: sum.splits)
        done
      done
  next
    case (Some j)
    show ?thesis
      using 3(2)[OF Some 3(3,4)] Some
      apply auto
      subgoal for m'
        by (auto simp: fun_upd_def intro!: exI[of _ m'] split: sum.splits)
      done
  qed
qed auto

lemma ad_agr_map: length xs = length ys ⟹ inj_on m (dom m) ⟹
  (⋀x y. (x, y) ∈ set (zip xs ys) ⟹ (case x of Inl x' ⇒
    if x' ∈ AD then x = y else m x = Some y ∧ (case y of Inl z ⇒ z ∉ AD | Inr _ ⇒ True)
  | Inr n ⇒ m x = Some y ∧ (case y of Inl z ⇒ z ∉ AD | Inr _ ⇒ True))) ⟹
  ad_agr_list AD xs ys
  apply (auto simp: ad_agr_list_def ad_equiv_list_def)
  subgoal premises prems for a b
    unfolding ad_equiv_pair.simps
    using prems(3)[OF prems(4)]
    by (auto split: sum.splits if_splits)
  apply (auto simp: sp_equiv_list_def pairwise_def)
  subgoal premises prems for a b c
    using prems(3)[OF prems(4)] prems(3)[OF prems(5)] prems(2,6)
    apply (auto split: sum.splits if_splits)
       apply (metis domI inj_onD prems(6))+
    done
  subgoal premises prems for a b c
    using prems(3)[OF prems(4)] prems(3)[OF prems(5)] prems(2,6)
    apply (auto split: sum.splits if_splits)
    done
  done

lemma fo_nmlz_rec_take: take n (fo_nmlz_rec i m AD xs) = fo_nmlz_rec i m AD (take n xs)
  by (induction i m AD xs arbitrary: n rule: fo_nmlz_rec.induct)
     (auto simp: take_Cons' split: option.splits)

definition fo_nmlz :: 'a set ⇒ ('a + nat) list ⇒ ('a + nat) list where
  fo_nmlz = fo_nmlz_rec 0 Map.empty

lemma fo_nmlz_Nil[simp]: fo_nmlz AD [] = []
  by (auto simp: fo_nmlz_def)

lemma fo_nmlz_Cons: fo_nmlz AD [x] =
```

(*case x of Inl x ⇒ if x ∈ AD then* [*Inl x*] *else* [*Inr 0*] | _ ⇒ [*Inr 0*])
by (*auto simp*: *fo_nmlz_def split*: *sum.splits*)

lemma *fo_nmlz_Cons_Cons*: *fo_nmlz AD* [*x*, *x*] =
(*case x of Inl x ⇒ if x ∈ AD then* [*Inl x*, *Inl x*] *else* [*Inr 0*, *Inr 0*] | _ ⇒ [*Inr 0*, *Inr 0*])
by (*auto simp*: *fo_nmlz_def split*: *sum.splits*)

lemma *fo_nmlz_sound*: *fo_nmlzd AD* (*fo_nmlz AD xs*)
using *fo_nmlz_rec_sound*[*of Map.empty 0*] *fo_nmlz_rec_set*[*of Map.empty 0 AD xs*]
by (*auto simp*: *fo_nmlzd_def fo_nmlz_def nats_def Let_def*)

lemma *fo_nmlz_length*: *length* (*fo_nmlz AD xs*) = *length xs*
using *fo_nmlz_rec_length*
by (*auto simp*: *fo_nmlz_def*)

lemma *fo_nmlz_map*: ∃ *τ*. *fo_nmlz AD* (*map σ ns*) = *map τ ns*
proof −
obtain *m′* **where** *m′_def*: ∀ (*x*, *y*)∈*set* (*zip* (*map σ ns*) (*fo_nmlz AD* (*map σ ns*))).
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*
using *fo_nmlz_rec_map*[*of Map.empty 0*, *of map σ ns*]
by (*auto simp*: *fo_nmlz_def*)
define *τ* **where** *τ* ≡ (*λn*. *case σ n of Inl x ⇒ if x ∈ AD then Inl x else Inr* (*the* (*m′* (*Inl x*)))
| *Inr j* ⇒ *Inr* (*the* (*m′* (*Inr j*))))
have *fo_nmlz AD* (*map σ ns*) = *map τ ns*
proof (*rule nth_equalityI*)
show *length* (*fo_nmlz AD* (*map σ ns*)) = *length* (*map τ ns*)
using *fo_nmlz_length*[*of AD map σ ns*]
by *auto*
fix *i*
assume *i* < *length* (*fo_nmlz AD* (*map σ ns*))
then show *fo_nmlz AD* (*map σ ns*) ! *i* = *map τ ns* ! *i*
using *m′_def fo_nmlz_length*[*of AD map σ ns*]
apply (*auto simp*: *set_zip τ_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*) ≤ *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 ∩ *Inl* ‘ *AD* ∪ *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*) ⊆ *Inl* ‘ *AD* ⟹ *set xs* ⊆ *Inl* ‘ *AD*
using *fo_nmlz_rec_set_rev*[*of 0 Map.empty AD xs*]
by (*auto simp*: *fo_nmlz_def*)

lemma *fo_nmlz_ad_agr*: *ad_agr_list AD xs* (*fo_nmlz AD xs*)
unfolding *fo_nmlz_def*
using *fo_nmlz_rec_map*[*of Map.empty 0 xs AD*]
apply *auto*

```
  subgoal for m'
    apply (rule ad_agr_map[OF fo_nmlz_rec_length[symmetric],
        of map_option Inr ∘ m' xs 0 Map.empty AD AD])
     apply (auto simp: inj_on_def dom_def split: sum.splits if_splits)
    done
  done

lemma fo_nmlzd_mono: Inl -` set xs ⊆ AD ⟹ fo_nmlzd AD' xs ⟹ fo_nmlzd AD xs
  by (auto simp: fo_nmlzd_def)

lemma fo_nmlz_idem: fo_nmlzd AD ys ⟹ 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 ⇒ nat ⇒ nat ⇒ ('a + nat) table where
  nall_tuples_rec AD i 0 = {[]}
| nall_tuples_rec AD i (Suc n) = ⋃((λas. (λx. x # as) ` (Inl ` AD ∪ Inr ` {..<i})) `
    nall_tuples_rec AD i n) ∪ (λas. Inr i # as) ` nall_tuples_rec AD (Suc i) n

lemma nall_tuples_rec_Inl: vs ∈ nall_tuples_rec AD i n ⟹ Inl -` set vs ⊆ AD
  by (induction AD i n arbitrary: vs rule: nall_tuples_rec.induct) (fastforce simp: vimage_def)+

lemma nall_tuples_rec_length: xs ∈ nall_tuples_rec AD i n ⟹ length xs = n
  by (induction AD i n arbitrary: xs rule: nall_tuples_rec.induct) auto

lemma fun_upd_id_map: id_map i(Inr i ↦ i) = id_map (Suc i)
  by (rule ext) (auto simp: id_map_def split: sum.splits)

lemma id_mapD: id_map j (Inr i) = None ⟹ j ≤ i id_map j (Inr i) = Some x ⟹ i < j ∧ i = x
  by (auto simp: id_map_def split: if_splits)

lemma nall_tuples_rec_fo_nmlz_rec_sound: i ≤ j ⟹ xs ∈ nall_tuples_rec AD i n ⟹
  fo_nmlz_rec j (id_map j) AD xs = xs
  apply (induction n arbitrary: i j xs)
   apply (auto simp: fun_upd_id_map dest!: id_mapD split: option.splits)
    apply (meson dual_order.strict_trans2 id_mapD(1) not_Some_eq sup.strict_order_iff)
  using Suc_leI apply blast+
  done

lemma nall_tuples_rec_fo_nmlz_rec_complete:
  assumes fo_nmlz_rec j (id_map j) AD xs = xs
  shows xs ∈ nall_tuples_rec AD j (length xs)
  using assms
proof (induction xs arbitrary: j)
  case (Cons x xs)
  show ?case
  proof (cases x)
    case (Inl a)
    have a_AD: a ∈ AD
      using Cons(2)
      by (auto simp: Inl split: if_splits option.splits)
    show ?thesis
      using Cons a_AD
      by (auto simp: Inl)
```

```
  next
    case (Inr b)
    have b_j: b ≤ j
      using Cons(2)
      by (auto simp: Inr split: option.splits dest: id_mapD)
    show ?thesis
    proof (cases b = j)
      case True
      have preds: fo_nmlz_rec (Suc j) (id_map (Suc j)) AD xs = xs
        using Cons(2)
        by (auto simp: Inr True fun_upd_id_map dest: id_mapD split: option.splits)
      show ?thesis
        using Cons(1)[OF preds]
        by (auto simp: Inr True)
    next
      case False
      have b_lt_j: b < j
        using b_j False
        by auto
      have id_map: id_map j (Inr b) = Some b
        using b_lt_j
        by (auto simp: id_map_def)
      have preds: fo_nmlz_rec j (id_map j) AD xs = xs
        using Cons(2)
        by (auto simp: 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 ∈ nall_tuples_rec AD 0 (length xs) ⟷ 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]: fo_nmlzd AD xs ⟷ fo_nmlz AD xs = xs
  using fo_nmlz_idem fo_nmlz_sound
  by metis

lemma nall_tuples_code[code]: nall_tuples AD n = 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: length xs = length ys ⟹ distinct xs ⟹ ∃f. ys = map f xs
proof (induction xs ys rule: list_induct2)
  case (Cons x xs y ys)
  then obtain f where f_def: ys = map f xs
    by auto
  with Cons(3) have y # ys = map (f(x := y)) (x # xs)
    by auto
  then show ?case
    by metis
qed auto

lemma exists_fo_nmlzd:
```

assumes *length xs* = *length ys distinct xs fo_nmlzd AD ys*
shows $\exists f$. *ys* = *fo_nmlz AD* (*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*]: *length xs* = *length ys* $\Longrightarrow$ (*P* [] []) $\Longrightarrow$
($\bigwedge$*x y xs ys*. *P xs ys* $\Longrightarrow$ *P* (*xs* @ [*x*]) (*ys* @ [*y*])) $\Longrightarrow$ *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 *ad_agr_list_fo_nmlzd*:
assumes *ad_agr_list AD vs vs'* *fo_nmlzd AD vs fo_nmlzd AD vs'*
shows *vs* = *vs'*
using *ad_agr_list_length*[*OF assms*(*1*)] *assms*
proof (*induction vs vs' rule*: *list_induct2_rev*)
case (*2 x y xs ys*)
have *norms*: *fo_nmlzd AD xs 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 *ad_agr*: *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* $\longrightarrow$ *projl x* $\in$ *AD isl y* $\longrightarrow$ *projl 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′ ⟷ 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 ⊆ AD′ Inl −‘ set xs ⊆ 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 ⇒ nat list ⇒ ′c table* **where**
proj_vals R ns = (*λτ. map τ ns*) ‘ *R*

definition *proj_fmla* :: (*′a*, *′b*) *fo_fmla ⇒ ′c val set ⇒ ′c table* **where**
proj_fmla φ R = proj_vals R (*fv_fo_fmla_list φ*)

lemmas *proj_fmla_map = proj_fmla_def*[*unfolded proj_vals_def*]

definition *extends_subst σ τ* = (∀ *x. σ x ≠ None ⟶ σ x = τ x*)

definition *ext_tuple* :: *′a set ⇒ nat list ⇒ nat list ⇒*
(*′a + nat*) *list ⇒* (*′a + nat*) *list set* **where**
ext_tuple AD fv_sub fv_sub_comp as = (*if fv_sub_comp* = [] *then* {*as*}
else (*λfs. 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 ⟹*
ext_tuple AD fv_sub fv_sub_comp as =
(*λfs. 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 ⟹ distinct xs ⟹*

```
  ys = map (the ∘ (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 ⟹
  i' = card (Inr −` (Inr ` {..<i} ∪ set (take n xs))) ⟹ n ≤ length xs ⟹
  fo_nmlz_rec i' (id_map i') AD (drop n xs) = drop n xs
proof (induction i id_map i :: 'a + nat ⇀ nat AD xs arbitrary: n rule: fo_nmlz_rec.induct)
  case (2 i AD x xs)
  have preds: x ∈ 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 ≤ length xs
      using 2(6)
      by (auto simp: Suc)
    have i'_def: i' = card (Inr −` (Inr ` {..<i} ∪ 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 ` {..<Suc i} ∪ set (take k xs)))
        using 3(4)
        by (auto simp: Suc True inj_vimage_image_eq)
          (metis Un_insert_left image_insert inj_Inr inj_vimage_image_eq lessThan_Suc vimage_Un)
      show ?thesis
        using 3(1)[OF id_map norm_xs i'_def k_le]
        by (auto simp: Suc)
```

```
    next
      case False
      have id_map: id_map i (Inr j) = Some j
        using j_le_i False
        by (auto simp: id_map_def)
      have norm_xs: fo_nmlz_rec i (id_map i) AD xs = xs
        using 3(3)
        by (auto simp: id_map)
      have i'_def: i' = card (Inr -` (Inr ` {..<i} ∪ set (take k xs)))
        using 3(4) j_le_i False
        by (auto simp: Suc inj_vimage_image_eq insert_absorb)
      show ?thesis
        using 3(2)[OF id_map norm_xs i'_def k_le]
        by (auto simp: Suc)
    qed
  qed (auto simp: inj_vimage_image_eq)
qed auto

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: proj_tuple (map fst nxs) nxs = map snd nxs
  by (induction nxs) auto

lemma proj_tuple_merge: sorted_distinct (map fst nxs) ⟹ sorted_distinct (map fst mys) ⟹
  set (map fst nxs) ∩ set (map fst mys) = {} ⟹
  proj_tuple (map fst nxs) (merge nxs mys) = map snd nxs
  using proj_tuple_idle
  by (induction nxs mys rule: merge.induct) auto+

lemma proj_tuple_map:
  assumes sorted_distinct ns sorted_distinct ms set ns ⊆ set ms
  shows proj_tuple ns (zip ms (map σ ms)) = map σ ns
proof -
  define ns' where ns' = filter (λn. n ∉ set ns) ms
  have sd_ns': sorted_distinct ns'
    using assms(2) sorted_filter[of id]
    by (auto simp: ns'_def)
  have disj: set ns ∩ set ns' = {}
    by (auto simp: ns'_def)
  have ms_def: ms = sort (ns @ ns')
    apply (rule sorted_distinct_set_unique)
    using assms
    by (auto simp: ns'_def)
  have zip: zip ms (map σ ms) = merge (zip ns (map σ ns)) (zip ns' (map σ 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 *sorted_distinct ns sorted_distinct ms set ns* $\subseteq$ *set ms length ms* = *length xs*
  shows *length* (*proj_tuple ns* (*zip ms xs*)) = *length ns*
proof –
  obtain σ **where** σ: *xs* = *map* σ *ms*
    using *exists_map*[*OF assms*(*4*)] *assms*(*2*)
    by *auto*
  show *?thesis*
    unfolding σ
    by (*auto simp*: *proj_tuple_map*[*OF assms*(*1*–*3*)])
qed

lemma *ext_tuple_sound*:
  assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
    set fv_sub $\cap$ *set fv_sub_comp* = {} *set fv_sub* $\cup$ *set fv_sub_comp* = *set fv_all*
    ass = *fo_nmlz AD* ' *proj_vals R fv_sub*
    $\bigwedge\sigma\ \tau$. *ad_agr_sets* (*set fv_sub*) (*set fv_sub*) *AD* σ τ $\Longrightarrow$ $\sigma \in R \longleftrightarrow \tau \in R$
    xs $\in$ *fo_nmlz AD* ' $\bigcup$(*ext_tuple AD fv_sub fv_sub_comp* ' *ass*)
  shows *fo_nmlz AD* (*proj_tuple fv_sub* (*zip fv_all xs*)) $\in$ *ass*
    xs $\in$ *fo_nmlz AD* ' *proj_vals R fv_all*
proof –
  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*)
  have *len_in_ass*: $\bigwedge$*xs*. *xs* $\in$ *ass* $\Longrightarrow$ *xs* = *fo_nmlz AD xs* $\wedge$ *length xs* = *length 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* $\in$ *ass*
    fs $\in$ *nall_tuples_rec AD* (*card* (*Inr* –' *set as*)) (*length 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 σ **where** σ*_def*: $\sigma \in R$ *as* = *fo_nmlz AD* (*map* σ *fv_sub*)
    using *as_fs_def*(*1*) *assms*(*6*)
    by (*auto simp*: *proj_vals_def*)
  then obtain τ **where** τ*_def*: *as* = *map* τ *fv_sub ad_agr_list AD* (*map* σ *fv_sub*) (*map* τ *fv_sub*)
    using *fo_nmlz_map fo_nmlz_ad_agr*
    by *metis*
  have τ*_R*: $\tau \in R$
    using *assms*(*7*) *ad_agr_list_link* σ*_def*(*1*) τ*_def*(*2*)
    by *fastforce*
  define σ' **where** $\sigma' \equiv \lambda n$. *if n* $\in$ *set fv_sub_comp then the* (*map_of* (*zip fv_sub_comp fs*) *n*)
      else τ *n*
  then have $\forall$ *n* $\in$ *set fv_sub*. τ *n* = σ' *n*
    using *assms*(*4*) **by** *auto*
  then have σ'*_S*: $\sigma' \in R$
    using *assms*(*7*) τ*_R*
    by (*fastforce simp*: *ad_agr_sets_def sp_equiv_def pairwise_def ad_equiv_pair.simps*)
  have *length_as*: *length as* = *length fv_sub*
    using *as_fs_def*(*1*) *assms*(*6*)
    by (*auto simp*: *proj_vals_def fo_nmlz_length*)
  have *length_fs*: *length fs* = *length fv_sub_comp*
    using *as_fs_def*(*2*)
    by (*auto simp*: *nall_tuples_rec_length*)
  have *map_fv_sub*: *map* σ' *fv_sub* = *map* τ *fv_sub*

using $assms(4)\ \tau\_def(2)$
by $(auto\ simp: \sigma'\_def)$
have $fs\_map\_map\_of: fs = map\ (the \circ (map\_of\ (zip\ fv\_sub\_comp\ fs)))\ fv\_sub\_comp$
using $map\_map\_of\ length\_fs\ assms(2)$
by $metis$
have $fs\_map: fs = map\ \sigma'\ fv\_sub\_comp$
using $\sigma'\_def\ length\_fs$ **by** $(subst\ fs\_map\_map\_of)\ simp$
have $vs\_map\_fv\_all: xs = fo\_nmlz\ AD\ (map\ \sigma'\ fv\_all)$
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 $xs \in fo\_nmlz\ AD\ `\ proj\_vals\ R\ fv\_all$
using $\sigma'\_S\ vs\_map\_fv\_all$
by $(auto\ simp: proj\_vals\_def)$
obtain σ'' **where** $\sigma''\_def: xs = map\ \sigma''\ fv\_all$
using $exists\_map[of\ fv\_all\ xs]\ fo\_nmlz\_map\ vs\_map\_fv\_all$
by $blast$
have $proj: proj\_tuple\ fv\_sub\ (zip\ fv\_all\ xs) = map\ \sigma''\ fv\_sub$
using $proj\_tuple\_map\ assms(1,3,5)$
unfolding σ''_def
by $blast$
have $\sigma''\_\sigma': fo\_nmlz\ AD\ (map\ \sigma''\ fv\_sub) = as$
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 $fo\_nmlz\ AD\ (proj\_tuple\ fv\_sub\ (zip\ fv\_all\ xs)) \in ass$
unfolding $proj\ \sigma''\_\sigma'\ map\_fv\_sub$
by $(rule\ as\_fs\_def(1))$
qed

lemma $ext\_tuple\_complete$:
assumes $sorted\_distinct\ fv\_sub\ sorted\_distinct\ fv\_sub\_comp\ sorted\_distinct\ fv\_all$
$set\ fv\_sub \cap set\ fv\_sub\_comp = \{\}\ set\ fv\_sub \cup set\ fv\_sub\_comp = set\ fv\_all$
$ass = fo\_nmlz\ AD\ `\ proj\_vals\ R\ fv\_sub$
$\bigwedge\sigma\ \tau.\ ad\_agr\_sets\ (set\ fv\_sub)\ (set\ fv\_sub)\ AD\ \sigma\ \tau \Longrightarrow \sigma \in R \longleftrightarrow \tau \in R$
$xs = fo\_nmlz\ AD\ (map\ \sigma\ fv\_all)\ \sigma \in R$
shows $xs \in fo\_nmlz\ AD\ `\ \bigcup(ext\_tuple\ AD\ fv\_sub\ fv\_sub\_comp\ `\ ass)$
proof −
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 $vs\_norm: fo\_nmlzd\ AD\ xs$
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 $len\_fv\_sub\_nos: length\ fv\_sub \le length\ nos$
by $(auto\ simp: nos\_def\ fo\_nmlz\_length\ as\_def)$
have $norm\_as': fo\_nmlzd\ AD\ as'$
using $fo\_nmlzd\_take[OF\ fo\_nmlz\_sound]$
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 τ **where** *τ_def*: *as′ = map τ fv_sub fs′ = map τ 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 ≤ 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 ⊆ Inl ‘ AD ∪ 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′ ∈ 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 ad_agr_list_subset*[*OF equalityD1*, *OF set_sort′*])
using *fo_nmlz_ad_agr*[*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*: $A \subseteq B \Longrightarrow$ *ext_tuple_set AD ns ns'* $A \subseteq$ *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 $\cap$ *set fv_sub_comp* $= \{\}$ *set fv_sub* $\cup$ *set fv_sub_comp* $=$ *set fv_all*
ass $=$ *fo_nmlz AD ‘ proj_vals R fv_sub*
$\bigwedge\sigma\ \tau.$ *ad_agr_sets* (*set fv_sub*) (*set fv_sub*) *AD* $\sigma\ \tau \Longrightarrow \sigma \in R \longleftrightarrow \tau \in R$
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* $\in$ *ext_tuple_set AD fv_sub fv_sub_comp ass*
show *xs* $\in$ *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*] *image_iff*
split: *if_splits*)
next
fix *xs*
assume *xs* $\in$ *fo_nmlz AD ‘ proj_vals R fv_all*
then obtain σ **where** σ*_def*: *xs* $=$ *fo_nmlz AD* (*map* σ *fv_all*) $\sigma \in R$
by (*auto simp*: *proj_vals_def*)
show *xs* $\in$ *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*
$\bigwedge\sigma\ \tau.$ *ad_agr_sets* (*set fv_sub*) (*set fv_sub*) *AD* $\sigma\ \tau \Longrightarrow \sigma \in R \longleftrightarrow \tau \in R$
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 σ **where** σ*_def*: *xs* $=$ *map* σ *fv_all*
using *exists_map*[*of fv_all xs*] *assms*(*3*,*9*)
by *auto*
have *xs_norm*: *xs* $=$ *fo_nmlz AD* (*map* σ *fv_all*)
using *assms*(*8*)
by (*auto simp*: σ*_def*)
have *proj*: *proj_tuple fv_sub* (*zip fv_all xs*) $=$ *map* σ *fv_sub*
unfolding σ*_def*
apply (*rule proj_tuple_map*[*OF assms*(*1*,*3*)])
using *assms*(*5*)
by *blast*
obtain τ **where** τ*_def*: *fo_nmlz AD* (*map* σ *fv_sub*) $=$ *fo_nmlz AD* (*map* τ *fv_sub*) $\tau \in R$
using *assms*(*10*)
by (*auto simp*: *assms*(*6*) *proj proj_vals_def*)
have σ*_R*: $\sigma \in R$
using *assms*(*7*) *fo_nmlz_eqD*[*OF* τ*_def*(*1*)] τ*_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 σ_R*]) *assumption*
qed

lemma *proj_tuple_correct*:
assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
set fv_sub ∩ *set fv_sub_comp* = {} *set fv_sub* ∪ *set fv_sub_comp* = *set fv_all*
ass = *fo_nmlz AD* ‘ *proj_vals R fv_sub*
⋀*σ τ*. *ad_agr_sets* (*set fv_sub*) (*set fv_sub*) *AD σ τ* ⟹ *σ* ∈ *R* ⟷ *τ* ∈ *R*
fo_nmlz AD xs = *xs length xs* = *length fv_all*
shows *xs* ∈ *fo_nmlz AD* ‘ ⋃(*ext_tuple AD fv_sub fv_sub_comp* ‘ *ass*) ⟷
fo_nmlz AD (*proj_tuple fv_sub* (*zip fv_all xs*)) ∈ *ass*
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* ⇒ (*'a fo_term*) *list* ⇒ (*nat* ⇀ (*'a* + *'c*)) ⇒
(*nat* ⇀ (*'a* + *'c*)) *option* **where**
unify_vals_terms [] [] *σ* = *Some σ*
| *unify_vals_terms* (*v* # *vs*) ((*Const c'*) # *ts*) *σ* =
(*if v* = *Inl c' then unify_vals_terms vs ts σ else None*)
| *unify_vals_terms* (*v* # *vs*) ((*Var n*) # *ts*) *σ* =
(*case σ n of Some x* ⇒ (*if v* = *x then unify_vals_terms vs ts σ else None*)
| *None* ⇒ *unify_vals_terms vs ts* (*σ*(*n* := *Some v*)))
| *unify_vals_terms* _ _ _ = *None*

lemma *unify_vals_terms_extends*: *unify_vals_terms vs ts σ* = *Some σ'* ⟹ *extends_subst σ σ'*
unfolding *extends_subst_def*
by (*induction vs ts σ arbitrary*: *σ' rule*: *unify_vals_terms.induct*)
(*force split*: *if_splits option.splits*)+

lemma *unify_vals_terms_sound*: *unify_vals_terms vs ts σ* = *Some σ'* ⟹ (*the* ∘ *σ'*) ⊙*e ts* = *vs*
using *unify_vals_terms_extends*
by (*induction vs ts σ arbitrary*: *σ' 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*: *σ''* ⊙*e ts* = *vs* ⟹ (⋀*n*. *σ n* ≠ *None* ⟹ *σ n* = *Some* (*σ'' n*)) ⟹
∃*σ'*. *unify_vals_terms vs ts σ* = *Some σ'*
by (*induction vs ts σ 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* ⇒ (*'a* + *'c*) *table* ⇒ (*'a* + *'c*) *table* **where**
eval_table ts X = (*let fvs* = *fv_fo_terms_list ts in*
⋃((*λvs*. *case unify_vals_terms vs ts Map.empty of Some σ* ⇒
{*map* (*the* ∘ *σ*) *fvs*} | _ ⇒ {}) ‘ *X*))

lemma *eval_table*:
fixes *X* :: (*'a* + *'c*) *table*
shows *eval_table ts X* = *proj_vals* {*σ*. *σ* ⊙*e ts* ∈ *X*} (*fv_fo_terms_list ts*)
proof (*rule set_eqI*, *rule iffI*)
fix *vs*
assume *vs* ∈ *eval_table ts X*
then obtain *as σ* **where** *as_def*: *as* ∈ *X unify_vals_terms as ts Map.empty* = *Some σ*
vs = *map* (*the* ∘ *σ*) (*fv_fo_terms_list ts*)
by (*auto simp*: *eval_table_def split*: *option.splits*)
have (*the* ∘ *σ*) ⊙*e ts* ∈ *X*

```
    using unify_vals_terms_sound[OF as_def(2)] as_def(1)
    by auto
  with as_def(3) show vs ∈ proj_vals {σ. σ ⊙e ts ∈ X} (fv_fo_terms_list ts)
    by (fastforce simp: proj_vals_def)
next
  fix vs :: ('a + 'c) list
  assume vs ∈ proj_vals {σ. σ ⊙e ts ∈ X} (fv_fo_terms_list ts)
  then obtain σ where σ_def: vs = map σ (fv_fo_terms_list ts) σ ⊙e ts ∈ X
    by (auto simp: proj_vals_def)
  obtain σ' where σ'_def: unify_vals_terms (σ ⊙e ts) ts Map.empty = Some σ'
    using unify_vals_terms_complete[OF refl, of Map.empty σ ts]
    by auto
  have (the ∘ σ') ⊙e ts = (σ ⊙e ts)
    using unify_vals_terms_sound[OF σ'_def(1)]
    by auto
  then have vs = map (the ∘ σ') (fv_fo_terms_list ts)
    using fv_fo_terms_set_list eval_eterms_fv_fo_terms_set
    unfolding σ_def(1)
    by fastforce
  then show vs ∈ eval_table ts X
    using σ_def(2) σ'_def
    by (force simp: eval_table_def)
qed

fun ad_agr_close_rec :: nat ⇒ (nat ⇀ 'a + nat) ⇒ 'a set ⇒
  ('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 ⇒ z ∉ X | Inr x ⇒ 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 ad_agr_close_rec_complete:
  fixes xs :: ('a + nat) list
  shows 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 ⟹
  (⋀n b. (Inr n, b) ∈ set (zip xs ys) ⟹ case m n of Some v ⇒ v = b | None ⇒ b ∉ ran m) ⟹
  fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) ys = ys ⟹ ad_agr_list X xs ys ⟹
  ys ∈ ad_agr_close_rec i m AD xs
proof (induction j id_map j :: 'a + nat ⇒ 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 ∈ 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: ad_agr_list_def ad_equiv_list_def ad_equiv_pair.simps)
  have norm_zs: fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) zs = zs
```

```
      using 2(13) ys_def(2) x_X(1)
      by (auto simp: ys_def(1))
    have ad_agr: ad_agr_list X xs zs
      using 2(14)
      by (auto simp: ys_def ad_agr_list_def ad_equiv_list_def sp_equiv_list_def pairwise_def)
    show ?case
      using 2(1)[OF x_X 2(5,6,7,8,9,10,11) _ norm_zs ad_agr] 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: ad_agr_list_def)
      done
    show ?case
    proof (cases j ≤ 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 ↦ 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 ad_agr: ad_agr_list X xs zs
      using 3(13)
      by (auto simp: ys_def ad_agr_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 ≤ 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 ∪ Inr ` {..<Suc i}
      using 3(9) None
      by (auto simp: m'_def)
    have zs ∈ ad_agr_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 ad_agr])
      using 3(11,13)
      unfolding 3(9) ys_def Inr i_b m'_def
      unfolding ran[unfolded m'_def i_b]
      apply (auto simp: ad_agr_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 ∪ Inr ` {..<i}
      using 3(9) Some
      by (auto simp: ran_def)
    have ad_agr: ad_agr_list X xs zs
      using 3(13)
      by (auto simp: ad_agr_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 ∪ AD
        using 3(12,13)
        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)
      show ?thesis
        using 3(2)[OF id_map norm_xs 3(4,5,6,7,8,9,10) _ norm_zs ad_agr] 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 ad_agr] 3(11)
        by (auto simp: ys_def Inr Some)
    qed
  qed
qed (auto simp: ad_agr_list_def)

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

lemma ad_agr_close_sound:
  assumes ys ∈ ad_agr_close Y xs fo_nmlzd X xs X ∩ Y = {}
  shows fo_nmlzd (X ∪ Y) ys ∧ ad_agr_list X xs ys
  using ad_agr_close_rec_sound[OF assms(1)[unfolded ad_agr_close_def]
    fo_nmlz_idem[OF assms(2), unfolded fo_nmlz_def, folded id_map_empty] assms(3)
    Int_empty_right Int_empty_left]
    ad_agr_map[OF ad_agr_close_rec_length[OF assms(1)[unfolded ad_agr_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 ad_agr_close_complete:
  assumes X ∩ Y = {} fo_nmlzd X xs fo_nmlzd (X ∪ Y) ys ad_agr_list X xs ys
  shows ys ∈ ad_agr_close Y xs
  using ad_agr_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 *ad_agr_close_def*
by (*auto simp*: *fo_nmlz_def*)

lemma *ad_agr_close_empty*: *fo_nmlzd X xs* $\Longrightarrow$ *ad_agr_close* {} *xs* = {*xs*}
using *ad_agr_close_complete*[**where** *?X=X* **and** *?Y*={} **and** *?xs=xs* **and** *?ys=xs*]
ad_agr_close_sound[**where** *?X=X* **and** *?Y*={} **and** *?xs=xs*] *ad_agr_list_refl ad_agr_list_fo_nmlzd*
by *fastforce*

lemma *ad_agr_close_set_correct*:
assumes $AD' \subseteq AD$ *sorted_distinct ns*
$\bigwedge\sigma\ \tau.$ *ad_agr_sets* (*set ns*) (*set ns*) $AD'\ \sigma\ \tau \Longrightarrow \sigma \in R \longleftrightarrow \tau \in R$
shows $\bigcup$(*ad_agr_close* $(AD - AD')$ ‘ *fo_nmlz* AD' ‘ *proj_vals R ns*) = *fo_nmlz AD* ‘ *proj_vals R ns*
proof (*rule set_eqI*, *rule iffI*)
fix *vs*
assume *vs* $\in \bigcup$(*ad_agr_close* $(AD - AD')$ ‘ *fo_nmlz* AD' ‘ *proj_vals R ns*)
then obtain σ **where** *σ_def*: *vs* $\in$ *ad_agr_close* $(AD - AD')$ (*fo_nmlz* AD' (*map* σ *ns*)) $\sigma \in R$
by (*auto simp*: *proj_vals_def*)
have *vs*: *fo_nmlzd AD vs ad_agr_list* AD' (*fo_nmlz* AD' (*map* σ *ns*)) *vs*
using *ad_agr_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*: *ad_agr_list_def fo_nmlz_length*)
show *vs* $\in$ *fo_nmlz AD* ‘ *proj_vals R ns*
apply (*subst fo_nmlz_idem*[*OF vs*(*1*), *symmetric*])
using *iffD1*[*OF assms*(*3*) *σ_def*(*2*), *OF iffD2*[*OF ad_agr_list_link ad_agr_list_trans*[*OF*
fo_nmlz_ad_agr[*of* AD' *map* σ *ns*] *vs*(*2*), *unfolded τ_def*]]]
unfolding *τ_def*
by (*auto simp*: *proj_vals_def*)
next
fix *vs*
assume *vs* $\in$ *fo_nmlz AD* ‘ *proj_vals R ns*
then obtain σ **where** *σ_def*: *vs* = *fo_nmlz AD* (*map* σ *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*: *σ_def*(*1*) *xs_def*)+
obtain τ **where** *τ_def*: *vs* = *map* τ *ns*
using *exists_map*[*of ns vs*] *assms*(*2*) *σ_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 τ_def*
using *iffD1*[*OF assms*(*3*) *σ_def*(*2*), *OF ad_agr_sets_mono*[*OF assms*(*1*) *iffD2*[*OF ad_agr_list_link*
fo_nmlz_ad_agr[*of AD map* σ *ns*, *folded σ_def*(*1*), *unfolded τ_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* φ)) (*set* (*fv_fo_fmla_list* φ)) $AD'\ \sigma\ \tau \Longrightarrow$
$\sigma \in R \longleftrightarrow \tau \in R$
shows $\bigcup$(*ad_agr_close* $(AD - AD')$ ‘ *fo_nmlz* AD' ‘ *proj_fmla* φ *R*) = *fo_nmlz AD* ‘ *proj_fmla* φ *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* ⋃(*ad_agr_close AD ‘ X*))

lemma *ad_agr_close_set_eq*: *Ball X* (*fo_nmlzd AD′*) ⟹ *ad_agr_close_set AD X* = ⋃(*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* ⇒ ′*a table* ⇒ (′*a*, ′*c*) *fo_t* **where**
eval_pred ts X = (*let AD* = ⋃(*set* (*map set_fo_term ts*)) ∪ ⋃(*set ‘ X*) *in*
(*AD*, *length* (*fv_fo_terms_list ts*), *eval_table ts* (*map Inl ‘ X*)))

definition *eval_bool* :: *bool* ⇒ (′*a*, ′*c*) *fo_t* **where**
eval_bool b = (*if b then* ({}, *0*, {[]}) *else* ({}, *0*, {}))

definition *eval_eq* :: ′*a fo_term* ⇒ ′*a fo_term* ⇒ (′*a*, *nat*) *fo_t* **where**
eval_eq t t′ = (*case t of Var n* ⇒
(*case t*′ *of Var n*′ ⇒
if n = *n*′ *then* ({}, *1*, {[*Inr 0*]})
else ({}, *2*, {[*Inr 0*, *Inr 0*]})
| *Const c*′ ⇒ ({*c*′}, *1*, {[*Inl c*′]}))
| *Const c* ⇒
(*case t*′ *of Var n*′ ⇒ ({*c*}, *1*, {[*Inl c*]})
| *Const c*′ ⇒ *if c* = *c*′ *then* ({*c*}, *0*, {[]}) *else* ({*c*, *c*′}, *0*, {})))

fun *eval_neg* :: *nat list* ⇒ (′*a*, *nat*) *fo_t* ⇒ (′*a*, *nat*) *fo_t* **where**
eval_neg ns (*AD*, _, *X*) = (*AD*, *length ns*, *nall_tuples AD* (*length ns*) − *X*)

definition *eval_conj_tuple AD nsφ nsψ xs ys* =
(*let cxs* = *filter* (λ(*n*, *x*). *n* ∉ *set nsψ* ∧ *isl x*) (*zip nsφ xs*);
nxs = *map fst* (*filter* (λ(*n*, *x*). *n* ∉ *set nsψ* ∧ ¬*isl x*) (*zip nsφ xs*));
cys = *filter* (λ(*n*, *y*). *n* ∉ *set nsφ* ∧ *isl y*) (*zip nsψ ys*);
nys = *map fst* (*filter* (λ(*n*, *y*). *n* ∉ *set nsφ* ∧ ¬*isl y*) (*zip nsψ ys*)) *in*
fo_nmlz AD ‘ ext_tuple {} (*sort* (*nsφ* @ *map fst cys*)) *nys* (*map snd* (*merge* (*zip nsφ xs*) *cys*)) ∩
fo_nmlz AD ‘ ext_tuple {} (*sort* (*nsψ* @ *map fst cxs*)) *nxs* (*map snd* (*merge* (*zip nsψ ys*) *cxs*)))

definition *eval_conj_set AD nsφ Xφ nsψ Xψ* = ⋃((λ*xs*. ⋃(*eval_conj_tuple AD nsφ nsψ xs ‘ Xψ*)) ‘ *Xφ*)

definition *idx_join AD ns nsφ Xφ nsψ Xψ* =
(*let idxφ*′ = *cluster* (*Some* ∘ (λ*xs*. *fo_nmlz AD* (*proj_tuple ns* (*zip nsφ xs*)))) *Xφ*;
idxψ′ = *cluster* (*Some* ∘ (λ*ys*. *fo_nmlz AD* (*proj_tuple ns* (*zip nsψ ys*)))) *Xψ in*
set_of_idx (*mapping_join* (λ*Xφ*′′ *Xψ*′′. *eval_conj_set AD nsφ Xφ*′′ *nsψ Xψ*′′) *idxφ*′ *idxψ*′))

fun *eval_conj* :: *nat list* ⇒ (′*a*, *nat*) *fo_t* ⇒ *nat list* ⇒ (′*a*, *nat*) *fo_t* ⇒
(′*a*, *nat*) *fo_t* **where**
eval_conj nsφ (*ADφ*, _, *Xφ*) *nsψ* (*ADψ*, _, *Xψ*) = (*let AD* = *ADφ* ∪ *ADψ*; *ADΔφ* = *AD* − *ADφ*; *ADΔψ* = *AD* − *ADψ*; *ns* = *filter* (λ*n*. *n* ∈ *set nsψ*) *nsφ in*
(*AD*, *card* (*set nsφ* ∪ *set nsψ*), *idx_join AD ns nsφ* (*ad_agr_close_set ADΔφ Xφ*) *nsψ* (*ad_agr_close_set ADΔψ Xψ*)))

fun *eval_ajoin* :: *nat list* ⇒ (′*a*, *nat*) *fo_t* ⇒ *nat list* ⇒ (′*a*, *nat*) *fo_t* ⇒
(′*a*, *nat*) *fo_t* **where**

eval_ajoin nsφ (ADφ, _, Xφ) nsψ (ADψ, _, Xψ) = (let AD = ADφ ∪ ADψ; ADΔφ = AD − ADφ; ADΔψ = AD − ADψ;
ns = filter (λn. n ∈ set nsψ) nsφ; nsφ′ = filter (λn. n ∉ set nsφ) nsψ;
idxφ = cluster (Some ∘ (λxs. fo_nmlz ADψ (proj_tuple ns (zip nsφ xs)))) (ad_agr_close_set ADΔφ Xφ);
idxψ = cluster (Some ∘ (λys. fo_nmlz ADψ (proj_tuple ns (zip nsψ ys)))) Xψ in
(AD, card (set nsφ ∪ set nsψ), set_of_idx (Mapping.map_values (λxs X. case Mapping.lookup idxψ xs of Some Y ⇒
idx_join AD ns nsφ X nsψ (ad_agr_close_set ADΔψ (ext_tuple_set ADψ ns nsφ′ {xs} − Y)) | _ ⇒ ext_tuple_set AD nsφ nsφ′ X) idxφ)))

fun *eval_disj* :: *nat list ⇒ (′a, nat) fo_t ⇒ nat list ⇒ (′a, nat) fo_t ⇒*
(′a, nat) fo_t **where**
eval_disj nsφ (ADφ, _, Xφ) nsψ (ADψ, _, Xψ) = (let AD = ADφ ∪ ADψ;
nsφ′ = filter (λn. n ∉ set nsφ) nsψ;
nsψ′ = filter (λn. n ∉ set nsψ) nsφ;
ADΔφ = AD − ADφ; ADΔψ = AD − ADψ in
(AD, card (set nsφ ∪ set nsψ),
ext_tuple_set AD nsφ nsφ′ (ad_agr_close_set ADΔφ Xφ) ∪
ext_tuple_set AD nsψ nsψ′ (ad_agr_close_set ADΔψ Xψ)))

fun *eval_exists* :: *nat ⇒ nat list ⇒ (′a, nat) fo_t ⇒ (′a, nat) fo_t* **where**
eval_exists i ns (AD, _, X) = (case pos i ns of Some j ⇒
(AD, length ns − 1, fo_nmlz AD ‘ rem_nth j ‘ X)
| None ⇒ (AD, length ns, X))

fun *eval_forall* :: *nat ⇒ nat list ⇒ (′a, nat) fo_t ⇒ (′a, nat) fo_t* **where**
eval_forall i ns (AD, _, X) = (case pos i ns of Some j ⇒
let n = card AD in
(AD, length ns − 1, Mapping.keys (Mapping.filter (λt Z. n + card (Inr −‘ set t) + 1 ≤ card Z)
(cluster (Some ∘ (λts. fo_nmlz AD (rem_nth j ts))) X)))
| None ⇒ (AD, length ns, X))

lemma *combine_map2*: **assumes** *length ys = length xs length ys′ = length xs′*
distinct xs distinct xs′ set xs ∩ set xs′ = {}
shows *∃f. ys = map f xs ∧ ys′ = map f xs′*
proof −
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 _ λx. if x ∈ 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 ∩ set xs′ = {} set xs ∩ set xs′′ = {} set xs′ ∩ set xs′′ = {}
shows *∃f. ys = map f xs ∧ ys′ = map f xs′ ∧ ys′′ = map f xs′′*
proof −
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 _ λx. if x ∈ set xs then f x else if x ∈ set xs′ then g x else h x]*)
qed

lemma *distinct_set_zip*: *length nsx = length xs ⟹ distinct nsx ⟹*

(a, b) ∈ set (zip nsx xs) ⟹ (a, ba) ∈ set (zip nsx xs) ⟹ b = ba
by (induction nsx xs rule: list_induct2) (auto dest: set_zip_leftD)

lemma fo_nmlz_idem_isl:
assumes ⋀x. x ∈ set xs ⟹ (case x of Inl z ⇒ z ∈ X | _ ⇒ False)
shows fo_nmlz X xs = xs
proof −
have F1: Inl x ∈ set xs ⟹ x ∈ 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 ∈ set xs ⟹ (f x, g x) ∈ 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 ∈ 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 ⊆ set nsx ∩ set nsy
fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsy ys)) ∨
(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))
xs′ ∈ ad_agr_close ((X ∪ Y) − X) xs ys′ ∈ ad_agr_close ((X ∪ Y) − Y) ys
shows eval_conj_tuple (X ∪ Y) nsx nsy xs′ ys′ = {}
proof −
define cxs **where** cxs = filter (λ(n, x). n ∉ set nsy ∧ isl x) (zip nsx xs′)
define nxs **where** nxs = map fst (filter (λ(n, x). n ∉ set nsy ∧ ¬isl x) (zip nsx xs′))
define cys **where** cys = filter (λ(n, y). n ∉ set nsx ∧ isl y) (zip nsy ys′)
define nys **where** nys = map fst (filter (λ(n, y). n ∉ set nsx ∧ ¬isl y) (zip nsy ys′))
define both **where** both = sorted_list_of_set (set nsx ∪ set nsy)
have close: fo_nmlzd (X ∪ Y) xs′ ad_agr_list X xs xs′ fo_nmlzd (X ∪ 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 $zs \in fo\_nmlz\ (X \cup Y)$ ` $(\lambda fs.\ map\ snd\ (merge\ (zip\ (sort\ (nsx\ @\ map\ fst\ cys))\ (map\ snd\ (merge\ (zip\ nsx\ xs')\ cys)))\ (zip\ nys\ fs)))$ `
    $nall\_tuples\_rec\ \{\}\ (card\ (Inr\ -`\ set\ (map\ snd\ (merge\ (zip\ nsx\ xs')\ cys))))\ (length\ nys)$
  $zs \in fo\_nmlz\ (X \cup Y)$ ` $(\lambda fs.\ map\ snd\ (merge\ (zip\ (sort\ (nsy\ @\ map\ fst\ cxs))\ (map\ snd\ (merge\ (zip\ nsy\ ys')\ cxs)))\ (zip\ nxs\ fs)))$ `
    $nall\_tuples\_rec\ \{\}\ (card\ (Inr\ -`\ set\ (map\ snd\ (merge\ (zip\ nsy\ ys')\ cxs))))\ (length\ nxs)$
  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_map_fst_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 *xsf* **where** *xsf_def*: $xs = map\ xsf\ nsx$
    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** *?σ=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* ∪ *Y*) (*map xf both*)
using *merge_map*[**where** *σ=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** *?σ=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* ∪ *Y*) (*map xg both*)
using *merge_map*[**where** *σ=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 xsf ns proj_tuple ns* (*zip nsy ys*) = *map ysf ns*
unfolding *xf_def*(*2*) *xg_def*(*2*) *xsf_def ysf_def*
using *assms*(*5*,*6*,*7*,*8*) *proj_tuple_map*
by *auto*
have *ad_agr_list* (*X* ∪ *Y*) (*map xg both*) (*map xf both*)
using *zs_xg zs_xf*
by (*fastforce dest*: *fo_nmlz_eqD*)
then have *ad_agr_list* (*X* ∪ *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*) $\neq$ *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*)
have *case x of Inl z* $\Rightarrow$ *z* $\in$ *X* $\cup$ *Y* | *Inr b* $\Rightarrow$ *False* **if** *x* $\in$ *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*=*xsf* **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* $\cup$ *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* $\Rightarrow$ *z* $\in$ *X* $\cup$ *Y* | *Inr b* $\Rightarrow$ *False* **if** *y* $\in$ *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* $\cup$ *Y*) (*proj_tuple ns* (*zip nsy ys'*)) = *proj_tuple ns* (*zip nsy ys'*)
by (*rule fo_nmlz_idem_isl*)
have *ad*: *ad_agr_list X* (*map xsf ns*) (*map xg ns*)
using *assms*(*8*) *close*(*2*)[*unfolded xsf_def xg_def*(*2*)] *ad_agr_list_subset*
by *blast*
have $\forall$ *x*$\in$*set* (*proj_tuple ns* (*zip nsx xs*)). *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 *ad_agr_list_fo_nmlzd_isl*[*OF close*(*2*)[*unfolded xsf_def xg_def*(*2*)] *assms*(*1*)[*unfolded xsf_def*]]
by *auto*
have $\forall$ *x*$\in$*set* (*proj_tuple ns* (*zip nsy ys*)). *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 *ad_agr_list_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 (λn. n ∈ set nsy) nsx
    fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsy ys))
    xs' ∈ ad_agr_close ((X ∪ Y) − X) xs ys' ∈ ad_agr_close ((X ∪ Y) − Y) ys
  shows eval_conj_tuple (X ∪ Y) nsx nsy xs' ys' = {}
proof −
  have aux: sorted_distinct ns set ns ⊆ set nsx ∩ 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 ⊆ set nsx ∩ set nsy
    fo_nmlz Z (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz Z (proj_tuple ns (zip nsy ys)) ∨
      (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))
  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 (λn. n ∈ set nsy) nsx
    fo_nmlz Z (proj_tuple ns (zip nsx xs)) ≠ 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 ⊆ set nsx ∩ 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 ∈ nall_tuples_rec AD n (length xs) ys = filter (λx. ¬isl x) xs
  shows ys ∈ 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 ≤ n
```

```
      using Cons(2)
      by (auto simp: Inr)
    obtain zs where ys_def: ys = Inr b # zs zs = filter (λx. ¬ 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 ∈ nall_tuples_rec {} n (length ys) ys = filter (λx. ¬isl x) xs
    Inl −‘ set xs ⊆ AD
  shows xs ∈ 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 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)) 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 ⋀x. Inl x ∈ set xs ∪ set fsφ ⟹ x ∈ AD ⋀y. Inl y ∈ set ys ∪ set fsψ ⟹ y ∈ AD
    using xs_ys_def fo_nmlz_set[of AD] nall_tuples_rec_Inl[OF fsφ_def]
      nall_tuples_rec_Inl[OF fsψ_def]
    by (auto simp: Xφ_def Xψ_def)
  then have Inl_xs_ys:
    ⋀n. n ∈ set nsφ ∪ set nsψ ⟹ isl (σxs n) ⟷ (∃x. σxs n = Inl x ∧ x ∈ AD)
    ⋀n. n ∈ set nsφ ∪ set nsψ ⟹ isl (σys n) ⟷ (∃y. σys n = Inl y ∧ y ∈ AD)
    unfolding σxs_def σys_def nsφ'_def nsψ'_def
    by (auto simp: isl_def) (smt imageI mem_Collect_eq)+
  have sort_sort: sort (nsφ @ nsφ') = sort (nsψ @ nsψ')
    apply (rule sorted_distinct_set_unique)
    using distinct
    by (auto simp: nsφ'_def nsψ'_def)
  have isl_iff: ⋀n. n ∈ set nsφ' ∪ set nsψ' ⟹ isl (σxs n) ∨ isl (σys n) ⟹ σxs n = σys 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 ⋀n. n ∈ set (map fst cys) ⟹ isl (σxs n)
    ⋀n. n ∈ set (map fst cxs) ⟹ isl (σys 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: Inr −` set (map σxs (sort (nsφ @ map fst cys))) = Inr −` 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: map σxs nys = filter (λx. ¬isl x) fsφ
```

using *isl_iff*[*unfolded nsφ′_def*]
unfolding *nys_def σys_def*(*1*) *σxs_def*(*2*) *nsφ′_def filter_map*
by (*induction nsψ*) *force+*
have *map_nys_in_nall*: *map σxs nys* ∈ *nall_tuples_rec* {} (*card* (*Inr* −‘ *set xs*)) (*length nys*)
using *nall_tuples_rec_filter*[*OF fsφ_def*[*folded len_fsφ*] *map_nys*]
by *auto*
have *map_cys*: *map snd cys* = *map σxs* (*map fst cys*)
using *isl_iff*
by (*auto simp*: *cys_def set_zip nsφ′_def σys_def*(*1*)) (*metis nth_mem*)
show *merge_xs_cys*: *map snd* (*merge* (*zip nsφ xs*) *cys*) = *map σxs* (*sort* (*nsφ* @ *map 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_prems*: *sorted_distinct* (*sort* (*nsφ* @ *map fst cys*)) *sorted_distinct nys*
set (*sort* (*nsφ* @ *map fst cys*)) ∩ *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*: *map σxs* (*sort* (*sort* (*nsφ* @ *map fst cys*) @ *nys*)) =
map snd (*merge* (*zip* (*sort* (*nsφ* @ *map fst cys*)) (*map σxs* (*sort* (*nsφ* @ *map fst cys*))))
(*zip nys* (*map σxs nys*)))
by (*rule merge_map*[*OF merge_nys_prems*, *symmetric*])
have *sort_sort_nys*: *sort* (*sort* (*nsφ* @ *map fst cys*) @ *nys*) = *sort* (*nsφ* @ *nsφ′*)
apply (*rule sorted_distinct_set_unique*)
using *distinct merge_nys_prems 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* (λ*n*. *n* ∉ *set nsφ*) *nsψ*
and *nsψ′_def*: *nsψ′* = *filter* (λ*n*. *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*: *⋀x. Inl x ∈ set xs ∪ set (map snd cys) ∪ set ysψ ⟹ x ∈ AD*
⋀y. Inl y ∈ set ys ∪ set (map snd cxs) ∪ set xsφ ⟹ y ∈ 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*:
⋀n. n ∈ set nsφ′ ∪ set nsψ′ ⟹ isl (σxs n) ⟷ (∃x. σxs n = Inl x ∧ x ∈ AD)
⋀n. n ∈ set nsφ′ ∪ set nsψ′ ⟹ isl (σys n) ⟷ (∃y. σys n = Inl y ∧ y ∈ AD)
using *set_σ_ns*
by *(auto simp: isl_def rev_image_eqI)*
have *sort_sort*: *sort (nsφ @ nsφ′) = sort (nsψ @ nsψ′)*
apply *(rule sorted_distinct_set_unique)*
using *distinct*
by *(auto simp: nsφ′_def nsψ′_def)*
have *isl_iff*: *⋀n. n ∈ set nsφ′ ∪ set nsψ′ ⟹ isl (σxs n) ∨ isl (σys n) ⟹ σxs n = σys 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 *⋀n. n ∈ set (map fst cys) ⟹ isl (σxs n)*
⋀n. n ∈ set (map fst cxs) ⟹ isl (σys 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: Inr -` set (map σxs (sort (nsφ @ map fst cys))) = Inr -` 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: map σxs nys = filter (λx. ¬isl x) fsφ
    using isl_iff[unfolded nsφ'_def]
    unfolding nys_def σys_def(1) fsφ_def nsφ'_def
    by (induction nsψ) force+
  have map_cys: map snd cys = map σxs (map fst cys)
    using isl_iff
    by (auto simp: cys_def set_zip nsφ'_def σys_def(1)) (metis nth_mem)
  show merge_xs_cys: map snd (merge (zip nsφ xs) cys) = map σxs (sort (nsφ @ map 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_prems: sorted_distinct (sort (nsφ @ map fst cys)) sorted_distinct nys
    set (sort (nsφ @ map fst cys)) ∩ 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: map σxs (sort (sort (nsφ @ map fst cys) @ nys)) =
    map snd (merge (zip (sort (nsφ @ map fst cys)) (map σxs (sort (nsφ @ map fst cys))))
     (zip nys (map σxs nys)))
    by (rule merge_map[OF merge_nys_prems, symmetric])
  have sort_sort_nys: sort (sort (nsφ @ map fst cys) @ nys) = sort (nsφ @ nsφ')
    apply (rule sorted_distinct_set_unique)
    using distinct merge_nys_prems set_ns
    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 fsφ_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
  have Inl -` set fsφ ⊆ AD
    using Inl_sub_AD(1) set_σ_ns
    by (force simp: fsφ_def)
  then show fsφ ∈ nall_tuples_rec AD (card (Inr -` set xs)) (length nsφ')
    unfolding len_fsφ[symmetric]
    using nall_tuples_rec_filter_rev[OF _ map_nys] ysψ_def[unfolded Inr_sort]
    by auto
qed

lemma eval_conj_set_correct:
  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
  shows eval_conj_set AD nsφ Xφ nsψ Xψ = ext_tuple_set AD nsφ nsφ' Xφ ∩ ext_tuple_set AD nsψ
nsψ' Xψ
```

proof −
have aux: ext_tuple_set AD nsφ nsφ′ Xφ = fo_nmlz AD ‘ ⋃(ext_tuple AD nsφ nsφ′ ‘ Xφ)
ext_tuple_set AD nsψ nsψ′ Xψ = fo_nmlz AD ‘ ⋃(ext_tuple AD nsψ nsψ′ ‘ Xψ)
by (auto simp: ext_tuple_set_def ext_tuple_def Xφ_def Xψ_def image_iff fo_nmlz_idem[OF fo_nmlz_sound])
show ?thesis
unfolding aux
proof (rule set_eqI, rule iffI)
fix vs
assume vs ∈ fo_nmlz AD ‘ ⋃(ext_tuple AD nsφ nsφ′ ‘ Xφ) ∩
fo_nmlz AD ‘ ⋃(ext_tuple AD nsψ nsψ′ ‘ Xψ)
then obtain xs ys **where** xs_ys_def: xs ∈ Xφ vs ∈ fo_nmlz AD ‘ ext_tuple AD nsφ nsφ′ xs
ys ∈ Xψ vs ∈ fo_nmlz AD ‘ ext_tuple AD nsψ nsψ′ ys
by auto
have len_xs_ys: length xs = length nsφ length ys = length nsψ
using xs_ys_def(1,3)
by (auto simp: Xφ_def Xψ_def proj_vals_def fo_nmlz_length)
obtain fsφ **where** fsφ_def: vs = fo_nmlz AD (map snd (merge (zip nsφ xs) (zip nsφ′ fsφ)))
fsφ ∈ nall_tuples_rec AD (card (Inr −‘ set xs)) (length nsφ′)
using xs_ys_def(1,2)
by (auto simp: Xφ_def proj_vals_def ext_tuple_def split: if_splits)
(metis fo_nmlz_map length_map map_snd_zip)
obtain fsψ **where** fsψ_def: vs = fo_nmlz AD (map snd (merge (zip nsψ ys) (zip nsψ′ fsψ)))
fsψ ∈ nall_tuples_rec AD (card (Inr −‘ set ys)) (length nsψ′)
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_fsψ = nall_tuples_rec_length[OF fsψ_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 fsψ = map σys nsψ′
using exists_map[of nsψ @ nsψ′ ys @ fsψ] len_xs_ys(2) len_fsψ 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_fsψ: map snd (merge (zip nsψ ys) (zip nsψ′ fsψ)) = 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] fsψ_def(1)],
unfolded map_merge_fsφ map_merge_fsψ]
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) fsψ_def(2) ad_agr2]
by blast
obtain σys **where** aux2:
map snd (merge (zip nsψ ys) (zip nsψ' fsψ)) =
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 fsψ_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 fsψ_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) image_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 map_fst_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 map_fst_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* ∉ *fv_fo_fmla φ* ⟹ *X* ≠ {} ⟹
esat (*Exists n φ*) *I σ X* ⟷ *esat φ I σ X*
proof (*rule iffI*)
assume *assms*: *n* ∉ *fv_fo_fmla φ esat* (*Exists n φ*) *I σ X*
then obtain *x* **where** *esat φ I* (*σ*(*n* := *x*)) *X*
by *auto*
with *assms*(*1*) **show** *esat φ I σ X*
using *esat_fv_cong*[*of φ σ σ*(*n* := *x*)] **by** *fastforce*
next
assume *assms*: *n* ∉ *fv_fo_fmla φ X* ≠ {} *esat φ I σ X*
from *assms*(*2*) **obtain** *x* **where** *x_def*: *x* ∈ *X*
by *auto*
with *assms*(*1*,*3*) **have** *esat φ I* (*σ*(*n* := *x*)) *X*
using *esat_fv_cong*[*of φ σ σ*(*n* := *x*)] **by** *fastforce*
with *x_def* **show** *esat* (*Exists n φ*) *I σ X*
by *auto*
qed

```
lemma esat_forall_not_fv: n ∉ fv_fo_fmla φ ⟹ X ≠ {} ⟹
  esat (Forall n φ) I σ X ⟷ esat φ I σ X
  using esat_exists_not_fv[of n Neg φ X I σ]
  by auto

lemma proj_sat_vals: proj_sat φ I =
  proj_vals {σ. sat φ I σ} (fv_fo_fmla_list φ)
  by (auto simp: proj_sat_def proj_vals_def)

lemma fv_fo_fmla_list_Pred: remdups_adj (sort (fv_fo_terms_list ts)) = 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': ⋃(set (map set_fo_term ts)) ⊆ X ⟹
  ad_agr_list X (map σ (fv_fo_terms_list ts)) (map τ (fv_fo_terms_list ts)) ⟹
  ad_agr_list X (σ ⊙e ts) (τ ⊙e ts)
proof (induction ts)
  case (Cons t ts)
  have IH: ad_agr_list X (σ ⊙e ts) (τ ⊙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: ⋀i. i ∈ fv_fo_term_set t ∪ ⋃(fv_fo_term_set ‘ set ts) ⟹
    ad_equiv_pair X (σ i, τ i)
    using Cons(3)
    by (auto simp: ad_agr_list_def ad_equiv_list_link[symmetric] fv_fo_terms_set_list
        fv_fo_terms_set_def)
  have sp_equiv: ⋀i j. i ∈ fv_fo_term_set t ∪ ⋃(fv_fo_term_set ‘ set ts) ⟹
    j ∈ fv_fo_term_set t ∪ ⋃(fv_fo_term_set ‘ set ts) ⟹ sp_equiv_pair (σ i, τ i) (σ j, τ j)
    using Cons(3)
    by (auto simp: ad_agr_list_def sp_equiv_list_link fv_fo_terms_set_list
        fv_fo_terms_set_def sp_equiv_def pairwise_def)
  show ?case
  proof (cases t)
    case (Const c)
    show ?thesis
      using IH Cons(2)
      apply (auto simp: ad_agr_list_def eval_eterms_def ad_equiv_list_def Const
          sp_equiv_list_def pairwise_def set_zip)
      unfolding ad_equiv_pair.simps
        apply (metis nth_map rev_image_eqI)+
      done
  next
    case (Var n)
    note t_def = Var
    have ad: ad_equiv_pair X (σ n, τ n)
      using ad_equiv
      by (auto simp: Var)
    have ⋀y. y ∈ set (zip (map ((·e) σ) ts) (map ((·e) τ) ts)) ⟹ y ≠ (σ n, τ n) ⟹
      sp_equiv_pair (σ n, τ n) y ∧ sp_equiv_pair y (σ n, τ n)
    proof −
      fix y
      assume y ∈ set (zip (map ((·e) σ) ts) (map ((·e) τ) ts))
      then obtain t' where y_def: t' ∈ set ts y = (σ ·e t', τ ·e t')
        using nth_mem
        by (auto simp: set_zip) blast
      show sp_equiv_pair (σ n, τ n) y ∧ sp_equiv_pair y (σ n, τ n)
      proof (cases t')
```

```
      case (Const c′)
      have c′_X: c′ ∈ X
        using Cons(2) y_def(1)
        by (auto simp: Const) (meson SUP_le_iff fo_term.set_intros subsetD)
      then show ?thesis
        using ad_equiv[of n] y_def(1)
        unfolding y_def
        apply (auto simp: Const t_def)
        unfolding ad_equiv_pair.simps
           apply fastforce+
         apply force
        apply (metis rev_image_eqI)
        done
    next
      case (Var n′)
      show ?thesis
        using sp_equiv[of n n′] y_def(1)
        unfolding y_def
        by (fastforce simp: t_def Var)
    qed
  qed
  then show ?thesis
    using IH Cons(3)
    by (auto simp: ad_agr_list_def eval_eterms_def ad_equiv_list_def Var ad sp_equiv_list_def
        pairwise_insert)
 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 Sφ_def: Sφ ≡ {σ. esat φ I σ UNIV}
    and AD_sub: act_edom φ I ⊆ ADφ ADφ ⊆ AD
    and Xφ_def: Xφ = fo_nmlz ADφ ‘ proj_vals Sφ (fv_fo_fmla_list φ)
    and nsφ′_def: nsφ′ = filter (λn. n ∉ fv_fo_fmla φ) nsψ
    and sd_nsψ: sorted_distinct nsψ
    and fv_Un: fv_fo_fmla ψ = fv_fo_fmla φ ∪ set nsψ
  shows ext_tuple_set AD (fv_fo_fmla_list φ) nsφ′ (ad_agr_close_set (AD − ADφ) Xφ) =
    fo_nmlz AD ‘ proj_vals Sφ (fv_fo_fmla_list ψ)
    ad_agr_close_set (AD − ADφ) Xφ = fo_nmlz AD ‘ proj_vals Sφ (fv_fo_fmla_list φ)
proof −
  have ad_agr_φ:
    ⋀σ τ. ad_agr_sets (set (fv_fo_fmla_list φ)) (set (fv_fo_fmla_list φ)) ADφ σ τ ⟹
      σ ∈ Sφ ⟷ τ ∈ Sφ
    using esat_UNIV_cong[OF ad_agr_sets_restrict, OF _ subset_refl] ad_agr_sets_mono AD_sub
    unfolding Sφ_def
    by blast
  show ad_close_alt: ad_agr_close_set (AD − ADφ) Xφ = fo_nmlz AD ‘ proj_vals Sφ (fv_fo_fmla_list
φ)
    using ad_agr_close_correct[OF AD_sub(2) ad_agr_φ] AD_sub(2)
    unfolding Xφ_def Sφ_def[symmetric] proj_fmla_def
    by (auto simp: ad_agr_close_set_def Set.is_empty_def)
  have fv_φ: set (fv_fo_fmla_list φ) ⊆ set (fv_fo_fmla_list ψ)
    using fv_Un
    by (auto simp: fv_fo_fmla_list_set)
  have sd_nsφ′: sorted_distinct nsφ′
    using sd_nsψ sorted_filter[of id]
    by (auto simp: nsφ′_def)
  show ext_tuple_set AD (fv_fo_fmla_list φ) nsφ′ (ad_agr_close_set (AD − ADφ) Xφ) =
    fo_nmlz AD ‘ proj_vals Sφ (fv_fo_fmla_list ψ)
```

apply (*rule ext_tuple_correct*)
using *sorted_distinct_fv_list ad_close_alt ad_agr_φ ad_agr_sets_mono*[*OF AD_sub*(*2*)]
fv_Un sd_nsφ′
by (*fastforce simp*: *nsφ′_def fv_fo_fmla_list_set*)+
qed

lemma *proj_ext_tuple*:
assumes *Sφ_def*: *Sφ* ≡ {*σ*. *esat φ I σ UNIV*}
and *AD_sub*: *act_edom φ I* ⊆ *AD*
and *Xφ_def*: *Xφ* = *fo_nmlz AD* ‘ *proj_vals Sφ* (*fv_fo_fmla_list φ*)
and *nsφ′_def*: *nsφ′* = *filter* (*λn*. *n* ∉ *fv_fo_fmla φ*) *nsψ*
and *sd_nsψ*: *sorted_distinct nsψ*
and *fv_Un*: *fv_fo_fmla ψ* = *fv_fo_fmla φ* ∪ *set nsψ*
and *Z_props*: ⋀*xs*. *xs* ∈ *Z* ⟹ *fo_nmlz AD xs* = *xs* ∧ *length xs* = *length* (*fv_fo_fmla_list ψ*)
shows *Z* ∩ *ext_tuple_set AD* (*fv_fo_fmla_list φ*) *nsφ′ Xφ* =
{*xs* ∈ *Z*. *fo_nmlz AD* (*proj_tuple* (*fv_fo_fmla_list φ*) (*zip* (*fv_fo_fmla_list ψ*) *xs*)) ∈ *Xφ*}
Z − *ext_tuple_set AD* (*fv_fo_fmla_list φ*) *nsφ′ Xφ* =
{*xs* ∈ *Z*. *fo_nmlz AD* (*proj_tuple* (*fv_fo_fmla_list φ*) (*zip* (*fv_fo_fmla_list ψ*) *xs*)) ∉ *Xφ*}
proof −
have *ad_agr_φ*:
⋀*σ τ*. *ad_agr_sets* (*set* (*fv_fo_fmla_list φ*)) (*set* (*fv_fo_fmla_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*
unfolding *Sφ_def*
by *blast*
have *sd_nsφ′*: *sorted_distinct nsφ′*
using *sd_nsψ sorted_filter*[*of id*]
by (*auto simp*: *nsφ′_def*)
have *disj*: *set* (*fv_fo_fmla_list φ*) ∩ *set nsφ′* = {}
by (*auto simp*: *nsφ′_def fv_fo_fmla_list_set*)
have *Un*: *set* (*fv_fo_fmla_list φ*) ∪ *set nsφ′* = *set* (*fv_fo_fmla_list ψ*)
using *fv_Un*
by (*auto simp*: *nsφ′_def fv_fo_fmla_list_set*)
note *proj* = *proj_tuple_correct*[*OF sorted_distinct_fv_list sd_nsφ′ sorted_distinct_fv_list*
disj Un Xφ_def ad_agr_φ, *simplified*]
have *fo_nmlz AD* ‘ *Xφ* = *Xφ*
using *fo_nmlz_idem*[*OF fo_nmlz_sound*]
by (*auto simp*: *Xφ_def image_iff*)
then have *aux*: *ext_tuple_set AD* (*fv_fo_fmla_list φ*) *nsφ′ Xφ* = *fo_nmlz AD* ‘ ⋃(*ext_tuple AD*
(*fv_fo_fmla_list φ*) *nsφ′* ‘ *Xφ*)
by (*auto simp*: *ext_tuple_set_def ext_tuple_def*)
show *Z* ∩ *ext_tuple_set AD* (*fv_fo_fmla_list φ*) *nsφ′ Xφ* =
{*xs* ∈ *Z*. *fo_nmlz AD* (*proj_tuple* (*fv_fo_fmla_list φ*) (*zip* (*fv_fo_fmla_list ψ*) *xs*)) ∈ *Xφ*}
using *Z_props proj*
by (*auto simp*: *aux*)
show *Z* − *ext_tuple_set AD* (*fv_fo_fmla_list φ*) *nsφ′ Xφ* =
{*xs* ∈ *Z*. *fo_nmlz AD* (*proj_tuple* (*fv_fo_fmla_list φ*) (*zip* (*fv_fo_fmla_list ψ*) *xs*)) ∉ *Xφ*}
using *Z_props proj*
by (*auto simp*: *aux*)
qed

lemma *fo_nmlz_proj_sub*: *fo_nmlz AD* ‘ *proj_fmla φ R* ⊆ *nall_tuples AD* (*nfv φ*)
by (*auto simp*: *proj_fmla_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* ⋀*vs*. *vs* ∈ *Z* ⟹ *length vs* = *n*

$Z = \{ts.\ \exists\, ts' \in X.\ ad\_agr\_list\ AD\ (map\ Inl\ ts)\ ts'\}$
shows $finite\ Z \longleftrightarrow \bigcup(set\ `\ Z) \subseteq AD$
proof $(rule\ iffI,\ rule\ ccontr)$
assume fin: $finite\ Z$
assume $\neg\bigcup(set\ `\ Z) \subseteq AD$
then obtain $\sigma\ i\ vs$ **where** σ_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 $\bigwedge y.\ y \notin Y \Longrightarrow map\ ((\lambda z.\ if\ z = \sigma\ i\ then\ y\ else\ z) \circ \sigma)\ [0..<n] \in Z$
using $\sigma\_def(3)$
by $(auto\ simp: assms(3)\ intro!: bexI[OF\ \_\ \sigma\_def(4)]\ ad\_agr\_list\_trans[OF\ \_\ \sigma\_def(5)])$
$(auto\ simp: ad\_agr\_list\_def\ ad\_equiv\_list\_def\ set\_zip\ Y\_def\ ad\_equiv\_pair.simps$
$sp\_equiv\_list\_def\ pairwise\_def\ split: if\_splits)$
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 τ **where** $\tau = case\_sum\ id\ f \circ \sigma$
have f_out: $\bigwedge i\ x.\ i < length\ ns \Longrightarrow \sigma\ (ns\ !\ i) = Inl\ (f\ x) \Longrightarrow False$
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 $ad\_agr\_list\ AD\ (map\ \sigma\ ns)\ (map\ (Inl \circ \tau)\ ns)$

apply (*auto simp*: *ad_agr_list_def ad_equiv_list_def*)
subgoal for *a b*
using *f_def*(*2*)
by (*auto simp*: *set_zip* τ*_def ad_equiv_pair.simps split*: *sum.splits*)+
using *f_def*(*1*) *f_out*
apply (*auto simp*: *sp_equiv_list_def pairwise_def set_zip* τ*_def inj_def split*: *sum.splits*)+
done
then show *?thesis*
by (*auto simp*: τ*_def intro*!: *exI*[*of _* τ])
qed

lemma *proj_out*:
fixes φ :: (*'a* :: *infinite*, *'b*) *fo_fmla*
and *J* :: ((*'a*, *nat*) *fo_t*, *'b*) *fo_intp*
assumes *wf_fo_intp* φ *I esat* φ *I* σ *UNIV*
shows $\exists\tau$. *esat* φ *I* (*Inl* $\circ$ τ) *UNIV* $\wedge$ ($\forall$ *i x*. *i* $\in$ *fv_fo_fmla* φ $\wedge$ σ *i* = *Inl x* $\longrightarrow$ τ *i* = *x*) $\wedge$
ad_agr_list (*act_edom* φ *I*) (*map* σ (*fv_fo_fmla_list* φ)) (*map* (*Inl* $\circ$ τ) (*fv_fo_fmla_list* φ))
using *proj_out_list*[*OF finite_act_edom*[*OF assms*(*1*)], *of* σ *fv_fo_fmla_list* φ]
esat_UNIV_ad_agr_list[*OF _ subset_refl*] *assms*(*2*)
unfolding *fv_fo_fmla_list_set*
by *fastforce*

lemma *proj_fmla_esat_sat*:
fixes φ :: (*'a* :: *infinite*, *'b*) *fo_fmla*
and *J* :: ((*'a*, *nat*) *fo_t*, *'b*) *fo_intp*
assumes *wf*: *wf_fo_intp* φ *I*
shows *proj_fmla* φ {σ. *esat* φ *I* σ *UNIV*} $\cap$ *map Inl* ‘ *UNIV* =
map Inl ‘ *proj_fmla* φ {σ. *sat* φ *I* σ}
unfolding *sat_esat_conv*[*OF wf*]
proof (*rule set_eqI*, *rule iffI*)
fix *vs*
assume *vs* $\in$ *proj_fmla* φ {σ. *esat* φ *I* σ *UNIV*} $\cap$ *map Inl* ‘ *UNIV*
then obtain σ **where** σ*_def*: *vs* = *map* σ (*fv_fo_fmla_list* φ) *esat* φ *I* σ *UNIV*
set vs $\subseteq$ *range Inl*
by (*auto simp*: *proj_fmla_map*) (*metis image_subset_iff list.set_map range_eqI*)
obtain τ **where** τ*_def*: *esat* φ *I* (*Inl* $\circ$ τ) *UNIV*
$\bigwedge$*i x*. *i* $\in$ *fv_fo_fmla* φ $\Longrightarrow$ σ *i* = *Inl x* $\Longrightarrow$ τ *i* = *x*
using *proj_out*[*OF assms* σ*_def*(*2*)]
by *fastforce*
have *vs* = *map* (*Inl* $\circ$ τ) (*fv_fo_fmla_list* φ)
using σ*_def*(*1*,*3*) τ*_def*(*2*)
by (*auto simp*: *fv_fo_fmla_list_set*)
then show *vs* $\in$ *map Inl* ‘ *proj_fmla* φ {σ. *esat* φ *I* (*Inl* $\circ$ σ) *UNIV*}
using τ*_def*(*1*)
by (*force simp*: *proj_fmla_map*)
qed (*auto simp*: *proj_fmla_map*)

lemma *norm_proj_fmla_esat_sat*:
fixes φ :: (*'a* :: *infinite*, *'b*) *fo_fmla*
assumes *wf_fo_intp* φ *I*
shows *fo_nmlz* (*act_edom* φ *I*) ‘ *proj_fmla* φ {σ. *esat* φ *I* σ *UNIV*} =
fo_nmlz (*act_edom* φ *I*) ‘ *map Inl* ‘ *proj_fmla* φ {σ. *sat* φ *I* σ}
unfolding *proj_fmla_esat_sat*[*OF assms*, *symmetric*]
apply (*auto simp*: *image_iff proj_fmla_map*)
subgoal for σ
using *proj_out*[*OF assms*, *of* σ]
apply *auto*
subgoal for τ

```
    by (auto intro!: bexI[of _ map (Inl ∘ τ) (fv_fo_fmla_list φ)] fo_nmlz_eqI)
      (metis map_map range_eqI)
  done
done

lemma proj_sat_fmla: proj_sat φ I = proj_fmla φ {σ. sat φ I σ}
  by (auto simp: proj_sat_def proj_fmla_map)

fun fo_wf :: ('a, 'b) fo_fmla ⇒ ('b × nat ⇒ 'a list set) ⇒ ('a, nat) fo_t ⇒ bool where
  fo_wf φ I (AD, n, X) ⟷ finite AD ∧ finite X ∧ n = nfv φ ∧
    wf_fo_intp φ I ∧ AD = act_edom φ I ∧ fo_rep (AD, n, X) = proj_sat φ I ∧
    Inl −` ⋃(set ` X) ⊆ AD ∧ (∀ vs ∈ X. fo_nmlzd AD vs ∧ length vs = n)

fun fo_fin :: ('a, nat) fo_t ⇒ bool where
  fo_fin (AD, n, X) ⟷ (∀ x ∈ ⋃(set ` X). isl x)

lemma fo_rep_fin:
  assumes fo_wf φ 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 ∈ 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_fmla ⇒ ('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_fmla 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_fmla
  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_fmla 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_fmla φ {σ. sat φ I σ}
    using AD_X_def norm_proj_fmla_esat_sat[OF assms]
    by auto
  have {ts. ∃ts' ∈ X. ad_agr_list AD (map Inl ts) ts'} = proj_fmla 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_fmla 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_fmla_list φ)
      using vs'_def
      by (auto simp: proj_fmla_map ad_agr_list_def fo_nmlz_length)
    then obtain σ where σ_def: vs = map σ (fv_fo_fmla_list φ)
      using exists_map[of fv_fo_fmla_list φ vs] sorted_distinct_fv_list
      by fastforce
    obtain τ where τ_def: fo_nmlz AD (map Inl vs') = map τ (fv_fo_fmla_list φ)
      using vs'_def fo_nmlz_map
      by (fastforce simp: proj_fmla_map)
    have ad_agr: ad_agr_list AD (map (Inl ∘ σ) (fv_fo_fmla_list φ)) (map τ (fv_fo_fmla_list φ))
      by (metis σ_def τ_def map_map vs'_def(2))
    obtain τ' where τ'_def: map Inl vs' = map (Inl ∘ τ') (fv_fo_fmla_list φ)
      sat φ I τ'
      using vs'_def(1)
      by (fastforce simp: proj_fmla_map)
    have ad_agr': ad_agr_list AD (map τ (fv_fo_fmla_list φ))
        (map (Inl ∘ τ') (fv_fo_fmla_list φ))
      by (rule ad_agr_list_comm) (metis fo_nmlz_ad_agr τ'_def(1) τ_def map_map map_projl_Inl)
    have esat: esat φ I τ UNIV
      using esat_UNIV_ad_agr_list[OF ad_agr' AD_sub, folded sat_esat_conv[OF assms]] τ'_def(2)
      by auto
    show vs ∈ proj_fmla φ {σ. sat φ I σ}
      using esat_UNIV_ad_agr_list[OF ad_agr AD_sub, folded sat_esat_conv[OF assms]] esat
      unfolding σ_def
      by (auto simp: proj_fmla_map)
  next
    fix vs
    assume vs ∈ proj_fmla φ {σ. sat φ I σ}
    then have vs_X: fo_nmlz AD (map Inl vs) ∈ X
      using X_def
      by auto
    then show vs ∈ {ts. ∃ts' ∈ X. ad_agr_list AD (map Inl ts) ts'}
      using fo_nmlz_ad_agr
      by auto
  qed
  then show ?thesis
    by (auto simp: AD_X_def proj_sat_fmla)
qed
```

lemma *fo_wf_eval_abs*:
  fixes φ :: ('*a* :: *infinite*, '*b*) *fo_fmla*
  assumes *wf_fo_intp* φ *I*
  shows *fo_wf* φ *I* (*eval_abs* φ *I*)
  using *fo_nmlz_set*[*of act_edom* φ *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_fmla_map*)
blast

lemma *fo_fin*:
  fixes *t* :: ('*a* :: *infinite*, *nat*) *fo_t*
  assumes *fo_wf* φ *I t*
  shows *fo_fin t* = *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* $\Longrightarrow$ *length vs* = *n*
    using *assms*
    by (*auto simp*: *t_def*)
  have *Inl_X_AD*: $\bigwedge$*x*. *Inl x* $\in \bigcup$(*set* ‘ *X*) $\Longrightarrow$ *x* $\in$ *AD*
    using *assms*
    by (*fastforce simp*: *t_def*)
  define *Z* **where** *Z* = {*ts*. $\exists$ *ts'* $\in$ *X*. *ad_agr_list AD* (*map Inl ts*) *ts'*}
  have *fin_Z_iff*: *finite Z* = ($\bigcup$(*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 ($\bigcup$(*set* ‘ *Z*) $\subseteq$ *AD*) $\longleftrightarrow$ ($\forall$ *x* $\in \bigcup$(*set* ‘ *X*). *isl x*)
  proof (*rule iffI*, *rule ccontr*)
    fix *x*
    assume *Z_sub_AD*: $\bigcup$(*set* ‘ *Z*) $\subseteq$ *AD*
    assume $\neg$($\forall$ *x* $\in \bigcup$(*set* ‘ *X*). *isl x*)
    then obtain *vs i m* **where** *vs_def*: *vs* $\in$ *X i* < *n vs* ! *i* = *Inr m*
      using *len_in_X*
      by (*auto simp*: *in_set_conv_nth*) (*metis sum.collapse*(*2*))
    obtain σ **where** σ*_def*: *vs* = *map* σ [*0*..<*n*]
      using *exists_map*[*of* [*0*..<*n*] *vs*] *len_in_X*[*OF vs_def*(*1*)]
      by *auto*
    obtain τ **where** τ*_def*: *ad_agr_list AD vs* (*map Inl* (*map* τ [*0*..<*n*]))
      using *proj_out_list*[*OF fin*(*1*), *of* σ [*0*..<*n*]]
      by (*auto simp*: σ*_def*)
    have *map_*τ*_in_Z*: *map* τ [*0*..<*n*] $\in$ *Z*
      using *vs_def*(*1*) *ad_agr_list_comm*[*OF* τ*_def*]
      by (*auto simp*: *Z_def*)
    moreover have τ *i* $\notin$ *AD*
      using τ*_def vs_def*(*2*,*3*)
      apply (*auto simp*: *ad_agr_list_def ad_equiv_list_def set_zip comp_def* σ*_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 $\forall x \in \bigcup(set\ `\ X).\ isl\ x$
  then show $\bigcup(set\ `\ Z) \subseteq AD$
    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 φ **where** $\varphi = Pred\ r\ ts$
  have *nfv_len*: $nfv\ \varphi = length\ (fv\_fo\_terms\_list\ ts)$
    by (*auto simp*: *φ_def nfv_def fv_fo_fmla_list_def fv_fo_fmla_list_Pred*)
  have *vimage_unfold*: $Inl\ -`\ (\bigcup x{\in}I\ (r,\ length\ ts).\ Inl\ `\ set\ x) = \bigcup(set\ `\ I\ (r,\ length\ ts))$
    by *auto*
  have *eval_table ts* (*map Inl* ‘ *I* (*r*, *length ts*)) $\subseteq$ *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_fmla_def eval_pred_def eval_table fv_fo_fmla_list_def*
        fv_fo_fmla_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 *ad_agr_list_eval*: $\bigcup(set\ (map\ set\_fo\_term\ ts)) \subseteq AD \Longrightarrow ad\_agr\_list\ AD\ (\sigma \odot e\ ts)\ zs \Longrightarrow$
  $\exists \tau.\ zs = \tau \odot e\ ts$
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*: *ad_agr_list_def eval_eterms_def*)
  obtain τ **where** *τ_def*: $ws = \tau \odot e\ ts$
    using *Cons*
    by (*fastforce simp*: *zs_split ad_agr_list_def ad_equiv_list_def sp_equiv_list_def 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 ad_agr_list_def ad_equiv_list_def*)
      unfolding *ad_equiv_pair.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 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 sp_equiv_pair _ (σ n, w) (σ ·e (ts ! i), τ ·e (ts ! i))]
        by (force simp: Var eval_eterms_def ad_agr_list_def sp_equiv_list_def 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: ad_agr_list_def eval_eterms_def)

lemma sp_equiv_list_fv_list:
  assumes sp_equiv_list (σ ⊙e ts) (τ ⊙e ts)
  shows sp_equiv_list (map σ (fv_fo_terms_list ts)) (map τ (fv_fo_terms_list ts))
proof −
  have sp_equiv_list (σ ⊙e (map Var (fv_fo_terms_list ts)))
    (τ ⊙e (map Var (fv_fo_terms_list ts)))
    unfolding eval_eterms_def
    by (rule sp_equiv_list_subset[OF _ assms[unfolded eval_eterms_def]])
      (auto simp: fv_fo_terms_set_list dest: fv_fo_terms_setD)
  then show ?thesis
    by (auto simp: eval_eterms_def comp_def)
qed

lemma ad_agr_list_fv_list: ad_agr_list X (σ ⊙e ts) (τ ⊙e ts) ⟹
  ad_agr_list X (map σ (fv_fo_terms_list ts)) (map τ (fv_fo_terms_list ts))
  using sp_equiv_list_fv_list
  by (auto simp: eval_eterms_def ad_agr_list_def ad_equiv_list_def sp_equiv_list_def set_zip)
    (metis (no_types, opaque_lifting) eval_eterm.simps(2) fv_fo_terms_setD fv_fo_terms_set_list
     in_set_conv_nth nth_map)

lemma eval_bool: fo_wf (Bool b) I (eval_bool b)
  by (auto simp: eval_bool_def fo_nmlzd_def nats_def Let_def List.map_filter_simps
      proj_sat_def fv_fo_fmla_list_def ad_agr_list_def ad_equiv_list_def sp_equiv_list_def nfv_def)

lemma eval_eq: fixes I :: 'b × nat ⇒ 'a :: infinite list set
  shows fo_wf (Eqa t t') I (eval_eq t t')
proof −
  define φ :: ('a, 'b) fo_fmla where φ = Eqa t t'
  obtain AD n X where AD_X_def: eval_eq t t' = (AD, n, X)
    by (cases eval_eq t t') auto
  have AD_def: AD = act_edom φ I
    using AD_X_def
    by (auto simp: eval_eq_def φ_def split: fo_term.splits if_splits)
```

have n_def: $n = nfv\ \varphi$
  using AD_X_def
  by $(cases\ t;\ cases\ t')$
    $(auto\ simp: \varphi\_def\ fv\_fo\_fmla\_list\_def\ eval\_eq\_def\ nfv\_def\ split: if\_splits)$
have X_def: $X = fo\_nmlz\ AD\ `\ proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
proof $(rule\ set\_eqI,\ rule\ iffI)$
  fix vs
  assume $assm$: $vs \in X$
  define pes **where** $pes = proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
  have $\bigwedge c\ c'.\ t = Const\ c \wedge t' = Const\ c' \Longrightarrow$
    $fo\_nmlz\ AD\ `\ pes = (if\ c = c'\ then\ \{[]\}\ else\ \{\})$
    by $(auto\ simp: \varphi\_def\ pes\_def\ proj\_fmla\_map\ fo\_nmlz\_def\ fv\_fo\_fmla\_list\_def)$
  moreover have $\bigwedge c\ n.\ (t = Const\ c \wedge t' = Var\ n) \vee (t' = Const\ c \wedge t = Var\ n) \Longrightarrow$
    $fo\_nmlz\ AD\ `\ pes = \{[Inl\ c]\}$
    by $(auto\ simp: \varphi\_def\ AD\_def\ pes\_def\ proj\_fmla\_map\ fo\_nmlz\_Cons\ fv\_fo\_fmla\_list\_def\ image\_def$
      $split: sum.splits)\ (auto\ simp: fo\_nmlz\_def)$
  moreover have $\bigwedge n.\ t = Var\ n \Longrightarrow t' = Var\ n \Longrightarrow fo\_nmlz\ AD\ `\ pes = \{[Inr\ 0]\}$
    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 $\bigwedge n\ n'.\ t = Var\ n \Longrightarrow t' = Var\ n' \Longrightarrow n \neq n' \Longrightarrow$
    $fo\_nmlz\ AD\ `\ pes = \{[Inr\ 0,\ Inr\ 0]\}$
    apply $(auto\ simp: \varphi\_def\ AD\_def\ pes\_def\ proj\_fmla\_map\ fo\_nmlz\_Cons\ fv\_fo\_fmla\_list\_def$
      $split: sum.splits)$
    subgoal for $i\ i'\ \sigma$
      by $(cases\ \sigma\ i')\ (auto\ simp: fo\_nmlz\_def\ split: if\_splits)$
    subgoal for $i\ i'$
      by $(auto\ simp: image\_def\ fo\_nmlz\_def\ intro!: exI[of\ \_\ [Inr\ 0,\ Inr\ 0]])$
    done
  ultimately show $vs \in fo\_nmlz\ AD\ `\ pes$
    using $assm\ AD\_X\_def$
    by $(cases\ t;\ cases\ t')\ (auto\ simp: eval\_eq\_def\ split: if\_splits)$
next
  fix vs
  assume $assm$: $vs \in fo\_nmlz\ AD\ `\ proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
  obtain σ **where** σ_def: $vs = fo\_nmlz\ AD\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \varphi))$
    $esat\ (Eqa\ t\ t')\ I\ \sigma\ UNIV$
    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 $fv\_fo\_terms\_list\_Var$: $fv\_fo\_terms\_list\_rec\ (map\ Var\ ns) = ns$
  by $(induction\ ns)\ auto$

```
lemma eval_eterms_map_Var: σ ⊙e map Var ns = map σ ns
  by (auto simp: eval_eterms_def)

lemma fo_wf_eval_table:
  fixes AD :: 'a set
  assumes fo_wf φ I (AD, n, X)
  shows X = fo_nmlz AD ` eval_table (map Var [0..<n]) X
proof -
  have AD_sup: Inl -` ⋃(set ` X) ⊆ AD
    using assms
    by fastforce
  have fvs: fv_fo_terms_list (map Var [0..<n]) = [0..<n]
    by (auto simp: fv_fo_terms_list_def fv_fo_terms_list_Var remdups_adj_distinct)
  have ⋀vs. vs ∈ X ⟹ length vs = n
    using assms
    by auto
  then have X_map: ⋀vs. vs ∈ X ⟹ ∃σ. vs = map σ [0..<n]
    using exists_map[of [0..<n]]
    by auto
  then have proj_vals_X: proj_vals {σ. σ ⊙e map Var [0..<n] ∈ X} [0..<n] = X
    by (auto simp: eval_eterms_map_Var proj_vals_def)
  then show X = fo_nmlz AD ` 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 φ I (AD, n, X)
  shows X = fo_nmlz AD ` map Inl ` fo_rep (AD, n, X)
proof (rule set_eqI, rule iffI)
  fix vs
  assume vs_in: vs ∈ 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 τ where τ_def: ad_agr_list AD vs (map Inl (map τ [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 τ [0..<n] ∈ 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 ∈ fo_nmlz AD ` map Inl ` 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' where vs_def: xs' ∈ 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 $X = fo\_nmlz\ AD\ `\ proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
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: $fo\_rep\ (AD, n, X) = proj\_fmla\ \varphi\ \{\sigma.\ sat\ \varphi\ I\ \sigma\}$
using wf
by $(auto\ simp: proj\_sat\_def\ proj\_fmla\_map)$
show *?thesis*
using $fo\_rep\_norm[OF\ fo\_wf]\ norm\_proj\_fmla\_esat\_sat[OF\ fin]$
unfolding $fo\_rep\ AD\_def[symmetric]$
by *auto*
qed

lemma $eval\_neg$:
fixes $\varphi :: ('a :: infinite, 'b)\ fo\_fmla$
assumes wf: $fo\_wf\ \varphi\ I\ t$
shows $fo\_wf\ (Neg\ \varphi)\ I\ (eval\_neg\ (fv\_fo\_fmla\_list\ \varphi)\ t)$
proof −
obtain $AD\ n\ X$ **where** t_def: $t = (AD, n, X)$
by $(cases\ t)\ auto$
have $eval\_neg$: $eval\_neg\ (fv\_fo\_fmla\_list\ \varphi)\ t = (AD, nfv\ \varphi, nall\_tuples\ AD\ (nfv\ \varphi) - X)$
by $(auto\ simp: t\_def\ nfv\_def)$
have fv_unfold: $fv\_fo\_fmla\_list\ (Neg\ \varphi) = fv\_fo\_fmla\_list\ \varphi$
by $(auto\ simp: fv\_fo\_fmla\_list\_def)$
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 $esat\_iff$: $\bigwedge vs.\ vs \in nall\_tuples\ AD\ (nfv\ \varphi) \Longrightarrow$
$vs \in fo\_nmlz\ AD\ `\ proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\} \longleftrightarrow$
$vs \notin fo\_nmlz\ AD\ `\ proj\_fmla\ \varphi\ \{\sigma.\ esat\ (Neg\ \varphi)\ I\ \sigma\ UNIV\}$
proof $(rule\ iffI; rule\ ccontr)$
fix vs
assume $vs \in fo\_nmlz\ AD\ `\ proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$

then obtain σ **where** σ*_def*: $vs = fo\_nmlz\ AD\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \varphi))$
$esat\ \varphi\ I\ \sigma\ UNIV$
by (*auto simp*: *proj_fmla_map*)
assume $\neg vs \notin fo\_nmlz\ AD$ ‘ $proj\_fmla\ \varphi\ \{\sigma.\ esat\ (Neg\ \varphi)\ I\ \sigma\ UNIV\}$
then obtain σ' **where** σ'*_def*: $vs = fo\_nmlz\ AD\ (map\ \sigma'\ (fv\_fo\_fmla\_list\ \varphi))$
$esat\ (Neg\ \varphi)\ I\ \sigma'\ UNIV$
by (*auto simp*: *proj_fmla_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* σ*_def*(*1*)[*symmetric*] σ'*_def*(*1*)]]]]
by (*auto simp*: *AD_def*)
then show *False*
using σ*_def*(*2*) σ'*_def*(*2*) **by** *simp*
next
fix *vs*
assume *assms*: $vs \notin fo\_nmlz\ AD$ ‘ $proj\_fmla\ \varphi\ \{\sigma.\ esat\ (Neg\ \varphi)\ I\ \sigma\ UNIV\}$
$vs \notin fo\_nmlz\ AD$ ‘ $proj\_fmla\ \varphi\ \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
assume $vs \in nall\_tuples\ AD\ (nfv\ \varphi)$
then have *l_vs*: $length\ vs = length\ (fv\_fo\_fmla\_list\ \varphi)\ fo\_nmlzd\ AD\ vs$
by (*auto simp*: *nfv_def dest*: *nall_tuplesD*)
obtain σ **where** $vs = fo\_nmlz\ AD\ (map\ \sigma\ (fv\_fo\_fmla\_list\ \varphi))$
using *l_vs sorted_distinct_fv_list exists_fo_nmlzd* **by** *metis*
with *assms* **show** *False*
by (*auto simp*: *proj_fmla_map*)
qed
moreover have $\bigwedge R.\ fo\_nmlz\ AD$ ‘ $proj\_fmla\ \varphi\ R \subseteq nall\_tuples\ AD\ (nfv\ \varphi)$
by (*auto simp*: *proj_fmla_map nfv_def nall_tuplesI fo_nmlz_length fo_nmlz_sound*)
ultimately have *eval*: $eval\_neg\ (fv\_fo\_fmla\_list\ \varphi)\ t = eval\_abs\ (Neg\ \varphi)\ I$
unfolding *eval_neg eval_abs_def AD_def*[*symmetric*]
by (*auto simp*: *X_def proj_fmla_def fv_unfold nfv_unfold image_subset_iff*)
have *wf_neg*: $wf\_fo\_intp\ (Neg\ \varphi)\ I$
using *wf*
by (*auto simp*: *t_def*)
show *?thesis*
using *fo_wf_eval_abs*[*OF wf_neg*]
by (*auto simp*: *eval*)
qed

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

lemma *mapping_join_cross_with*:
assumes $\bigwedge x\ x'.\ x \in t \Longrightarrow x' \in t' \Longrightarrow h\ x \neq h'\ x' \Longrightarrow f\ x\ x' = \{\}$
shows $set\_of\_idx\ (mapping\_join\ (cross\_with\ f)\ (cluster\ (Some \circ h)\ t)\ (cluster\ (Some \circ h')\ t')) = cross\_with\ f\ t\ t'$
proof −
have *sub*: $cross\_with\ f\ \{y \in t.\ h\ y = h\ x\}\ \{y \in t'.\ h'\ y = h\ x\} \subseteq cross\_with\ f\ t\ t'$ **for** $t\ t'\ x$
by (*auto simp*: *cross_with_def*)
have $\exists a.\ a \in h$ ‘ $t \wedge a \in h'$ ‘ $t' \wedge z \in cross\_with\ f\ \{y \in t.\ h\ y = a\}\ \{y \in t'.\ h'\ y = a\}$ **if** z: $z \in cross\_with\ f\ t\ t'$ **for** z
proof −
obtain *xs ys* **where** *wit*: $xs \in t\ ys \in t'\ z \in f\ xs\ ys$
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*: $h'\ ys \in h$ ‘ 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*: $X \subseteq X' \Longrightarrow$ *fo_nmlzd X xs* $\Longrightarrow$ *fo_nmlzd X' xs*
by (*meson fo_nmlzd_def order_trans*)

lemma *idx_join*:
assumes *Xφ_props*: $\bigwedge$*vs. vs* $\in$ *Xφ* $\Longrightarrow$ *fo_nmlzd AD vs* $\wedge$ *length vs* = *length nsφ*
assumes *Xψ_props*: $\bigwedge$*vs. vs* $\in$ *Xψ* $\Longrightarrow$ *fo_nmlzd AD vs* $\wedge$ *length vs* = *length nsψ*
assumes *sd_ns*: *sorted_distinct nsφ sorted_distinct nsψ*
assumes *ns_def*: *ns* = *filter* (*λn. n* $\in$ *set nsψ*) *nsφ*
shows *idx_join AD ns nsφ Xφ nsψ Xψ* = *eval_conj_set AD nsφ Xφ nsψ Xψ*
proof −
have *ect_empty*: *x* $\in$ *Xφ* $\Longrightarrow$ *x'* $\in$ *Xψ* $\Longrightarrow$ *fo_nmlz AD* (*proj_tuple ns* (*zip nsφ x*)) $\neq$ *fo_nmlz AD* (*proj_tuple ns* (*zip nsψ x'*)) $\Longrightarrow$
eval_conj_tuple AD nsφ nsψ x x' = {}
if *Xφ'* $\subseteq$ *Xφ Xψ'* $\subseteq$ *Xψ* **for** *Xφ' Xψ'* **and** *x x'*
apply (*rule eval_conj_tuple_empty*[**where** *?ns=filter* (*λn. n* $\in$ *set nsψ*) *nsφ*])
using *Xφ_props Xψ_props that sd_ns*
by (*auto simp*: *ns_def ad_agr_close_set_def split*: *if_splits*)
have *cross_eval_conj_tuple*: (*λXφ''. eval_conj_set AD nsφ Xφ'' nsψ*) = *cross_with* (*eval_conj_tuple AD nsφ nsψ*) **for** *AD* :: *'a set* **and** *nsφ nsψ*
by (*rule ext*)+ (*auto simp*: *eval_conj_set_def cross_with_def*)
have *idx_join AD ns nsφ Xφ nsψ Xψ* = *cross_with* (*eval_conj_tuple AD nsφ nsψ*) *Xφ Xψ*
unfolding *idx_join_def Let_def cross_eval_conj_tuple*
by (*rule mapping_join_cross_with*[*OF ect_empty*]) *auto*
moreover have ... = *eval_conj_set AD nsφ Xφ nsψ Xψ*
by (*auto simp*: *cross_with_def eval_conj_set_def*)
finally show *?thesis* **.**
qed

lemma *proj_fmla_conj_sub*:
assumes *AD_sub*: *act_edom ψ I* $\subseteq$ *AD*
shows *fo_nmlz AD* ` *proj_fmla* (*Conj φ ψ*) {*σ. esat φ I σ UNIV*} $\cap$
fo_nmlz AD ` *proj_fmla* (*Conj φ ψ*) {*σ. esat ψ I σ UNIV*} $\subseteq$
fo_nmlz AD ` *proj_fmla* (*Conj φ ψ*) {*σ. esat* (*Conj φ ψ*) *I σ UNIV*}
proof (*rule subsetI*)
fix *vs*
assume *vs* $\in$ *fo_nmlz AD* ` *proj_fmla* (*Conj φ ψ*) {*σ. esat φ I σ UNIV*} $\cap$
fo_nmlz AD ` *proj_fmla* (*Conj φ ψ*) {*σ. esat ψ I σ UNIV*}
then obtain *σ σ'* **where** *σ_def*:
σ $\in$ {*σ. esat φ I σ UNIV*} *vs* = *fo_nmlz AD* (*map σ* (*fv_fo_fmla_list* (*Conj φ ψ*)))
σ' $\in$ {*σ. esat ψ I σ UNIV*} *vs* = *fo_nmlz AD* (*map σ'* (*fv_fo_fmla_list* (*Conj φ ψ*)))
unfolding *proj_fmla_map*
by *blast*
have *ad_sub*: *act_edom ψ I* $\subseteq$ *AD*
using *assms*(*1*)

by *auto*
have *ad_agr*: *ad_agr_list AD* (*map σ* (*fv_fo_fmla_list ψ*)) (*map σ′* (*fv_fo_fmla_list ψ*))
by (*rule ad_agr_list_subset*[*OF _ fo_nmlz_eqD*[*OF trans*[*OF σ_def*(*2*)[*symmetric*] *σ_def*(*4*)]]]])
(*auto simp*: *fv_fo_fmla_list_set*)
have *σ* ∈ {*σ*. *esat ψ I σ UNIV*}
using *esat_UNIV_cong*[*OF ad_agr_sets_restrict*[*OF iffD2*[*OF ad_agr_list_link*]],
OF ad_agr ad_sub] *σ_def*(*3*)
by *blast*
then show *vs* ∈ *fo_nmlz AD* ‘ *proj_fmla* (*Conj φ ψ*) {*σ*. *esat* (*Conj φ ψ*) *I σ UNIV*}
using *σ_def*(*1*,*2*)
by (*auto simp*: *proj_fmla_map*)
qed

lemma *eval_conj*:
fixes *φ* :: (*′a* :: *infinite*, *′b*) *fo_fmla*
assumes *wf*: *fo_wf φ I tφ fo_wf ψ I tψ*
shows *fo_wf* (*Conj φ ψ*) *I* (*eval_conj* (*fv_fo_fmla_list φ*) *tφ* (*fv_fo_fmla_list ψ*) *tψ*)
proof −
obtain *ADφ nφ Xφ ADψ nψ Xψ* **where** *ts_def*:
tφ = (*ADφ*, *nφ*, *Xφ*) *tψ* = (*ADψ*, *nψ*, *Xψ*)
ADφ = *act_edom φ I ADψ* = *act_edom ψ I*
using *assms*
by (*cases tφ*, *cases tψ*) *auto*
have *AD_sub*: *act_edom φ I* ⊆ *ADφ act_edom ψ I* ⊆ *ADψ*
by (*auto simp*: *ts_def*(*3*,*4*))

obtain *AD n X* **where** *AD_X_def*:
eval_conj (*fv_fo_fmla_list φ*) *tφ* (*fv_fo_fmla_list ψ*) *tψ* = (*AD*, *n*, *X*)
by (*cases eval_conj* (*fv_fo_fmla_list φ*) *tφ* (*fv_fo_fmla_list ψ*) *tψ*) *auto*
have *AD_def*: *AD* = *act_edom* (*Conj φ ψ*) *I act_edom* (*Conj φ ψ*) *I* ⊆ *AD*
ADφ ⊆ *AD ADψ* ⊆ *AD AD* = *ADφ* ∪ *ADψ*
using *AD_X_def*
by (*auto simp*: *ts_def Let_def*)
have *n_def*: *n* = *nfv* (*Conj φ ψ*)
using *AD_X_def*
by (*auto simp*: *ts_def Let_def nfv_card fv_fo_fmla_list_set*)

define *Sφ* **where** *Sφ* ≡ {*σ*. *esat φ I σ UNIV*}
define *Sψ* **where** *Sψ* ≡ {*σ*. *esat ψ I σ UNIV*}
define *ADΔφ* **where** *ADΔφ* = *AD* − *ADφ*
define *ADΔψ* **where** *ADΔψ* = *AD* − *ADψ*
define *nsφ* **where** *nsφ* = *fv_fo_fmla_list φ*
define *nsψ* **where** *nsψ* = *fv_fo_fmla_list ψ*
define *ns* **where** *ns* = *filter* (*λn*. *n* ∈ *fv_fo_fmla φ*) (*fv_fo_fmla_list ψ*)
define *nsφ′* **where** *nsφ′* = *filter* (*λn*. *n* ∉ *fv_fo_fmla φ*) (*fv_fo_fmla_list ψ*)
define *nsψ′* **where** *nsψ′* = *filter* (*λn*. *n* ∉ *fv_fo_fmla ψ*) (*fv_fo_fmla_list φ*)

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

have *sd_ns*: *sorted_distinct nsφ sorted_distinct nsψ*
by (*auto simp*: *nsφ_def nsψ_def sorted_distinct_fv_list*)
have *ad_agr_Xφ*: *ad_agr_close_set ADΔφ Xφ* = *fo_nmlz AD* ‘ *proj_vals Sφ nsφ*
unfolding *Xφ_def ad_agr_close_set_nmlz_eq nsφ_def*[*symmetric*] *ADΔφ_def*
apply (*rule ad_agr_close_set_correct*[*OF AD_def*(*3*) *sd_ns*(*1*)])
using *AD_sub*(*1*) *esat_UNIV_ad_agr_list*
by (*fastforce simp*: *ad_agr_list_link Sφ_def nsφ_def*)
have *ad_agr_Xψ*: *ad_agr_close_set ADΔψ Xψ* = *fo_nmlz AD* ‘ *proj_vals Sψ nsψ*

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) 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_fmla (Conj φ ψ) = fv_fo_fmla φ $\cup$ set (fv_fo_fmla_list ψ)
fv_fo_fmla (Conj φ ψ) = fv_fo_fmla ψ $\cup$ set (fv_fo_fmla_list φ)
by (auto simp: fv_fo_fmla_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_fmla_list_set]
$ns\psi'$_def[folded fv_fo_fmla_list_set] res_left_alt(2) res_right_alt(2)
sorted_distinct_fv_list sorted_distinct_fv_list]
have X = fo_nmlz AD ‘ proj_fmla (Conj φ ψ) $\{\sigma.$ esat φ I σ UNIV$\}$ $\cap$
fo_nmlz AD ‘ proj_fmla (Conj φ ψ) $\{\sigma.$ 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 $ns\varphi$_def $ns\psi$_def $AD\Delta\varphi$_def $AD\Delta\psi$_def])
unfolding eval_conj_set proj_fmla_def
unfolding res_left_alt(1) res_right_alt(1) $S\varphi$_def $S\psi$_def
by auto
then have eval: eval_conj (fv_fo_fmla_list φ) $t\varphi$ (fv_fo_fmla_list ψ) $t\psi$ =
eval_abs (Conj φ ψ) I
using proj_fmla_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_fmla_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: $(\bigwedge w\ z\ Z.\ Z \subseteq X \Longrightarrow z \in Z \Longrightarrow w \in f\ (h\ z)\ \{z\} \Longrightarrow w \in f\ (h\ z)\ Z) \Longrightarrow$
$(\bigwedge w\ z\ Z.\ Z \subseteq X \Longrightarrow z \in Z \Longrightarrow w \in f\ (h\ z)\ Z \Longrightarrow (\exists z' \in Z.\ w \in f\ (h\ z)\ \{z'\})) \Longrightarrow$
set_of_idx (Mapping.map_values f (cluster (Some $\circ$ h) X)) = $\bigcup((\lambda x.\ f\ (h\ x)\ \{x\})$ ‘ 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 $\subseteq$ 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 $\bigwedge x\ y.$ *Mapping.lookup t x = Some y* $\Longrightarrow$ *f x y = f' x y*
shows *Mapping.map_values f t = Mapping.map_values f' t*
apply (*auto simp*: *lookup_map_values intro*!: *mapping_eqI*)
subgoal for *x*
using *assms*
by (*cases Mapping.lookup t x*) *auto*
done

lemma *ad_agr_close_set_length*: *z* $\in$ *ad_agr_close_set AD X* $\Longrightarrow$ ($\bigwedge x.$ *x* $\in$ *X* $\Longrightarrow$ *length x = n*) $\Longrightarrow$ *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* $\in$ *ad_agr_close_set* (*AD* − *AD'*) *X* $\Longrightarrow$ ($\bigwedge x.$ *x* $\in$ *X* $\Longrightarrow$ *fo_nmlzd AD' x*) $\Longrightarrow$ *AD'* $\subseteq$ *AD* $\Longrightarrow$ *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* $\in$ *ext_tuple_set AD ns ns' X* $\Longrightarrow$ ($\bigwedge x.$ *x* $\in$ *X* $\Longrightarrow$ *length x = length ns*) $\Longrightarrow$ *length z = length ns + 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 φ :: (*'a* :: *infinite*, *'b*) *fo_fmla*
assumes *wf*: *fo_wf* φ *I* $t\varphi$ *fo_wf* ψ *I* $t\psi$
shows *fo_wf* (*Conj* φ (*Neg* ψ)) *I*
(*eval_ajoin* (*fv_fo_fmla_list* φ) $t\varphi$ (*fv_fo_fmla_list* ψ) $t\psi$)
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* φ *I* $AD\psi$ = *act_edom* ψ *I*
using *assms*
by (*cases* $t\varphi$, *cases* $t\psi$) *auto*
have *AD_sub*: *act_edom* φ *I* $\subseteq$ $AD\varphi$ *act_edom* ψ *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* φ) $t\varphi$ (*fv_fo_fmla_list* ψ) $t\psi$ = (*AD*, *n*, *X*)
by (*cases eval_ajoin* (*fv_fo_fmla_list* φ) $t\varphi$ (*fv_fo_fmla_list* ψ) $t\psi$) *auto*
have *AD_def*: *AD = act_edom* (*Conj* φ (*Neg* ψ)) *I*
act_edom (*Conj* φ (*Neg* ψ)) *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* φ (*Neg* ψ))
using *AD_X_def*
by (*auto simp*: *ts_def Let_def nfv_card fv_fo_fmla_list_set*)

define $S\varphi$ **where** $S\varphi \equiv \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
define $S\psi$ **where** $S\psi \equiv \{\sigma.\ esat\ \psi\ I\ \sigma\ UNIV\}$
define *both* **where** *both* = *remdups_adj* (*sort* (*fv_fo_fmla_list* φ @ *fv_fo_fmla_list* ψ))
define $ns\varphi'$ **where** $ns\varphi'$ = *filter* ($\lambda n.\ n \notin$ *fv_fo_fmla* φ) (*fv_fo_fmla_list* ψ)
define $ns\psi'$ **where** $ns\psi'$ = *filter* ($\lambda n.\ n \notin$ *fv_fo_fmla* ψ) (*fv_fo_fmla_list* φ)

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_fmla_list* φ
define $ns\psi$ **where** $ns\psi$ = *fv_fo_fmla_list* ψ
define *ns* **where** *ns* = *filter* ($\lambda n.\ n \in$ *set* $ns\psi$) $ns\varphi$
define $X\varphi'$ **where** $X\varphi'$ = *ext_tuple_set* *AD* $ns\varphi$ $ns\varphi'$ (*ad_agr_close_set* $AD\Delta\varphi$ $X\varphi$)
define $idx\varphi$ **where** $idx\varphi$ = *cluster* (*Some* $\circ$ ($\lambda xs.$ *fo_nmlz* $AD\psi$ (*proj_tuple* *ns* (*zip* $ns\varphi$ *xs*)))) (*ad_agr_close_set* $AD\Delta\varphi$ $X\varphi$)
define $idx\psi$ **where** $idx\psi$ = *cluster* (*Some* $\circ$ ($\lambda ys.$ *fo_nmlz* $AD\psi$ (*proj_tuple* *ns* (*zip* $ns\psi$ *ys*)))) $X\psi$
define *res* **where** *res* = *Mapping.map_values* ($\lambda xs\ X.$ *case* *Mapping.lookup* $idx\psi$ *xs* *of*
Some $Y \Rightarrow$ *eval_conj_set* *AD* $ns\varphi$ *X* $ns\psi$ (*ad_agr_close_set* $AD\Delta\psi$ (*ext_tuple_set* $AD\psi$ *ns* $ns\varphi'$ $\{xs\} - Y$))
| _ $\Rightarrow$ *ext_tuple_set* *AD* $ns\varphi$ $ns\varphi'$ *X*) $idx\varphi$

note $X\varphi$*_def* = *fo_wf_X*[*OF* *wf*(*1*)[*unfolded* *ts_def*(*1*)], *unfolded* *proj_fmla_def*, *folded* $S\varphi$*_def*]
note $X\psi$*_def* = *fo_wf_X*[*OF* *wf*(*2*)[*unfolded* *ts_def*(*2*)], *unfolded* *proj_fmla_def*, *folded* $S\psi$*_def*]

have *fv_sub*: *fv_fo_fmla* (*Conj* φ (*Neg* ψ)) = *fv_fo_fmla* ψ $\cup$ *set* (*fv_fo_fmla_list* φ)
by (*auto* *simp*: *fv_fo_fmla_list_set*)
have *fv_sort*: *fv_fo_fmla_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_fmla_list_def* *distinct_remdups_adj_sort*)

have *AD_disj*: $AD\varphi \cap AD\Delta\varphi = \{\}$ $AD\psi \cap AD\Delta\psi = \{\}$
by (*auto* *simp*: $AD\Delta\varphi$*_def* $AD\Delta\psi$*_def*)
have *AD_delta*: $AD = AD\varphi \cup AD\Delta\varphi$ $AD = AD\psi \cup AD\Delta\psi$
by (*auto* *simp*: $AD\Delta\varphi$*_def* $AD\Delta\psi$*_def* *AD_def* *ts_def*)
have *fo_nmlzd_X*: *Ball* $X\varphi$ (*fo_nmlzd* $AD\varphi$) *Ball* $X\psi$ (*fo_nmlzd* $AD\psi$)
using *wf*
by (*auto* *simp*: *ts_def*)
have *Ball_ad_agr*: *Ball* (*ad_agr_close_set* $AD\Delta\varphi$ $X\varphi$) (*fo_nmlzd* *AD*)
using *ad_agr_close_sound*[**where** *?X*=$AD\varphi$ **and** *?Y*=$AD\Delta\varphi$] *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:
$\bigwedge\sigma\ \tau.$ *ad_agr_sets* (*set* (*fv_fo_fmla_list* φ)) (*set* (*fv_fo_fmla_list* φ)) $AD\varphi\ \sigma\ \tau \Longrightarrow \sigma \in S\varphi \longleftrightarrow \tau \in S\varphi$
$\bigwedge\sigma\ \tau.$ *ad_agr_sets* (*set* (*fv_fo_fmla_list* φ)) (*set* (*fv_fo_fmla_list* φ)) $AD\ \sigma\ \tau \Longrightarrow \sigma \in S\varphi \longleftrightarrow \tau \in S\varphi$
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 $S\varphi$*_def*
by *blast+*
have *ad_agr_*$S\varphi$: $\tau' \in S\varphi \Longrightarrow$ *ad_agr_list* $AD\varphi$ (*map* τ' $ns\varphi$) (*map* τ'' $ns\varphi$) $\Longrightarrow \tau'' \in S\varphi$ **for** $\tau'\ \tau''$
using *ad_agr_*φ
by (*auto* *simp*: *ad_agr_list_link* $ns\varphi$*_def*)
have *ad_agr_*ψ:
$\bigwedge\sigma\ \tau.$ *ad_agr_sets* (*set* (*fv_fo_fmla_list* ψ)) (*set* (*fv_fo_fmla_list* ψ)) $AD\psi\ \sigma\ \tau \Longrightarrow \sigma \in S\psi \longleftrightarrow \tau \in S\psi$
using *esat_UNIV_cong*[*OF* *ad_agr_sets_restrict*, *OF* _ *subset_refl*] *ad_agr_sets_mono*[*OF* *AD_sub*(*2*)]

unfolding $S\psi$_def
by *blast+*
have *ad_agr_Sψ*: $\tau' \in S\psi \Longrightarrow$ *ad_agr_list* $AD\psi$ (*map* τ' $ns\psi$) (*map* τ'' $ns\psi$) $\Longrightarrow \tau'' \in S\psi$ **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* $\cap$ *set nsφ'* = {} *set nsφ* $\cup$ *set nsφ'* = *set both*
by (*auto simp*: *nsφ_def nsφ'_def fv_sort*[*symmetric*] *fv_fo_fmla_list_set sorted_distinct_fv_list intro*: *sorted_filter*[**where** *?f=id*, *simplified*])
have *aux2*: *nsφ'* = *filter* ($\lambda n.$ $n \notin$ *set nsφ*) *nsφ' nsφ* = *filter* ($\lambda n.$ $n \notin$ *set nsφ'*) *nsφ*
by (*auto simp*: *nsφ_def nsφ'_def nsψ_def nsψ'_def fv_fo_fmla_list_set*)
have *aux3*: *set nsφ'* $\cap$ *set ns* = {} *set nsφ'* $\cup$ *set ns* = *set nsψ*
by (*auto simp*: *nsφ_def nsφ'_def nsψ_def ns_def fv_fo_fmla_list_set*)
have *aux4*: *set ns* $\cap$ *set nsφ'* = {} *set ns* $\cup$ *set nsφ'* = *set nsψ*
by (*auto simp*: *nsφ_def nsφ'_def nsψ_def ns_def fv_fo_fmla_list_set*)
have *aux5*: *nsφ'* = *filter* ($\lambda n.$ $n \notin$ *set nsφ*) *nsψ nsψ'* = *filter* ($\lambda n.$ $n \notin$ *set nsψ*) *nsφ*
by (*auto simp*: *nsφ_def nsφ'_def nsψ_def nsψ'_def fv_fo_fmla_list_set*)
have *aux6*: *set nsψ* $\cap$ *set nsψ'* = {} *set nsψ* $\cup$ *set nsψ'* = *set both*
by (*auto simp*: *nsφ_def nsφ'_def nsψ_def nsψ'_def both_def fv_fo_fmla_list_set*)
have *ns_sd*: *sorted_distinct ns sorted_distinct nsφ sorted_distinct nsψ set ns* $\subseteq$ *set nsφ set ns* $\subseteq$ *set nsψ set ns* $\subseteq$ *set both set nsφ'* $\subseteq$ *set nsψ set nsψ* $\subseteq$ *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* ($\lambda n.$ $n \in$ *fv_fo_fmla* φ) (*fv_fo_fmla_list* ψ)
by (*rule sorted_distinct_set_unique*)
(*auto simp*: *ns_def nsφ_def nsψ_def fv_fo_fmla_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_compl*[**where** *?P=*$\lambda n.$ $n \in$ *fv_fo_fmla* φ **and** *?xs=fv_fo_fmla_list* ψ]
by (*auto simp*: *ns nsφ_def nsφ'_def nsψ_def fv_fo_fmla_list_set*)

have *res_eq*: *res* = *Mapping.map_values* (λ*xs X*. *case Mapping.lookup idxψ xs of*
Some Y $\Rightarrow$ *idx_join AD ns nsφ X nsψ* (*ad_agr_close_set ADΔψ* (*ext_tuple_set ADψ ns nsφ'* {*xs*} − *Y*))
| _ $\Rightarrow$ *ext_tuple_set AD nsφ nsφ' X*) *idxφ*
proof −
have *ad_agr_Xφ*: *ad_agr_close_set ADΔφ 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 ADΔφ_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* $\in$ *y* $\Longrightarrow$ *fo_nmlzd AD vs* $\wedge$ *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* $\in$ *ad_agr_close_set ADΔψ* (*ext_tuple_set ADψ ns nsφ'* {*x*} − *x2*) $\Longrightarrow$ *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* $\in$ *ad_agr_close_set AD*$\Delta\psi$ (*ext_tuple_set AD*ψ *ns ns*φ' $\{x\}$ $-$ *x2*) $\Longrightarrow$ *length vs* = *length ns*ψ **for** *vs*
apply (*erule ad_agr_close_set_length*)
apply (*rule ext_tuple_set_length*[**where** *?AD*=*AD*ψ **and** *?ns*=*ns* **and** *?ns'*=*ns*φ' **and** *?X*=$\{x\}$, *unfolded len_ns*ψ])
using *lup*(*1*) *ns_sd*(*1*,*2*,*4*)
by (*auto simp*: *idx*φ*_def lookup_cluster' fo_nmlz_length ad_agr_X*φ *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 ns*φ $\{x\}$ *ns*ψ (*ad_agr_close_set AD*$\Delta\psi$ (*ext_tuple_set AD*ψ *ns ns*φ' $\{$*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))$\}$ $-$ *Y*)) =
ext_tuple_set AD nsφ *ns*φ' $\{x\}$ $\cap$ *ext_tuple_set AD ns*ψ *ns*ψ' (*fo_nmlz AD* ‘ *proj_vals* $\{\sigma \in -$ *S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*ψ)
if *x_ns*: *proj_tuple ns* (*zip ns*φ *x*) = *map* σ' *ns*
and *x_proj_singleton*: $\{x\}$ = *fo_nmlz AD* ‘ *proj_vals* $\{\sigma\}$ *ns*φ
and *Some*: *Mapping.lookup idx*ψ (*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))) = *Some Y*
for *x Y* σ σ'
proof $-$
have *Y* = $\{$*ys* $\in$ *fo_nmlz AD*ψ ‘ *proj_vals S*ψ *ns*ψ. *fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*ψ *ys*)) = *fo_nmlz AD*ψ (*map* σ' *ns*)$\}$
using *Some*
apply (*auto simp*: *X*ψ*_def idx*ψ*_def ns*ψ*_def x_ns lookup_cluster' split*: *if_splits*)
done
moreover have ... = *fo_nmlz AD*ψ ‘ *proj_vals* $\{\sigma \in$ *S*ψ. *fo_nmlz AD*ψ (*map* σ *ns*) = *fo_nmlz AD*ψ (*map* σ' *ns*)$\}$ *ns*ψ
by (*auto simp*: *proj_vals_def fo_nmlz_twice*[*OF ns_sd*(*1*,*3*,*5*)])+
moreover have ... = *fo_nmlz AD*ψ ‘ *proj_vals* $\{\sigma \in$ *S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*ψ
by (*auto simp*: *fo_nmlz_eq*)
ultimately have *Y_def*: *Y* = *fo_nmlz AD*ψ ‘ *proj_vals* $\{\sigma \in$ *S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*ψ
by *auto*
have *R_def*: $\{$*fo_nmlz AD*ψ (*map* σ' *ns*)$\}$ = *fo_nmlz AD*ψ ‘ *proj_vals* $\{\sigma$. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*
using *ad_agr_list_refl*
by (*auto simp*: *proj_vals_def intro*: *fo_nmlz_eqI*)
have *ext_tuple_set AD*ψ *ns ns*φ' $\{$*fo_nmlz AD*ψ (*map* σ' *ns*)$\}$ = *fo_nmlz AD*ψ ‘ *proj_vals* $\{\sigma$. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*ψ
apply (*rule ext_tuple_correct*[*OF ns_sd*(*1*) *aux*(*2*) *ns_sd*(*3*) *aux4 R_def*])
using *ad_agr_list_trans ad_agr_list_comm*
apply (*auto simp*: *ad_agr_list_link*)
by *fast*
then have *ext_tuple_set AD*ψ *ns ns*φ' $\{$*fo_nmlz AD*ψ (*map* σ' *ns*)$\}$ $-$ *Y* = *fo_nmlz AD*ψ ‘ *proj_vals* $\{\sigma \in -$*S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*ψ
apply (*auto simp*: *Y_def proj_vals_def fo_nmlz_eq*)
using *ad_agr_S*ψ *ad_agr_list_comm*
by *blast*+
moreover have *ad_agr_close_set AD*$\Delta\psi$ (*fo_nmlz AD*ψ ‘ *proj_vals* $\{\sigma \in -$*S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*ψ) =
fo_nmlz AD ‘ *proj_vals* $\{\sigma \in -$*S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)$\}$ *ns*ψ
unfolding *ad_agr_close_set_eq*[*OF Ball_fo_nmlzd*]

apply (*rule ad_agr_close_set_correct*[*OF AD_def*(*4*) *ns_sd*(*3*), *folded AD*$\Delta\psi$*_def*])
apply (*auto simp*: *ad_agr_list_link*)
using *ad_agr_S*ψ *ad_agr_list_comm ad_agr_list_subset*[*OF ns_sd*(*5*)] *ad_agr_list_trans*
by *blast+*
ultimately have *comp_proj*: *ad_agr_close_set AD*$\Delta\psi$ (*ext_tuple_set AD*ψ *ns ns*φ' {*fo_nmlz AD*ψ (*map* σ' *ns*)} − *Y*) =
fo_nmlz AD ' *proj_vals* {$\sigma \in -S\psi$. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)} *ns*ψ
by *simp*
have *ext_tuple_set AD ns*ψ *ns*ψ' (*fo_nmlz AD* ' *proj_vals* {$\sigma \in -$ *S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)} *ns*ψ) = *fo_nmlz AD* ' *proj_vals* {$\sigma \in -$ *S*ψ. *ad_agr_list AD*ψ (*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_S*ψ *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+*
show *eval_conj_set AD ns*φ {*x*} *ns*ψ (*ad_agr_close_set AD*$\Delta\psi$ (*ext_tuple_set AD*ψ *ns ns*φ' {*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))} − *Y*)) =
ext_tuple_set AD nsφ *ns*φ' {*x*} $\cap$ *ext_tuple_set AD ns*ψ *ns*ψ' (*fo_nmlz AD* ' *proj_vals* {$\sigma \in -$ *S*ψ. *ad_agr_list AD*ψ (*map* σ *ns*) (*map* σ' *ns*)} *ns*ψ)
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_fmla_list_set*
nsφ'*_def*[*symmetric*] *fv_sort*[*symmetric*] *proj_fmla_def S*φ*_def*[*symmetric*] *S*ψ*_def*[*symmetric*]
AD$\Delta\varphi$*_def*[*symmetric*] *AD*$\Delta\psi$*_def*[*symmetric*]
nsφ*_def*[*symmetric*] *ns*φ'*_def*[*symmetric*, *folded fv_fo_fmla_list_set*[*of* φ, *folded ns*φ*_def*] *ns*ψ*_def*] *ns*ψ*_def*[*symmetric*] *ns_def*[*symmetric*]
Xφ'*_def*[*symmetric*] *idx*φ*_def*[*symmetric*] *idx*ψ*_def*[*symmetric*] *res_eq*[*symmetric*]
by *auto*
moreover have ... = ($\bigcup$*x*$\in$*ad_agr_close_set AD*$\Delta\varphi$ *X*φ.
case Mapping.lookup idxψ (*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))) *of None* $\Rightarrow$ *ext_tuple_set AD ns*φ *ns*φ' {*x*}
| *Some Y* $\Rightarrow$ *eval_conj_set AD ns*φ {*x*} *ns*ψ (*ad_agr_close_set AD*$\Delta\psi$ (*ext_tuple_set AD*ψ *ns ns*φ' {*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))} − *Y*)))
unfolding *res_def*[*unfolded idx*φ*_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_fmla* (*Conj* φ (*Neg* ψ)) {σ. *esat* φ *I* σ *UNIV*} −
fo_nmlz AD ' *proj_fmla* (*Conj* φ (*Neg* ψ)) {σ. *esat* ψ *I* σ *UNIV*}
unfolding *S*φ*_def*[*symmetric*] *S*ψ*_def*[*symmetric*] *proj_fmla_def fv_sort*
proof (*rule set_eqI*, *rule iffI*)
fix *t*
assume *t* $\in$ ($\bigcup$*x*$\in$*ad_agr_close_set AD*$\Delta\varphi$ *X*φ. *case Mapping.lookup idx*ψ (*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))) *of*
None $\Rightarrow$ *ext_tuple_set AD ns*φ *ns*φ' {*x*}
| *Some Y* $\Rightarrow$ *eval_conj_set AD ns*φ {*x*} *ns*ψ (*ad_agr_close_set AD*$\Delta\psi$ (*ext_tuple_set AD*ψ *ns ns*φ' {*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))} − *Y*)))
then obtain *x* **where** *x*: *x* $\in$ *ad_agr_close_set AD*$\Delta\varphi$ *X*φ
Mapping.lookup idxψ (*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))) = *None* $\Longrightarrow$ *t* $\in$ *ext_tuple_set AD ns*φ *ns*φ' {*x*}
$\bigwedge$*Y*. *Mapping.lookup idx*ψ (*fo_nmlz AD*ψ (*proj_tuple ns* (*zip ns*φ *x*))) = *Some Y* $\Longrightarrow$

```
  t ∈ 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))
    by (fastforce split: option.splits)
  obtain σ where val: σ ∈ Sφ x = fo_nmlz AD (map σ nsφ)
    using ad_agr_close_correct[OF AD_def(3) ad_agr_φ(1), folded ADΔφ_def] Xφ_def[folded
proj_fmla_def] ad_agr_close_set_eq[OF fo_nmlzd_X(1)] x(1)
    apply (auto simp: proj_fmla_def proj_vals_def nsφ_def)
    apply fast
    done
  obtain σ' 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 σ'
    by (rule proj_tuple_map[OF ns_sd(1−2,4)])
  have ad_agr_σ_σ': ad_agr_list AD (map σ nsφ) (map σ' nsφ)
    using σ'
    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': $\tau' \notin S\psi$ $t = fo\_nmlz\ AD\ (map\ \tau'\ both)$
using *t_both*
by (*auto simp*: *proj_vals_def*)
have *False*
using $\tau\ \tau'$
apply (*auto simp*: *fo_nmlz_eq*)
using *ad_agr_S*ψ *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_S*φ[*OF t_in*(*1*)]
by *auto*
qed
next
fix *t*
assume *t_in_asm*: $t \in fo\_nmlz\ AD$ ‘ $proj\_vals\ S\varphi\ both - fo\_nmlz\ AD$ ‘ $proj\_vals\ S\psi\ both$
then obtain σ **where** *val*: $\sigma \in S\varphi$ $t = fo\_nmlz\ AD\ (map\ \sigma\ both)$
by (*auto simp*: *proj_vals_def*)
define *x* **where** $x = fo\_nmlz\ AD\ (map\ \sigma\ ns\varphi)$
obtain σ' **where** σ': $x = map\ \sigma'\ ns\varphi$
using *exists_map*[**where** *?ys=x* **and** *?xs=ns*φ] *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_*φ(*1*), *folded AD*$\Delta\varphi$*_def*] *val*(*1*)
unfolding *ad_agr_close_set_eq*[*OF fo_nmlzd_X*(*1*)] *x_def*
unfolding *X*φ*_def*[*folded proj_fmla_def*] *proj_fmla_map*
by (*fastforce simp*: *x_def ns*φ*_def*)
have *ad_agr_*σ*_*σ': $ad\_agr\_list\ AD\ (map\ \sigma\ ns\varphi)\ (map\ \sigma'\ ns\varphi)$
using σ'
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_*σ*_*σ' *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 σ'
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_*σ*_*σ'
by (*auto simp*: *val*(*2*) *proj_vals_def*)
{
fix *Y*
assume *Some*: $Mapping.lookup\ idx\psi\ (fo\_nmlz\ AD\psi\ (map\ \sigma'\ ns)) = Some\ Y$
have *tmp*: $proj\_tuple\ ns\ (zip\ ns\varphi\ x) = map\ \sigma'\ ns$
unfolding σ'
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_Sψ 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 ADψ* (*map σ ns*) (*map σ′ ns*)
using *ad_agr_σ_σ′ 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 nsψ nsψ′* (*fo_nmlz AD* ‘ *proj_vals* {$\sigma \in - S\psi$. *ad_agr_list ADψ* (*map σ ns*) (*map σ′ ns*)} *nsψ*)
unfolding *unfold val*(*2*)
by (*auto simp*: *proj_vals_def*)
then have $t \in$ *eval_conj_set AD nsφ* {*x*} *nsψ* (*ad_agr_close_set AD*Δ*ψ* (*ext_tuple_set ADψ ns nsφ′* {*fo_nmlz ADψ* (*map σ′ 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 $t \in (\bigcup x \in$ *ad_agr_close_set AD*Δ*φ Xφ*. *case Mapping.lookup idxψ* (*fo_nmlz ADψ* (*proj_tuple ns* (*zip nsφ x*))) *of*
None ⇒ *ext_tuple_set AD nsφ nsφ′* {*x*}
| *Some Y* ⇒ *eval_conj_set AD nsφ* {*x*} *nsψ* (*ad_agr_close_set AD*Δ*ψ* (*ext_tuple_set ADψ ns nsφ′* {*fo_nmlz ADψ* (*proj_tuple ns* (*zip nsφ x*))} − *Y*)))
using *t_in_ext_x*
by (*intro UN_I*[*OF x_in_ad_agr_close*]) (*auto simp*: *x_ns split*: *option.splits*)
qed
ultimately have *X_def*: *X* = *fo_nmlz AD* ‘ *proj_fmla* (*Conj φ* (*Neg ψ*)) {*σ*. *esat φ I σ UNIV*} −
fo_nmlz AD ‘ *proj_fmla* (*Conj φ* (*Neg ψ*)) {*σ*. *esat ψ I σ UNIV*}
by *simp*

have *AD_Neg_sub*: *act_edom* (*Neg ψ*) *I* ⊆ *AD*
by (*auto simp*: *AD_def*(*1*))
have *X* = *fo_nmlz AD* ‘ *proj_fmla* (*Conj φ* (*Neg ψ*)) {*σ*. *esat φ I σ UNIV*} ∩
fo_nmlz AD ‘ *proj_fmla* (*Conj φ* (*Neg ψ*)) {*σ*. *esat* (*Neg ψ*) *I σ UNIV*}
unfolding *X_def*
by (*auto simp*: *proj_fmla_map dest*!: *fo_nmlz_eqD*)
(*metis AD_def*(*4*) *ad_agr_list_subset esat_UNIV_ad_agr_list fv_fo_fmla_list_set fv_sub sup_ge1 ts_def*(*4*))
then have *eval*: *eval_ajoin* (*fv_fo_fmla_list φ*) *tφ* (*fv_fo_fmla_list ψ*) *tψ* =
eval_abs (*Conj φ* (*Neg ψ*)) *I*
using *proj_fmla_conj_sub*[*OF AD_Neg_sub*, *of φ*]
unfolding *AD_X_def AD_def*(*1*)[*symmetric*] *n_def eval_abs_def*
by (*auto simp*: *proj_fmla_map*)
have *wf_conj_neg*: *wf_fo_intp* (*Conj φ* (*Neg ψ*)) *I*
using *wf*
by (*auto simp*: *ts_def*)
show *?thesis*
using *fo_wf_eval_abs*[*OF wf_conj_neg*]
by (*auto simp*: *eval*)
qed

lemma *eval_disj*:
fixes *φ* :: (*′a* :: *infinite*, *′b*) *fo_fmla*
assumes *wf*: *fo_wf φ I tφ fo_wf ψ I tψ*
shows *fo_wf* (*Disj φ ψ*) *I*
(*eval_disj* (*fv_fo_fmla_list φ*) *tφ* (*fv_fo_fmla_list ψ*) *tψ*)
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_disj (*fv_fo_fmla_list* φ) $t\varphi$ (*fv_fo_fmla_list* ψ) $t\psi$ = (*AD*, *n*, *X*)
by (*cases eval_disj* (*fv_fo_fmla_list* φ) $t\varphi$ (*fv_fo_fmla_list* ψ) $t\psi$) *auto*
have *AD_def*: *AD* = *act_edom* (*Disj* φ ψ) *I* *act_edom* (*Disj* φ ψ) *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* (*Disj* φ ψ)
using *AD_X_def*
by (*auto simp*: *ts_def Let_def nfv_card fv_fo_fmla_list_set*)

define $S\varphi$ **where** $S\varphi \equiv \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
define $S\psi$ **where** $S\psi \equiv \{\sigma.\ esat\ \psi\ I\ \sigma\ UNIV\}$
define $ns\varphi'$ **where** $ns\varphi'$ = *filter* ($\lambda n.\ n \notin$ *fv_fo_fmla* φ) (*fv_fo_fmla_list* ψ)
define $ns\psi'$ **where** $ns\psi'$ = *filter* ($\lambda n.\ n \notin$ *fv_fo_fmla* ψ) (*fv_fo_fmla_list* φ)

note $X\varphi$*_def* = *fo_wf_X*[*OF wf*(*1*)[*unfolded ts_def*(*1*)], *unfolded proj_fmla_def*, *folded* $S\varphi$*_def*]
note $X\psi$*_def* = *fo_wf_X*[*OF wf*(*2*)[*unfolded ts_def*(*2*)], *unfolded proj_fmla_def*, *folded* $S\psi$*_def*]
have *fv_sub*: *fv_fo_fmla* (*Disj* φ ψ) = *fv_fo_fmla* φ $\cup$ *set* (*fv_fo_fmla_list* ψ)
fv_fo_fmla (*Disj* φ ψ) = *fv_fo_fmla* ψ $\cup$ *set* (*fv_fo_fmla_list* φ)
by (*auto simp*: *fv_fo_fmla_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*)]

have *X* = *fo_nmlz AD* ‘ *proj_fmla* (*Disj* φ ψ) $\{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$ $\cup$
fo_nmlz AD ‘ *proj_fmla* (*Disj* φ ψ) $\{\sigma.\ esat\ \psi\ I\ \sigma\ UNIV\}$
using *AD_X_def*
apply (*simp add*: *ts_def*(*1*,*2*) *Let_def AD_def*(*5*)[*symmetric*])
unfolding *fv_fo_fmla_list_set proj_fmla_def* $ns\varphi'$*_def*[*symmetric*] $ns\psi'$*_def*[*symmetric*]
$S\varphi$*_def*[*symmetric*] $S\psi$*_def*[*symmetric*]
using *res_left_alt*(*1*) *res_right_alt*(*1*)
by *auto*
then have *eval*: *eval_disj* (*fv_fo_fmla_list* φ) $t\varphi$ (*fv_fo_fmla_list* ψ) $t\psi$ =
eval_abs (*Disj* φ ψ) *I*
unfolding *AD_X_def AD_def*(*1*)[*symmetric*] *n_def eval_abs_def*
by (*auto simp*: *proj_fmla_map*)
have *wf_disj*: *wf_fo_intp* (*Disj* φ ψ) *I*
using *wf*
by (*auto simp*: *ts_def*)
show *?thesis*
using *fo_wf_eval_abs*[*OF wf_disj*]
by (*auto simp*: *eval*)
qed

lemma *fv_ex_all*:
assumes *pos i* (*fv_fo_fmla_list* φ) = *None*
shows *fv_fo_fmla_list* (*Exists i* φ) = *fv_fo_fmla_list* φ
fv_fo_fmla_list (*Forall i* φ) = *fv_fo_fmla_list* φ

```
  using pos_complete[of i fv_fo_fmla_list φ] fv_fo_fmla_list_eq[of Exists i φ φ]
    fv_fo_fmla_list_eq[of Forall i φ φ] assms
  by (auto simp: fv_fo_fmla_list_set)

lemma nfv_ex_all:
  assumes Some: pos i (fv_fo_fmla_list φ) = Some j
  shows nfv φ = Suc (nfv (Exists i φ)) nfv φ = Suc (nfv (Forall i φ))
proof –
  have i ∈ fv_fo_fmla φ j < nfv φ i ∈ set (fv_fo_fmla_list φ)
    using fv_fo_fmla_list_set pos_set[of i fv_fo_fmla_list φ]
      pos_length[of i fv_fo_fmla_list φ] Some
    by (fastforce simp: nfv_def)+
  then show nfv φ = Suc (nfv (Exists i φ)) nfv φ = Suc (nfv (Forall i φ))
    using nfv_card[of φ] nfv_card[of Exists i φ] nfv_card[of Forall i φ]
    by (auto simp: finite_fv_fo_fmla)
qed

lemma fv_fo_fmla_list_exists: fv_fo_fmla_list (Exists n φ) = filter ((≠) n) (fv_fo_fmla_list φ)
  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 φ :: ('a :: infinite, 'b) fo_fmla
  assumes wf: fo_wf φ I t
  shows fo_wf (Exists i φ) I (eval_exists i (fv_fo_fmla_list φ) t)
proof –
  obtain AD n X where t_def: t = (AD, n, X)
    AD = act_edom φ I AD = act_edom (Exists i φ) I
    using assms
    by (cases t) auto
  note X_def = fo_wf_X[OF wf[unfolded t_def], folded t_def(2)]
  have eval: eval_exists i (fv_fo_fmla_list φ) t = eval_abs (Exists i φ) I
  proof (cases pos i (fv_fo_fmla_list φ))
    case None
    note fv_eq = fv_ex_all[OF None]
    have X = fo_nmlz AD ` proj_fmla (Exists i φ) {σ. esat φ I σ UNIV}
      unfolding X_def
      by (auto simp: proj_fmla_def fv_eq)
    also have ... = fo_nmlz AD ` proj_fmla (Exists i φ) {σ. esat (Exists i φ) I σ UNIV}
      using esat_exists_not_fv[of i φ 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 fo_nmlz AD ` rem_nth j ` X =
      fo_nmlz AD ` proj_fmla (Exists i φ) {σ. esat (Exists i φ) I σ UNIV}
    proof (rule set_eqI, rule iffI)
      fix vs
      assume vs ∈ fo_nmlz AD ` rem_nth j ` X
      then obtain ws where ws_def: ws ∈ fo_nmlz AD ` proj_fmla φ {σ. esat φ I σ UNIV}
        vs = fo_nmlz AD (rem_nth j ws)
        unfolding X_def
        by auto
      then obtain σ where σ_def: esat φ I σ UNIV
        ws = fo_nmlz AD (map σ (fv_fo_fmla_list φ))
```

```
    by (auto simp: proj_fmla_map)
  obtain τ where τ_def: ws = map τ (fv_fo_fmla_list φ)
    using fo_nmlz_map σ_def(2)
    by blast
  have esat_τ: esat (Exists i φ) I τ UNIV
    using esat_UNIV_ad_agr_list[OF fo_nmlz_ad_agr[of AD map σ (fv_fo_fmla_list φ),
          folded σ_def(2), unfolded τ_def]] σ_def(1)
    by (auto simp: t_def intro!: exI[of _ τ i])
  have rem_nth_ws: rem_nth j ws = map τ (fv_fo_fmla_list (Exists i φ))
    using rem_nth_sound[of fv_fo_fmla_list φ i j τ] sorted_distinct_fv_list Some
    unfolding fv_fo_fmla_list_exists τ_def
    by auto
  have vs ∈ fo_nmlz AD ‘ proj_fmla (Exists i φ) {σ. esat (Exists i φ) I σ UNIV}
    using ws_def(2) esat_τ
    unfolding rem_nth_ws
    by (auto simp: proj_fmla_map)
  then show vs ∈ fo_nmlz AD ‘ proj_fmla (Exists i φ) {σ. esat (Exists i φ) I σ UNIV}
    by auto
next
  fix vs
  assume assm: vs ∈ fo_nmlz AD ‘ proj_fmla (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_fmla_list (Exists i φ)))
    esat (Exists i φ) I σ UNIV
    by (auto simp: proj_fmla_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_fmla_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_fmla φ {σ. esat φ I σ UNIV}
    using x_def ws_def
    by (auto simp: fo_nmlz_sound proj_fmla_map)
  obtain τ where τ_def: ws = map τ (fv_fo_fmla_list φ)
    using fo_nmlz_map ws_def
    by blast
  have rem_nth_ws: rem_nth j ws = map τ (fv_fo_fmla_list (Exists i φ))
    using rem_nth_sound[of fv_fo_fmla_list φ i j] sorted_distinct_fv_list Some
    unfolding fv_fo_fmla_list_exists τ_def
    by auto
  have set (fv_fo_fmla_list (Exists i φ)) ⊆ set (fv_fo_fmla_list φ)
    by (auto simp: fv_fo_fmla_list_exists)
  then have ad_agr: ad_agr_list AD (map (σ(i := x)) (fv_fo_fmla_list (Exists i φ)))
    (map τ (fv_fo_fmla_list (Exists i φ)))
    by (rule ad_agr_list_subset)
      (rule fo_nmlz_ad_agr[of AD map (σ(i := x)) (fv_fo_fmla_list φ), folded ws_def,
          unfolded τ_def])
  have map_fv_cong: map (σ(i := x)) (fv_fo_fmla_list (Exists i φ)) =
    map σ (fv_fo_fmla_list (Exists i φ))
    by (auto simp: fv_fo_fmla_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_fmla_list_forall*: *fv_fo_fmla_list* (*Forall n* φ) = *filter* (($\neq$) *n*) (*fv_fo_fmla_list* φ)
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 *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* $\Longrightarrow$ *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 *ad_agr_list_take_drop*: *ad_agr_list AD xs ys* $\Longrightarrow$
ad_agr_list AD (*take i xs* @ *drop* (*Suc i*) *xs*) (*take i ys* @ *drop* (*Suc i*) *ys*)
apply (*auto simp*: *ad_agr_list_def ad_equiv_list_def sp_equiv_list_def*)
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 *ad_agr*: *ad_agr_list AD* (*add_nth i z zs*) (*fo_nmlz AD* (*add_nth i z zs*))
using *fo_nmlz_ad_agr*
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 *ad_agr_list_take_drop*[*OF ad_agr*, *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 *ad_agr_list_add*:
assumes *ad_agr_list AD xs ys i* $\leq$ *length xs*
shows $\exists$ *z'* $\in$ *Inl* ‘ *AD* $\cup$ *Inr* ‘ {..<*Suc* (*card* (*Inr* −‘ *set ys*))} $\cup$ *set ys*.
ad_agr_list 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 ($auto\ simp$: $ad\_agr\_list\_def\ n\_def$)
  obtain σ **where** σ_def: $xs = map\ \sigma\ [0..<n]$
    unfolding n_def
    by ($metis\ map\_nth$)
  obtain τ **where** τ_def: $ys = map\ \tau\ [0..<n]$
    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 ad_agr: $ad\_agr\_sets\ (\{..n\} - \{n\})\ (\{..n\} - \{n\})\ AD\ \sigma\ \tau$
    using $iffD2[OF\ ad\_agr\_list\_link,\ OF\ assms(1)[unfolded\ \sigma\_def\ \tau\_def]]$
    unfolding set_n **.**
  have set_ys: $\tau\ `\ (\{..n\} - \{n\}) = set\ ys$
    by ($auto\ simp$: τ_def)
  obtain z' **where** z'_def: $z' \in Inl\ `\ AD \cup Inr\ `\ \{..<Suc\ (card\ (Inr\ -`\ set\ ys))\} \cup set\ ys$
    $ad\_agr\_sets\ \{..n\}\ \{..n\}\ AD\ (\sigma(n := z))\ (\tau(n := z'))$
    using $extend\_\tau[OF\ ad\_agr\ subset\_refl,$
        $of\ Inl\ `\ AD \cup Inr\ `\ \{..<Suc\ (card\ (Inr\ -`\ set\ ys))\} \cup set\ ys\ z]$
    by ($auto\ simp$: set_ys)
  have map_take: $map\ (\sigma(n := z))\ ([0..<i]\ @\ n\ \#\ [i..<n]) = take\ i\ xs\ @\ z\ \#\ drop\ i\ xs$
    $map\ (\tau(n := z'))\ ([0..<i]\ @\ n\ \#\ [i..<n]) = take\ i\ ys\ @\ z'\ \#\ drop\ i\ ys$
    using i_le_n
    by ($auto\ simp$: $\sigma\_def\ \tau\_def\ take\_map\ drop\_map$)
  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 $fo\_nmlz\ AD\ zs = zs\ i \leq length\ zs$
  shows $\exists z' \in Inl\ `\ AD \cup Inr\ `\ \{..<Suc\ (card\ (Inr\ -`\ set\ zs))\}.$
    $fo\_nmlz\ AD\ (add\_nth\ i\ z\ zs) = fo\_nmlz\ AD\ (add\_nth\ i\ z'\ zs)$
proof −
  have $set\ zs \subseteq Inl\ `\ AD \cup Inr\ `\ \{..<Suc\ (card\ (Inr\ -`\ set\ zs))\}$
    using $fo\_nmlz\_set\_card[OF\ assms(1)]$
    by $auto$
  then obtain z' **where** z'_def:
    $z' \in Inl\ `\ AD \cup Inr\ `\ \{..<Suc\ (card\ (Inr\ -`\ set\ zs))\}$
    $ad\_agr\_list\ AD\ (take\ i\ zs\ @\ z\ \#\ drop\ i\ zs)\ (take\ i\ zs\ @\ z'\ \#\ drop\ i\ zs)$
    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 $i \leq length\ zs$
  shows $\exists z'.\ fo\_nmlz\ AD\ (add\_nth\ i\ z\ zs) = fo\_nmlz\ AD\ (add\_nth\ i\ z'\ (fo\_nmlz\ AD\ zs))$

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*: $i \leq$ *length* (*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 $i \leq$ *length zs*
  shows $\exists z'$. *fo_nmlz AD* (*add_nth i z* (*fo_nmlz AD zs*)) = *fo_nmlz AD* (*add_nth i z′ zs*)
proof −
  have *ad_agr*: *ad_agr_list AD* (*fo_nmlz AD zs*) *zs*
    using *ad_agr_list_comm*[*OF fo_nmlz_ad_agr*]
    by *auto*
  have *i_le_fo_nmlz*: $i \leq$ *length* (*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* (*fo_nmlz AD zs*)) (*add_nth i x zs*)
    using *ad_agr_list_add*[*OF ad_agr 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* < *length xs*
  shows $\exists z$. *fo_nmlz AD* (*add_nth i z* (*fo_nmlz AD* (*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*) $\Longrightarrow$
  $v \in$ *set xs* $\cup$ *set ys* $\vee$ $w \in$ *set xs* $\cup$ *set ys* $\Longrightarrow v = w$
  by (*auto simp*: *sp_equiv_list_def pairwise_def*) (*metis UnCI sp_equiv_pair.simps zip_same*)+

lemma *ad_agr_list_add_nth*:
  assumes $i \leq$ *length zs ad_agr_list AD* (*add_nth i v zs*) (*add_nth i w zs*) $v \neq w$
  shows $\{v, w\} \cap$ (*Inl* ‘ *AD* $\cup$ *set zs*) = {}
  using *assms*(*2*)[*unfolded add_nth_take_drop*[*OF assms*(*1*)]] *assms*(*1*,*3*) *sp_equiv_list_almost_same*
  by (*auto simp*: *ad_agr_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* < *card* (*Inr* −‘ *set zs*)
  shows *Inr n* $\in$ *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 card Z ≤ card {ts ∈ X. ∃ z ∈ Z. ts = f z}
  shows f ‘ Z ⊆ X
proof −
  have set_unfold: {ts ∈ X. ∃ z ∈ Z. ts = f z} = f ‘ Z ∩ 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 (⋀xs. xs ∈ X ⟹ fo_nmlz AD xs = xs) (⋀xs. xs ∈ X ⟹ i < length xs)
    fo_nmlz AD zs = zs i ≤ length zs finite AD finite X
  shows (∀ z. fo_nmlz AD (add_nth i z zs) ∈ X) ⟷
    Suc (card AD + card (Inr −‘ set zs)) ≤ card {ts ∈ X. ∃ z. ts = fo_nmlz AD (add_nth i z zs)}
proof −
  have inj: inj_on (λz. fo_nmlz AD (add_nth i z zs))
    (Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))})
    using ad_agr_list_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 ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}) =
      Suc (card AD + card (Inr −‘ set zs))
    using card_Un_disjoint[of Inl ‘ AD Inr ‘ {..<Suc (card (Inr −‘ set zs))}] assms(5)
    by (auto simp add: card_image disjoint_iff_not_equal)
  have restrict_z: (∀ z. fo_nmlz AD (add_nth i z zs) ∈ X) ⟷
    (∀ z ∈ Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}. fo_nmlz AD (add_nth i z zs) ∈ X)
    using add_nth_restrict[OF assms(3,4)]
    by metis
  have restrict_z′: {ts ∈ X. ∃ z. ts = fo_nmlz AD (add_nth i z zs)} =
    {ts ∈ X. ∃ z ∈ Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}.
      ts = fo_nmlz AD (add_nth i z zs)}
    using add_nth_restrict[OF assms(3,4)]
    by auto
  {
    assume ⋀z. fo_nmlz AD (add_nth i z zs) ∈ X
    then have image_sub: (λz. fo_nmlz AD (add_nth i z zs)) ‘
      (Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}) ⊆
      {ts ∈ X. ∃ z. ts = fo_nmlz AD (add_nth i z zs)}
      by auto
    have Suc (card AD + card (Inr −‘ set zs)) ≤
      card {ts ∈ X. ∃ z. ts = 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)) ≤
      card {ts ∈ X. ∃ z. ts = fo_nmlz AD (add_nth i z zs)}
      by (auto simp: card_image)
  }
  moreover
  {
    assume assm: card (Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}) ≤
      card {ts ∈ X. ∃ z ∈ Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}.
        ts = fo_nmlz AD (add_nth i z zs)}
    have ∀ z ∈ Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}. fo_nmlz AD (add_nth i z zs) ∈ 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 < length xs ⋀x. x ∈ X ⟹ fo_nmlz AD x = x
    ⋀x. x ∈ X ⟹ j < length x
  shows {ts ∈ X. ∃z. ts = fo_nmlz AD (add_nth j z (fo_nmlz AD (rem_nth j xs)))} =
  {y ∈ X. fo_nmlz AD (rem_nth j y) = fo_nmlz AD (rem_nth j xs)}
 using fo_nmlz_rem_nth_add_nth[where ?zs=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 φ :: ('a :: infinite, 'b) fo_fmla
  assumes wf: fo_wf φ I t
  shows fo_wf (Forall i φ) I (eval_forall i (fv_fo_fmla_list φ) t)
proof -
  obtain AD n X where t_def: t = (AD, n, X) AD = act_edom φ I
    AD = act_edom (Forall i φ) I
    using assms
    by (cases t) auto
  have AD_sub: act_edom φ I ⊆ 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 φ) t = eval_abs (Forall i φ) I
  proof (cases pos i (fv_fo_fmla_list φ))
    case None
    note fv_eq = fv_ex_all[OF None]
    have X = fo_nmlz AD ` proj_fmla (Forall i φ) {σ. esat φ I σ UNIV}
      unfolding X_def
      by (auto simp: proj_fmla_def fv_eq)
    also have ... = fo_nmlz AD ` proj_fmla (Forall i φ) {σ. esat (Forall i φ) I σ UNIV}
      using esat_forall_not_fv[of i φ 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 ∈ fv_fo_fmla φ
      by (rule pos_set[OF Some, unfolded fv_fo_fmla_list_set])
    have fo_nmlz_X: ⋀xs. xs ∈ X ⟹ fo_nmlz AD xs = xs
      by (auto simp: X_def proj_fmla_map fo_nmlz_idem[OF fo_nmlz_sound])
    have j_lt_len: ⋀xs. xs ∈ X ⟹ 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: ⋀xs. xs ∈ X ⟹ j ≤ 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 (λt Z. Suc (card AD + card (Inr -` set t)) ≤
```

```
card Z)
      (cluster (Some ∘ (λts. fo_nmlz AD (rem_nth j ts))) X)) =
      fo_nmlz AD ` proj_fmla (Forall i φ) {σ. esat (Forall i φ) I σ UNIV}
    proof (rule set_eqI, rule iffI)
      fix vs
      assume vs ∈ Mapping.keys (Mapping.filter (λt Z. Suc (card AD + card (Inr −` set t)) ≤ card Z)
        (cluster (Some ∘ (λts. fo_nmlz AD (rem_nth j ts))) X))
      then obtain ws where ws_def: ws ∈ X vs = fo_nmlz AD (rem_nth j ws)
        ⋀a. fo_nmlz AD (add_nth j a (fo_nmlz AD (rem_nth j ws))) ∈ 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 σ where σ_def:
        esat φ I σ UNIV ws = fo_nmlz AD (map σ (fv_fo_fmla_list φ))
        unfolding X_def
        by (auto simp: proj_fmla_map)
      obtain τ where τ_def: ws = map τ (fv_fo_fmla_list φ)
        using fo_nmlz_map σ_def(2)
        by blast
      have fo_nmlzd_τ: fo_nmlzd AD (map τ (fv_fo_fmla_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_fmla_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_fmla_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_fmla_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_fmla_list (Forall i φ))
        using rem_nth_sound[OF _ Some] sorted_distinct_fv_list
        by (auto simp: fv_fo_fmla_list_forall τ_def)
      then show vs ∈ fo_nmlz AD ` proj_fmla (Forall i φ) {σ. esat (Forall i φ) I σ UNIV}
        using ws_def(2) esat_τ
        by (auto simp: proj_fmla_map rem_nth_ws)
    next
      fix vs
      assume assm: vs ∈ fo_nmlz AD ` proj_fmla (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_fmla_list (Forall i φ)))
        esat (Forall i φ) I σ UNIV
        by (auto simp: proj_fmla_map)
      then have all_esat: ⋀x. esat φ I (σ(i := x)) UNIV
        by auto
      define ws where ws ≡ fo_nmlz AD (map σ (fv_fo_fmla_list φ))
      then have length ws = nfv φ
```

using *nfv_def fo_nmlz_length* **by** (*metis length_map*)
then have *ws_in*: $ws \in$ *fo_nmlz AD ‘ proj_fmla* φ $\{\sigma.$ *esat* φ *I* σ *UNIV*$\}$
using *all_esat*[*of* σ *i*] *ws_def*
by (*auto simp*: *fo_nmlz_sound proj_fmla_map*)
then have *ws_in_X*: $ws \in X$
by (*auto simp*: *X_def*)
obtain τ **where** τ*_def*: *ws* = *map* τ (*fv_fo_fmla_list* φ)
using *fo_nmlz_map ws_def*
by *blast*
have *rem_nth_ws*: *rem_nth j ws* = *map* τ (*fv_fo_fmla_list* (*Forall i* φ))
using *rem_nth_sound*[*of fv_fo_fmla_list* φ *i j*] *sorted_distinct_fv_list Some*
unfolding *fv_fo_fmla_list_forall* τ*_def*
by *auto*
have *set* (*fv_fo_fmla_list* (*Forall i* φ)) $\subseteq$ *set* (*fv_fo_fmla_list* φ)
by (*auto simp*: *fv_fo_fmla_list_forall*)
then have *ad_agr*: *ad_agr_list AD* (*map* σ (*fv_fo_fmla_list* (*Forall i* φ)))
(*map* τ (*fv_fo_fmla_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*: $\bigwedge x.$ *map* ($\sigma(i := x)$) (*fv_fo_fmla_list* (*Forall i* φ)) =
map σ (*fv_fo_fmla_list* (*Forall i* φ))
by (*auto simp*: *fv_fo_fmla_list_forall*)
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*] **.**
have $\bigwedge a.$ *fo_nmlz AD* (*add_nth j a* (*fo_nmlz AD* (*rem_nth j ws*))) $\in$
fo_nmlz AD ‘ proj_fmla φ $\{\sigma.$ *esat* φ *I* σ *UNIV*$\}$
proof −
fix *a*
obtain *x* **where** *add_rem*: *fo_nmlz AD* (*add_nth j a* (*fo_nmlz AD* (*rem_nth j ws*))) =
fo_nmlz AD (*map* ($\tau(i := x)$) (*fv_fo_fmla_list* φ))
using *add_nth_rem_nth_map*[*OF _ Some*, *of _* τ] *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*: τ*_def*)
have *esat* (*Forall i* φ) *I* τ *UNIV*
apply (*rule iffD1*[*OF esat_UNIV_ad_agr_list* σ*_def*(*2*), *OF _ subset_refl*, *folded t_def*])
using *fo_nmlz_ad_agr*[*of AD map* σ (*fv_fo_fmla_list* φ), *folded ws_def*, *unfolded* τ*_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* φ *I* ($\tau(i := x)$) *UNIV*
by *auto*
then show *fo_nmlz AD* (*add_nth j a* (*fo_nmlz AD* (*rem_nth j ws*))) $\in$
fo_nmlz AD ‘ proj_fmla φ $\{\sigma.$ *esat* φ *I* σ *UNIV*$\}$
by (*auto simp*: *add_rem proj_fmla_map*)
qed
then show *vs* $\in$ *Mapping.keys* (*Mapping.filter* (λt *Z*. *Suc* (*card AD* + *card* (*Inr* −‘ *set t*)) $\leq$ *card Z*)
(*cluster* (*Some* $\circ$ ($\lambda ts.$ *fo_nmlz AD* (*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 φ) 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 ⇒ '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 φ I t finite (fo_rep t)
  shows fo_res t = Fin (fo_rep t)
proof −
  obtain AD n X where t_def: t = (AD, n, X)
    using assms(1)
    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 φ I t ¬finite (fo_rep t)
  shows fo_res t = Infin
proof −
  obtain AD n X where t_def: t = (AD, n, X)
    using assms(1)
    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 φ I t ⟹ fo_rep t = proj_sat φ 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 ⇒ ('a table, 'b) fo_intp ⇒ ('a + nat) val ⇒ bool
where
  esat_UNIV φ I σ = esat φ I σ UNIV

lemma esat_UNIV_code[code]: esat_UNIV φ I σ ⟷ (if wf_fo_intp φ I then
  (case eval_fmla φ I of (AD, n, X) ⇒
    fo_nmlz (act_edom φ I) (map σ (fv_fo_fmla_list φ)) ∈ X)
  else esat_UNIV φ I σ)
proof −
  obtain AD n T where t_def: Ailamazyan.eval_fmla φ I = (AD, n, T)
    by (cases Ailamazyan.eval_fmla φ I) auto
  {
    assume wf_fo_intp: wf_fo_intp φ 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 φ I
      using fo_wf
      by auto
    have esat_UNIV φ I σ ⟷
      fo_nmlz (act_edom φ I) (map σ (fv_fo_fmla_list φ)) ∈ 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 φ :: ('a :: infinite, 'b) fo_fmla
  shows sat φ I σ ⟷ (if wf_fo_intp φ I then
  (case eval_fmla φ I of (AD, n, X) ⇒
    fo_nmlz (act_edom φ I) (map (Inl ∘ σ) (fv_fo_fmla_list φ)) ∈ X)
  else sat φ I σ)
  using esat_UNIV_code sat_esat_conv[folded esat_UNIV_def]
  by metis

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.