First-Order Query Evaluation

Martin Raszyk

March 17, 2025

Abstract

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

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

Contents

theory *FO*
  imports *Main*
begin

abbreviation *sorted_distinct xs* ≡ *sorted xs* ∧ *distinct xs*

datatype ′*a fo_term* = *Const* ′*a* | *Var nat*

type_synonym ′*a val* = *nat* ⇒ ′*a*

fun *list_fo_term* :: ′*a fo_term* ⇒ ′*a list* **where**
  list_fo_term (*Const c*) = [*c*]
| *list_fo_term* _ = []

fun *fv_fo_term_list* :: ′*a fo_term* ⇒ *nat list* **where**
  fv_fo_term_list (*Var n*) = [*n*]
| *fv_fo_term_list* _ = []

fun *fv_fo_term_set* :: *'a fo_term* ⇒ *nat set* **where**
fv_fo_term_set (*Var n*) = {*n*}
| *fv_fo_term_set* _ = {}

definition *fv_fo_terms_set* :: (*'a fo_term*) *list* ⇒ *nat set* **where**
fv_fo_terms_set ts = ⋃(*set* (*map fv_fo_term_set ts*))

fun *fv_fo_terms_list_rec* :: (*'a fo_term*) *list* ⇒ *nat list* **where**
fv_fo_terms_list_rec [] = []
| *fv_fo_terms_list_rec* (*t* # *ts*) = *fv_fo_term_list t* @ *fv_fo_terms_list_rec ts*

definition *fv_fo_terms_list* :: (*'a fo_term*) *list* ⇒ *nat list* **where**
fv_fo_terms_list ts = *remdups_adj* (*sort* (*fv_fo_terms_list_rec ts*))

fun *eval_term* :: *'a val* ⇒ *'a fo_term* ⇒ *'a* (**infix** ‹·› *60*) **where**
eval_term σ (*Const c*) = *c*
| *eval_term* σ (*Var n*) = σ *n*

definition *eval_terms* :: *'a val* ⇒ (*'a fo_term*) *list* ⇒ *'a list* (**infix** ‹⊙› *60*) **where**
eval_terms σ *ts* = *map* (*eval_term* σ) *ts*

lemma *finite_set_fo_term*: *finite* (*set_fo_term t*)
⟨*proof*⟩

lemma *list_fo_term_set*: *set* (*list_fo_term t*) = *set_fo_term t*
⟨*proof*⟩

lemma *finite_fv_fo_term_set*: *finite* (*fv_fo_term_set t*)
⟨*proof*⟩

lemma *fv_fo_term_setD*: *n* ∈ *fv_fo_term_set t* ⟹ *t* = *Var n*
⟨*proof*⟩

lemma *fv_fo_term_set_list*: *set* (*fv_fo_term_list t*) = *fv_fo_term_set t*
⟨*proof*⟩

lemma *sorted_distinct_fv_fo_term_list*: *sorted_distinct* (*fv_fo_term_list t*)
⟨*proof*⟩

lemma *fv_fo_term_set_cong*: *fv_fo_term_set t* = *fv_fo_term_set* (*map_fo_term f t*)
⟨*proof*⟩

lemma *fv_fo_terms_setI*: *Var m* ∈ *set ts* ⟹ *m* ∈ *fv_fo_terms_set ts*
⟨*proof*⟩

lemma *fv_fo_terms_setD*: *m* ∈ *fv_fo_terms_set ts* ⟹ *Var m* ∈ *set ts*
⟨*proof*⟩

lemma *finite_fv_fo_terms_set*: *finite* (*fv_fo_terms_set ts*)
⟨*proof*⟩

lemma *fv_fo_terms_set_list*: *set* (*fv_fo_terms_list ts*) = *fv_fo_terms_set ts*
⟨*proof*⟩

lemma *distinct_remdups_adj_sort*: *sorted xs* ⟹ *distinct* (*remdups_adj xs*)
⟨*proof*⟩

lemma *sorted_distinct_fv_fo_terms_list*: *sorted_distinct* (*fv_fo_terms_list ts*)

⟨proof⟩

lemma fv_fo_terms_set_cong: fv_fo_terms_set ts = fv_fo_terms_set (map (map_fo_term f) ts)
⟨proof⟩

lemma eval_term_cong: ($\bigwedge$n. n $\in$ fv_fo_term_set t $\Longrightarrow$ σ n = σ' n) $\Longrightarrow$
eval_term σ t = eval_term σ' t
⟨proof⟩

lemma eval_terms_fv_fo_terms_set: σ $\odot$ ts = σ' $\odot$ ts $\Longrightarrow$ n $\in$ fv_fo_terms_set ts $\Longrightarrow$ σ n = σ' n
⟨proof⟩

lemma eval_terms_cong: ($\bigwedge$n. n $\in$ fv_fo_terms_set ts $\Longrightarrow$ σ n = σ' n) $\Longrightarrow$
eval_terms σ ts = eval_terms σ' ts
⟨proof⟩

datatype ('a, 'b) fo_fmla =
Pred 'b ('a fo_term) list
| Bool bool
| Eqa 'a fo_term 'a fo_term
| Neg ('a, 'b) fo_fmla
| Conj ('a, 'b) fo_fmla ('a, 'b) fo_fmla
| Disj ('a, 'b) fo_fmla ('a, 'b) fo_fmla
| Exists nat ('a, 'b) fo_fmla
| Forall nat ('a, 'b) fo_fmla

fun fv_fo_fmla_list_rec :: ('a, 'b) fo_fmla $\Rightarrow$ nat list **where**
fv_fo_fmla_list_rec (Pred _ ts) = fv_fo_terms_list ts
| fv_fo_fmla_list_rec (Bool b) = []
| fv_fo_fmla_list_rec (Eqa t t') = fv_fo_term_list t @ fv_fo_term_list t'
| fv_fo_fmla_list_rec (Neg 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 $\neq$ m) (fv_fo_fmla_list_rec φ)
| fv_fo_fmla_list_rec (Forall n φ) = filter (λm. n $\neq$ m) (fv_fo_fmla_list_rec φ)

definition fv_fo_fmla_list :: ('a, 'b) fo_fmla $\Rightarrow$ 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 $\Rightarrow$ nat set **where**
fv_fo_fmla (Pred _ ts) = fv_fo_terms_set ts
| fv_fo_fmla (Bool b) = {}
| fv_fo_fmla (Eqa t t') = fv_fo_term_set t $\cup$ fv_fo_term_set t'
| fv_fo_fmla (Neg φ) = fv_fo_fmla φ
| fv_fo_fmla (Conj φ ψ) = fv_fo_fmla φ $\cup$ fv_fo_fmla ψ
| fv_fo_fmla (Disj φ ψ) = fv_fo_fmla φ $\cup$ 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)
⟨proof⟩

lemma fv_fo_fmla_list_set: set (fv_fo_fmla_list φ) = fv_fo_fmla φ
⟨proof⟩

lemma sorted_distinct_fv_list: sorted_distinct (fv_fo_fmla_list φ)
⟨proof⟩

lemma *length_fv_fo_fmla_list*: *length* (*fv_fo_fmla_list* φ) = *card* (*fv_fo_fmla* φ)
⟨*proof*⟩

lemma *fv_fo_fmla_list_eq*: *fv_fo_fmla* φ = *fv_fo_fmla* ψ $\Longrightarrow$ *fv_fo_fmla_list* φ = *fv_fo_fmla_list* ψ
⟨*proof*⟩

lemma *fv_fo_fmla_list_Conj*: *fv_fo_fmla_list* (*Conj* φ ψ) = *fv_fo_fmla_list* (*Conj* ψ φ)
⟨*proof*⟩

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*))
| *wf_fo_intp* (*Bool b*) *I* ⟷ *True*
| *wf_fo_intp* (*Eqa t t'*) *I* ⟷ *True*
| *wf_fo_intp* (*Neg* φ) *I* ⟷ *wf_fo_intp* φ *I*
| *wf_fo_intp* (*Conj* φ ψ) *I* ⟷ *wf_fo_intp* φ *I* ∧ *wf_fo_intp* ψ *I*
| *wf_fo_intp* (*Disj* φ ψ) *I* ⟷ *wf_fo_intp* φ *I* ∧ *wf_fo_intp* ψ *I*
| *wf_fo_intp* (*Exists n* φ) *I* ⟷ *wf_fo_intp* φ *I*
| *wf_fo_intp* (*Forall n* φ) *I* ⟷ *wf_fo_intp* φ *I*

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

lemma *sat_fv_cong*: (⋀*n*. *n* ∈ *fv_fo_fmla* φ $\Longrightarrow$ σ *n* = σ' *n*) $\Longrightarrow$
sat φ *I* σ ⟷ *sat* φ *I* σ'
⟨*proof*⟩

definition *proj_sat* :: (*'a*, *'b*) *fo_fmla* ⇒ (*'a table*, *'b*) *fo_intp* ⇒ *'a table* **where**
proj_sat φ *I* = ($\lambda\sigma$. *map* σ (*fv_fo_fmla_list* φ)) ‘ {σ. *sat* φ *I* σ}

end
theory *Eval_FO*
imports *HOL−Library.Infinite_Typeclass FO*
begin

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

locale *eval_fo* =
fixes *wf* :: (*'a* :: *infinite*, *'b*) *fo_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 $\mathit{eval\_disj} :: \mathit{nat\ list} \Rightarrow {'t} \Rightarrow \mathit{nat\ list} \Rightarrow {'t} \Rightarrow {'t}$
and $\mathit{eval\_exists} :: \mathit{nat} \Rightarrow \mathit{nat\ list} \Rightarrow {'t} \Rightarrow {'t}$
and $\mathit{eval\_forall} :: \mathit{nat} \Rightarrow \mathit{nat\ list} \Rightarrow {'t} \Rightarrow {'t}$
assumes $\mathit{fo\_rep}$: $\mathit{wf}\ \varphi\ I\ t \Longrightarrow \mathit{rep}\ t = \mathit{proj\_sat}\ \varphi\ I$
and $\mathit{fo\_res\_fin}$: $\mathit{wf}\ \varphi\ I\ t \Longrightarrow \mathit{finite}\ (\mathit{rep}\ t) \Longrightarrow \mathit{res}\ t = \mathit{Fin}\ (\mathit{rep}\ t)$
and $\mathit{fo\_res\_infin}$: $\mathit{wf}\ \varphi\ I\ t \Longrightarrow \neg \mathit{finite}\ (\mathit{rep}\ t) \Longrightarrow \mathit{res}\ t = \mathit{Infin}$
and $\mathit{fo\_abs}$: $\mathit{finite}\ (I\ (r,\ \mathit{length}\ \mathit{ts})) \Longrightarrow \mathit{wf}\ (\mathit{Pred}\ r\ \mathit{ts})\ I\ (\mathit{abs}\ \mathit{ts}\ (I\ (r,\ \mathit{length}\ \mathit{ts})))$
and $\mathit{fo\_bool}$: $\mathit{wf}\ (\mathit{Bool}\ b)\ I\ (\mathit{eval\_bool}\ b)$
and $\mathit{fo\_eq}$: $\mathit{wf}\ (\mathit{Eqa}\ \mathit{trm}\ \mathit{trm}')\ I\ (\mathit{eval\_eq}\ \mathit{trm}\ \mathit{trm}')$
and $\mathit{fo\_neg}$: $\mathit{wf}\ \varphi\ I\ t \Longrightarrow \mathit{wf}\ (\mathit{Neg}\ \varphi)\ I\ (\mathit{eval\_neg}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t)$
and $\mathit{fo\_conj}$: $\mathit{wf}\ \varphi\ I\ t\varphi \Longrightarrow \mathit{wf}\ \psi\ I\ t\psi \Longrightarrow (\mathit{case}\ \psi\ \mathit{of}\ \mathit{Neg}\ \psi' \Rightarrow \mathit{False} \mid \_ \Rightarrow \mathit{True}) \Longrightarrow$
$\mathit{wf}\ (\mathit{Conj}\ \varphi\ \psi)\ I\ (\mathit{eval\_conj}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t\varphi\ (\mathit{fv\_fo\_fmla\_list}\ \psi)\ t\psi)$
and $\mathit{fo\_ajoin}$: $\mathit{wf}\ \varphi\ I\ t\varphi \Longrightarrow \mathit{wf}\ \psi'\ I\ t\psi' \Longrightarrow$
$\mathit{wf}\ (\mathit{Conj}\ \varphi\ (\mathit{Neg}\ \psi'))\ I\ (\mathit{eval\_ajoin}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t\varphi\ (\mathit{fv\_fo\_fmla\_list}\ \psi')\ t\psi')$
and $\mathit{fo\_disj}$: $\mathit{wf}\ \varphi\ I\ t\varphi \Longrightarrow \mathit{wf}\ \psi\ I\ t\psi \Longrightarrow$
$\mathit{wf}\ (\mathit{Disj}\ \varphi\ \psi)\ I\ (\mathit{eval\_disj}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t\varphi\ (\mathit{fv\_fo\_fmla\_list}\ \psi)\ t\psi)$
and $\mathit{fo\_exists}$: $\mathit{wf}\ \varphi\ I\ t \Longrightarrow \mathit{wf}\ (\mathit{Exists}\ i\ \varphi)\ I\ (\mathit{eval\_exists}\ i\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t)$
and $\mathit{fo\_forall}$: $\mathit{wf}\ \varphi\ I\ t \Longrightarrow \mathit{wf}\ (\mathit{Forall}\ i\ \varphi)\ I\ (\mathit{eval\_forall}\ i\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t)$
begin

fun $\mathit{eval\_fmla} :: ('a,\ 'b)\ \mathit{fo\_fmla} \Rightarrow ('a\ \mathit{table},\ 'b)\ \mathit{fo\_intp} \Rightarrow {'t}$ **where**
$\mathit{eval\_fmla}\ (\mathit{Pred}\ r\ \mathit{ts})\ I = \mathit{abs}\ \mathit{ts}\ (I\ (r,\ \mathit{length}\ \mathit{ts}))$
$\mid \mathit{eval\_fmla}\ (\mathit{Bool}\ b)\ I = \mathit{eval\_bool}\ b$
$\mid \mathit{eval\_fmla}\ (\mathit{Eqa}\ t\ t')\ I = \mathit{eval\_eq}\ t\ t'$
$\mid \mathit{eval\_fmla}\ (\mathit{Neg}\ \varphi)\ I = \mathit{eval\_neg}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ (\mathit{eval\_fmla}\ \varphi\ I)$
$\mid \mathit{eval\_fmla}\ (\mathit{Conj}\ \varphi\ \psi)\ I = (\mathit{let}\ \mathit{ns}\varphi = \mathit{fv\_fo\_fmla\_list}\ \varphi;\ \mathit{ns}\psi = \mathit{fv\_fo\_fmla\_list}\ \psi;$
$X\varphi = \mathit{eval\_fmla}\ \varphi\ I\ \mathit{in}$
$\mathit{case}\ \psi\ \mathit{of}\ \mathit{Neg}\ \psi' \Rightarrow \mathit{let}\ X\psi' = \mathit{eval\_fmla}\ \psi'\ I\ \mathit{in}$
$\mathit{eval\_ajoin}\ \mathit{ns}\varphi\ X\varphi\ (\mathit{fv\_fo\_fmla\_list}\ \psi')\ X\psi'$
$\mid \_ \Rightarrow \mathit{eval\_conj}\ \mathit{ns}\varphi\ X\varphi\ \mathit{ns}\psi\ (\mathit{eval\_fmla}\ \psi\ I))$
$\mid \mathit{eval\_fmla}\ (\mathit{Disj}\ \varphi\ \psi)\ I = \mathit{eval\_disj}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ (\mathit{eval\_fmla}\ \varphi\ I)$
$(\mathit{fv\_fo\_fmla\_list}\ \psi)\ (\mathit{eval\_fmla}\ \psi\ I)$
$\mid \mathit{eval\_fmla}\ (\mathit{Exists}\ i\ \varphi)\ I = \mathit{eval\_exists}\ i\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ (\mathit{eval\_fmla}\ \varphi\ I)$
$\mid \mathit{eval\_fmla}\ (\mathit{Forall}\ i\ \varphi)\ I = \mathit{eval\_forall}\ i\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ (\mathit{eval\_fmla}\ \varphi\ I)$

lemma $\mathit{eval\_fmla\_correct}$:
fixes $\varphi :: ('a :: \mathit{infinite},\ 'b)\ \mathit{fo\_fmla}$
assumes $\mathit{wf\_fo\_intp}\ \varphi\ I$
shows $\mathit{wf}\ \varphi\ I\ (\mathit{eval\_fmla}\ \varphi\ I)$
$\langle \mathit{proof} \rangle$

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

lemma $\mathit{eval\_fmla\_proj\_sat}$:
fixes $\varphi :: ('a :: \mathit{infinite},\ 'b)\ \mathit{fo\_fmla}$
assumes $\mathit{wf\_fo\_intp}\ \varphi\ I$
shows $\mathit{rep}\ (\mathit{eval\_fmla}\ \varphi\ I) = \mathit{proj\_sat}\ \varphi\ I$
$\langle \mathit{proof} \rangle$

lemma $\mathit{eval\_sound}$:
fixes $\varphi :: ('a :: \mathit{infinite},\ 'b)\ \mathit{fo\_fmla}$
assumes $\mathit{eval}\ \varphi\ I = \mathit{Fin}\ Z$
shows $Z = \mathit{proj\_sat}\ \varphi\ I$
$\langle \mathit{proof} \rangle$

lemma $\mathit{eval\_complete}$:
fixes $\varphi :: ('a :: \mathit{infinite},\ 'b)\ \mathit{fo\_fmla}$

assumes *eval* φ *I* = *Infin*
shows *infinite* (*proj_sat* φ *I*)
⟨*proof*⟩

end

end
theory *Mapping_Code*
imports *Containers.Mapping_Impl*
begin

lift_definition *set_of_idx* :: (*'a*, *'b set*) *mapping* ⇒ *'b set* **is**
λ*m*. ⋃(*ran m*) ⟨*proof*⟩

lemma *set_of_idx_code*[*code*]:
fixes *t* :: (*'a* :: *ccompare*, *'b set*) *mapping_rbt*
shows *set_of_idx* (*RBT_Mapping t*) =
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''set_of_idx RBT_Mapping*: *ccompare* = *None''*) (λ_. *set_of_idx* (*RBT_Mapping t*))
| *Some* _ ⇒ ⋃(*snd* ‘ *set* (*RBT_Mapping2.entries t*)))
⟨*proof*⟩

lemma *mapping_combine*[*code*]:
fixes *t* :: (*'a* :: *ccompare*, *'b*) *mapping_rbt*
shows *Mapping.combine f* (*RBT_Mapping t*) (*RBT_Mapping u*) =
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *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*))
⟨*proof*⟩

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'*)) ⟨*proof*⟩

lemma *mapping_join_code*[*code*]:
fixes *t* :: (*'a* :: *ccompare*, *'b*) *mapping_rbt*
shows *mapping_join f* (*RBT_Mapping t*) (*RBT_Mapping u*) =
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*))

$\langle proof \rangle$

end

lift_definition *mapping_antijoin* :: $('a, 'b)$ *mapping* $\Rightarrow$ $('a, 'b)$ *mapping* $\Rightarrow$ $('a, 'b)$ *mapping* **is**
$\lambda m\ m'\ x.$ *case m x of None* $\Rightarrow$ *None* | *Some y* $\Rightarrow$ (*case m' x of None* $\Rightarrow$ *Some y* | *Some y'* $\Rightarrow$ *None*)
$\langle proof \rangle$

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* $\Rightarrow$ *Code.abort* (*STR ''mapping_antijoin RBT_Mapping: ccompare = None''*) (λ_. *mapping_antijoin* (*RBT_Mapping t*) (*RBT_Mapping u*))
| *Some* _ $\Rightarrow$ *RBT_Mapping* (*diff t u*))
$\langle proof \rangle$

end
theory *Cluster*
imports *Mapping_Code*
begin

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

lemma *these_insert*[*simp*]: *Option.these* (*insert x A*) = (*case x of Some a* $\Rightarrow$ *insert a* | *None* $\Rightarrow$ *id*) (*Option.these A*)
$\langle proof \rangle$

lemma *these_image_Un*[*simp*]: *Option.these* (f ‘ $(A \cup B)$) = *Option.these* (f ‘ A) $\cup$ *Option.these* (f ‘ B)
$\langle proof \rangle$

lemma *these_imageI*: $f\ x$ = *Some y* $\Longrightarrow x \in X \Longrightarrow y \in$ *Option.these* (f ‘ X)
$\langle proof \rangle$

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

lemma *set_of_idx_cluster*: *set_of_idx* (*cluster* (*Some* $\circ$ f) X) = X
$\langle proof \rangle$

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

context *ord*
begin

definition *add_to_rbt* :: $'a \times 'b \Rightarrow ('a, 'b$ *set*$)$ *rbt* $\Rightarrow ('a, 'b$ *set*$)$ *rbt* **where**
add_to_rbt = ($\lambda(a, b)\ t.$ *case rbt_lookup t a of Some X* $\Rightarrow$ *rbt_insert a* (*insert b X*) *t* | *None* $\Rightarrow$ *rbt_insert a* $\{b\}$ *t*)

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

definition *cluster_rbt* :: $('b \Rightarrow 'a$ *option*$) \Rightarrow ('b$, *unit*$)$ *rbt* $\Rightarrow ('a, 'b$ *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* $\Longrightarrow$ *is_rbt* (*add_to_rbt ab t*)
⟨*proof*⟩

lemma *is_rbt_fold_add_to_rbt*: *is_rbt t'* $\Longrightarrow$
is_rbt (*RBT_Impl.fold* (*add_option_to_rbt f*) *t t'*)
⟨*proof*⟩

lemma *is_rbt_cluster_rbt*: *is_rbt* (*cluster_rbt f t*)
⟨*proof*⟩

lemma *rbt_insert_entries_None*: *is_rbt t* $\Longrightarrow$ *rbt_lookup t k* = *None* $\Longrightarrow$
set (*RBT_Impl.entries* (*rbt_insert k v t*)) = *insert* (*k*, *v*) (*set* (*RBT_Impl.entries t*))
⟨*proof*⟩

lemma *rbt_insert_entries_Some*: *is_rbt t* $\Longrightarrow$ *rbt_lookup t k* = *Some v'* $\Longrightarrow$
set (*RBT_Impl.entries* (*rbt_insert k v t*)) = *insert* (*k*, *v*) (*set* (*RBT_Impl.entries t*) − {(*k*, *v'*)})
⟨*proof*⟩

lemma *keys_add_to_rbt*: *is_rbt t* $\Longrightarrow$ *set* (*RBT_Impl.keys* (*add_to_rbt* (*a*, *b*) *t*)) = *insert a* (*set* (*RBT_Impl.keys t*))
⟨*proof*⟩

lemma *keys_fold_add_to_rbt*: *is_rbt t'* $\Longrightarrow$ *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*⟩

lemma *rbt_lookup_add_to_rbt*: *is_rbt t* $\Longrightarrow$ *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*)
⟨*proof*⟩

lemma *rbt_lookup_fold_add_to_rbt*: *is_rbt t'* $\Longrightarrow$ *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*⟩

end

context
fixes *c* :: *'a comparator*
begin

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

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

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

context

assumes *c*: *comparator c*
begin

lemma *add_to_rbt_comp*: *add_to_rbt_comp* = *ord.add_to_rbt* (*lt_of_comp c*)
⟨*proof*⟩

lemma *cluster_rbt_comp*: *cluster_rbt_comp* = *ord.cluster_rbt* (*lt_of_comp c*)
⟨*proof*⟩

end

end

lift_definition *mapping_of_cluster* :: ($'b \Rightarrow 'a$ :: *ccompare option*) ⇒ ($'b$, *unit*) *rbt* ⇒ ($'a$, $'b$ *set*) *mapping_rbt* **is**
cluster_rbt_comp ccomp
⟨*proof*⟩

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

end
theory *Ailamazyan*
imports *Eval_FO Cluster Mapping_Code*
begin

fun *SP* :: ($'a$, $'b$) *fo_fmla* ⇒ *nat set* **where**
SP (*Eqa* (*Var n*) (*Var n'*)) = (*if* $n \neq n'$ *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*
⟨*proof*⟩

lemma *finite_SP*: *finite* (*SP φ*)
⟨*proof*⟩

fun *SP_list_rec* :: ($'a$, $'b$) *fo_fmla* ⇒ *nat list* **where**
SP_list_rec (*Eqa* (*Var n*) (*Var n'*)) = (*if* $n \neq n'$ *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*. $n \neq m$) (*SP_list_rec φ*)
| *SP_list_rec* (*Forall n φ*) = *filter* (*λm*. $n \neq m$) (*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*: $set\ (SP\_list\ \varphi) = SP\ \varphi$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *eval_eterms_fv_fo_terms_set*: $\sigma \odot e\ ts = \sigma' \odot e\ ts \Longrightarrow n \in fv\_fo\_terms\_set\ ts \Longrightarrow \sigma\ n = \sigma'\ n$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *ad_equiv_pair_refl*[*simp*]: *ad_equiv_pair* X $(a,\ a)$
⟨*proof*⟩

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

lemma *ad_equiv_pair_comm*: *ad_equiv_pair* X $(a,\ a') \longleftrightarrow$ *ad_equiv_pair* X $(a',\ a)$
⟨*proof*⟩

lemma *ad_equiv_pair_mono*: $X \subseteq Y \Longrightarrow$ *ad_equiv_pair* Y $(a,\ a') \Longrightarrow$ *ad_equiv_pair* X $(a,\ a')$
⟨*proof*⟩

lemma *sp_equiv_pair_comm*: *sp_equiv_pair* x $y \longleftrightarrow$ *sp_equiv_pair* y x
⟨*proof*⟩

definition *sp_equiv* :: $('a + 'c)$ *val* $\Rightarrow$ $('a + 'c)$ *val* $\Rightarrow$ *nat set* $\Rightarrow$ *bool* **where**
sp_equiv 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
⟨*proof*⟩

definition *ad_agr_sets* :: *nat set* $\Rightarrow$ *nat set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $('a + 'c)$ *val* $\Rightarrow$
$('a + 'c)$ *val* $\Rightarrow$ *bool* **where**
ad_agr_sets FV S X 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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *ad_equiv_list_comm*: *ad_equiv_list* X *xs ys* $\Longrightarrow$ *ad_equiv_list* X *ys xs*
⟨*proof*⟩

lemma *ad_equiv_list_mono*: $X \subseteq Y \Longrightarrow$ *ad_equiv_list* Y *xs ys* $\Longrightarrow$ *ad_equiv_list* X *xs ys*
⟨*proof*⟩

lemma *ad_equiv_list_trans*:
assumes *ad_equiv_list* X *xs ys* *ad_equiv_list* X *ys zs*
shows *ad_equiv_list* X *xs zs*
⟨*proof*⟩

lemma *ad_equiv_list_link*: $(\forall\, i \in$ *set ns*. *ad_equiv_pair* X $(\sigma\ i,\ \tau\ i)) \longleftrightarrow$
ad_equiv_list X $($*map* σ *ns*$)$ $($*map* τ *ns*$)$
⟨*proof*⟩

lemma *set_zip_comm*: $(x,\ y) \in$ *set* $($*zip xs ys*$) \Longrightarrow (y,\ x) \in$ *set* $($*zip ys xs*$)$
⟨*proof*⟩

lemma *set_zip_map*: *set* $($*zip* $($*map* σ *ns*$)$ $($*map* τ *ns*$)) = (\lambda n.\ (\sigma\ n,\ \tau\ n))$ ‘ *set ns*
⟨*proof*⟩

lemma *sp_equiv_list_comm*: *sp_equiv_list xs ys* $\Longrightarrow$ *sp_equiv_list ys xs*
⟨*proof*⟩

lemma *sp_equiv_list_trans*:
assumes *sp_equiv_list xs ys sp_equiv_list ys zs*
shows *sp_equiv_list xs zs*
⟨*proof*⟩

lemma *sp_equiv_list_link*: $sp\_equiv\_list\ (map\ \sigma\ ns)\ (map\ \tau\ ns) \longleftrightarrow sp\_equiv\ \sigma\ \tau\ (set\ ns)$
⟨*proof*⟩

lemma *ad_agr_list_comm*: $ad\_agr\_list\ X\ xs\ ys \Longrightarrow ad\_agr\_list\ X\ ys\ xs$
⟨*proof*⟩

lemma *ad_agr_list_mono*: $X \subseteq Y \Longrightarrow ad\_agr\_list\ Y\ ys\ xs \Longrightarrow ad\_agr\_list\ X\ ys\ xs$
⟨*proof*⟩

lemma *ad_agr_list_rev_mono*:
assumes $Y \subseteq X$ *ad_agr_list Y ys xs* $Inl\ -`\ set\ xs \subseteq Y$ $Inl\ -`\ set\ ys \subseteq Y$
shows *ad_agr_list X ys xs*
⟨*proof*⟩

lemma *ad_agr_list_trans*: $ad\_agr\_list\ X\ xs\ ys \Longrightarrow ad\_agr\_list\ X\ ys\ zs \Longrightarrow ad\_agr\_list\ X\ xs\ zs$
⟨*proof*⟩

lemma *ad_agr_list_refl*: *ad_agr_list X xs xs*
⟨*proof*⟩

lemma *ad_agr_list_set*: $ad\_agr\_list\ X\ xs\ ys \Longrightarrow y \in X \Longrightarrow Inl\ y \in set\ ys \Longrightarrow Inl\ y \in set\ xs$
⟨*proof*⟩

lemma *ad_agr_list_length*: $ad\_agr\_list\ X\ xs\ ys \Longrightarrow length\ xs = length\ ys$
⟨*proof*⟩

lemma *ad_agr_list_eq*: $set\ ys \subseteq AD \Longrightarrow ad\_agr\_list\ AD\ (map\ Inl\ xs)\ (map\ Inl\ ys) \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *sp_equiv_list_subset*:
assumes $set\ ms \subseteq set\ ns$ $sp\_equiv\_list\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns)$
shows $sp\_equiv\_list\ (map\ \sigma\ ms)\ (map\ \sigma'\ ms)$
⟨*proof*⟩

lemma *ad_agr_list_subset*: $set\ ms \subseteq set\ ns \Longrightarrow ad\_agr\_list\ X\ (map\ \sigma\ ns)\ (map\ \sigma'\ ns) \Longrightarrow$
$ad\_agr\_list\ X\ (map\ \sigma\ ms)\ (map\ \sigma'\ ms)$
⟨*proof*⟩

lemma *ad_agr_list_link*: $ad\_agr\_sets\ (set\ ns)\ (set\ ns)\ AD\ \sigma\ \tau \longleftrightarrow$
$ad\_agr\_list\ AD\ (map\ \sigma\ ns)\ (map\ \tau\ ns)$
⟨*proof*⟩

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

lemma *ad_agr_sets_restrict*:
$ad\_agr\_sets\ (set\ (fv\_fo\_fmla\_list\ \varphi))\ (set\ (fv\_fo\_fmla\_list\ \varphi))\ AD\ \sigma\ \tau \Longrightarrow ad\_agr\ \varphi\ AD\ \sigma\ \tau$
⟨*proof*⟩

lemma *finite_Inl*: $finite\ X \Longrightarrow finite\ (Inl\ -`\ X)$
⟨*proof*⟩

lemma *ex_out*:

assumes $\mathit{finite}\ X$
shows $\exists k.\ k \notin X \wedge k < \mathit{Suc}\ (\mathit{card}\ X)$
$\langle \mathit{proof} \rangle$

lemma $\mathit{extend\_\tau}$:
assumes $\mathit{ad\_agr\_sets}\ (FV - \{n\})\ (S - \{n\})\ X\ \sigma\ \tau\ S \subseteq FV\ \mathit{finite}\ S\ \tau\ `\ (FV - \{n\}) \subseteq Z$
$\mathit{Inl}\ `\ X \cup \mathit{Inr}\ `\ \{..<\mathit{max}\ 1\ (\mathit{card}\ (\mathit{Inr}\ -`\ \tau\ `\ (S - \{n\})) + (\mathit{if}\ n \in S\ \mathit{then}\ 1\ \mathit{else}\ 0))\} \subseteq Z$
shows $\exists k \in Z.\ \mathit{ad\_agr\_sets}\ FV\ S\ X\ (\sigma(n := x))\ (\tau(n := k))$
$\langle \mathit{proof} \rangle$

lemma $\mathit{esat\_Pred}$:
assumes $\mathit{ad\_agr\_sets}\ FV\ S\ (\bigcup(\mathit{set}\ `\ X))\ \sigma\ \tau\ \mathit{fv\_fo\_terms\_set}\ ts \subseteq FV\ \sigma \odot e\ ts \in \mathit{map}\ \mathit{Inl}\ `\ X$
$t \in \mathit{set}\ ts$
shows $\sigma \cdot e\ t = \tau \cdot e\ t$
$\langle \mathit{proof} \rangle$

lemma $\mathit{sp\_equiv\_list\_fv}$:
assumes $(\bigwedge i.\ i \in \mathit{fv\_fo\_terms\_set}\ ts \Longrightarrow \mathit{ad\_equiv\_pair}\ X\ (\sigma\ i,\ \tau\ i))$
$\bigcup(\mathit{set\_fo\_term}\ `\ \mathit{set}\ ts) \subseteq X\ \mathit{sp\_equiv}\ \sigma\ \tau\ (\mathit{fv\_fo\_terms\_set}\ ts)$
shows $\mathit{sp\_equiv\_list}\ (\mathit{map}\ ((\cdot e)\ \sigma)\ ts)\ (\mathit{map}\ ((\cdot e)\ \tau)\ ts)$
$\langle \mathit{proof} \rangle$

lemma $\mathit{esat\_Pred\_inf}$:
assumes $\mathit{fv\_fo\_terms\_set}\ ts \subseteq FV\ \mathit{fv\_fo\_terms\_set}\ ts \subseteq S$
$\mathit{ad\_agr\_sets}\ FV\ S\ AD\ \sigma\ \tau\ \mathit{ad\_agr\_list}\ AD\ (\sigma \odot e\ ts)\ vs$
$\bigcup(\mathit{set\_fo\_term}\ `\ \mathit{set}\ ts) \subseteq AD$
shows $\mathit{ad\_agr\_list}\ AD\ (\tau \odot e\ ts)\ vs$
$\langle \mathit{proof} \rangle$

type_synonym $('a,\ 'c)\ \mathit{fo\_t} = 'a\ \mathit{set} \times \mathit{nat} \times ('a + 'c)\ \mathit{table}$

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

fun $\mathit{sz\_fmla} :: ('a,\ 'b)\ \mathit{fo\_fmla} \Rightarrow \mathit{nat}$ **where**
$\mathit{sz\_fmla}\ (\mathit{Neg}\ \varphi) = \mathit{Suc}\ (\mathit{sz\_fmla}\ \varphi)$
$|\ \mathit{sz\_fmla}\ (\mathit{Conj}\ \varphi\ \psi) = \mathit{Suc}\ (\mathit{sz\_fmla}\ \varphi + \mathit{sz\_fmla}\ \psi)$
$|\ \mathit{sz\_fmla}\ (\mathit{Disj}\ \varphi\ \psi) = \mathit{Suc}\ (\mathit{sz\_fmla}\ \varphi + \mathit{sz\_fmla}\ \psi)$
$|\ \mathit{sz\_fmla}\ (\mathit{Exists}\ n\ \varphi) = \mathit{Suc}\ (\mathit{sz\_fmla}\ \varphi)$
$|\ \mathit{sz\_fmla}\ (\mathit{Forall}\ n\ \varphi) = \mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{sz\_fmla}\ \varphi))))$
$|\ \mathit{sz\_fmla}\ \_ = 0$

lemma $\mathit{sz\_fmla\_induct}[\mathit{case\_names}\ \mathit{Pred}\ \mathit{Bool}\ \mathit{Eqa}\ \mathit{Neg}\ \mathit{Conj}\ \mathit{Disj}\ \mathit{Exists}\ \mathit{Forall}]$:
$(\bigwedge r\ ts.\ P\ (\mathit{Pred}\ r\ ts)) \Longrightarrow (\bigwedge b.\ P\ (\mathit{Bool}\ b)) \Longrightarrow$
$(\bigwedge t\ t'.\ P\ (\mathit{Eqa}\ t\ t')) \Longrightarrow (\bigwedge \varphi.\ P\ \varphi \Longrightarrow P\ (\mathit{Neg}\ \varphi)) \Longrightarrow$
$(\bigwedge \varphi\ \psi.\ P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (\mathit{Conj}\ \varphi\ \psi)) \Longrightarrow (\bigwedge \varphi\ \psi.\ P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (\mathit{Disj}\ \varphi\ \psi)) \Longrightarrow$
$(\bigwedge n\ \varphi.\ P\ \varphi \Longrightarrow P\ (\mathit{Exists}\ n\ \varphi)) \Longrightarrow (\bigwedge n\ \varphi.\ P\ (\mathit{Exists}\ n\ (\mathit{Neg}\ \varphi)) \Longrightarrow P\ (\mathit{Forall}\ n\ \varphi)) \Longrightarrow P\ \varphi$
$\langle \mathit{proof} \rangle$

lemma $\mathit{esat\_fv\_cong}$: $(\bigwedge n.\ n \in \mathit{fv\_fo\_fmla}\ \varphi \Longrightarrow \sigma\ n = \sigma'\ n) \Longrightarrow \mathit{esat}\ \varphi\ I\ \sigma\ X \longleftrightarrow \mathit{esat}\ \varphi\ I\ \sigma'\ X$
$\langle \mathit{proof} \rangle$

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

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

lemma *finite_act_edom*: $wf\_fo\_intp\ \varphi\ I \Longrightarrow finite\ (act\_edom\ \varphi\ I)$
⟨*proof*⟩

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

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

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

lemma *esat_UNIV_cong*:
fixes $\sigma :: nat \Rightarrow 'a + nat$
assumes $ad\_agr\ \varphi\ AD\ \sigma\ \tau\ act\_edom\ \varphi\ I \subseteq AD$
shows $esat\ \varphi\ I\ \sigma\ UNIV \longleftrightarrow esat\ \varphi\ I\ \tau\ UNIV$
⟨*proof*⟩

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

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

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

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

$(map\ (Inl \circ \tau)\ (fv\_fo\_fmla\_list\ \varphi))\ act\_edom\ \varphi\ I \subseteq AD$
shows $sat\ \varphi\ I\ \sigma \longleftrightarrow sat\ \varphi\ I\ \tau$
⟨*proof*⟩

definition $nfv :: ('a, 'b)\ fo\_fmla \Rightarrow nat$ **where**
$nfv\ \varphi = length\ (fv\_fo\_fmla\_list\ \varphi)$

lemma nfv_card: $nfv\ \varphi = card\ (fv\_fo\_fmla\ \varphi)$
⟨*proof*⟩

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

lemma $filter\_rremdups\_filter$: $filter\ P\ (rremdups\ (filter\ Q\ xs)) =$
$rremdups\ (filter\ (\lambda x.\ P\ x \wedge Q\ x)\ xs)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

definition $fo\_nmlzd :: 'a\ set \Rightarrow ('a + nat)\ list \Rightarrow bool$ **where**
$fo\_nmlzd\ AD\ xs \longleftrightarrow Inl\ -`\ set\ xs \subseteq AD\ \wedge$
$(let\ ns = List.map\_filter\ (case\_sum\ Map.empty\ Some)\ xs\ in\ nats\ (rremdups\ ns))$

lemma $fo\_nmlzd\_all\_AD$:
assumes $set\ xs \subseteq Inl\ `\ AD$
shows $fo\_nmlzd\ AD\ xs$
⟨*proof*⟩

lemma $card\_Inr\_vimage\_le\_length$: $card\ (Inr\ -`\ set\ xs) \leq length\ xs$
⟨*proof*⟩

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*⟩

lemma *map_filter_take*: *∃ j. List.map_filter f (take i xs) = take j (List.map_filter f xs)*
⟨*proof*⟩

lemma *fo_nmlzd_take*: **assumes** *fo_nmlzd AD xs*
shows *fo_nmlzd AD (take i xs)*
⟨*proof*⟩

lemma *map_filter_app*: *List.map_filter f (xs @ [x]) = List.map_filter f xs @*
(case f x of Some y ⇒ [y] | _ ⇒ [])
⟨*proof*⟩

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′
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *nall_tuples_finite*: *finite AD ⟹ finite (nall_tuples AD n)*
⟨*proof*⟩

lemma *all_tuplesI*: *length vs = n ⟹ set vs ⊆ xs ⟹ vs ∈ all_tuples xs n*
⟨*proof*⟩

lemma *nall_tuplesI*: *length vs = n ⟹ fo_nmlzd AD vs ⟹ vs ∈ nall_tuples AD n*
⟨*proof*⟩

lemma *all_tuplesD*: *vs ∈ all_tuples xs n ⟹ length vs = n ∧ set vs ⊆ xs*
⟨*proof*⟩

lemma *all_tuples_setD*: *vs ∈ all_tuples xs n ⟹ set vs ⊆ xs*
⟨*proof*⟩

lemma *nall_tuplesD*: *vs ∈ nall_tuples AD n ⟹*
length vs = n ∧ set vs ⊆ Inl ‘ AD ∪ Inr ‘ {..<n} ∧ fo_nmlzd AD vs
⟨*proof*⟩

lemma *all_tuples_set*: *all_tuples xs n = {ys. length ys = n ∧ set ys ⊆ xs}*
⟨*proof*⟩

lemma *nall_tuples_set*: *nall_tuples AD n = {ys. length ys = n ∧ fo_nmlzd AD ys}*
⟨*proof*⟩

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*: $pos\ a\ xs = Some\ i \Longrightarrow a \in set\ xs$
  ⟨*proof*⟩

lemma *pos_length*: $pos\ a\ xs = Some\ i \Longrightarrow i < length\ xs$
  ⟨*proof*⟩

lemma *pos_sound*: $pos\ a\ xs = Some\ i \Longrightarrow i < length\ xs \wedge xs\ !\ i = a$
  ⟨*proof*⟩

lemma *pos_complete*: $pos\ a\ xs = None \Longrightarrow a \notin set\ xs$
  ⟨*proof*⟩

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

lemma *rem_nth_length*: $i < length\ xs \Longrightarrow length\ (rem\_nth\ i\ xs) = length\ xs - 1$
  ⟨*proof*⟩

lemma *rem_nth_take_drop*: $i < length\ xs \Longrightarrow rem\_nth\ i\ xs = take\ i\ xs\ @\ drop\ (Suc\ i)\ xs$
  ⟨*proof*⟩

lemma *rem_nth_sound*: $distinct\ xs \Longrightarrow pos\ n\ xs = Some\ i \Longrightarrow$
  $rem\_nth\ i\ (map\ \sigma\ xs) = map\ \sigma\ (filter\ ((\neq)\ n)\ xs)$
  ⟨*proof*⟩

fun *add_nth* :: $nat \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **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*: $i \leq length\ zs \Longrightarrow length\ (add\_nth\ i\ z\ zs) = Suc\ (length\ zs)$
  ⟨*proof*⟩

lemma *add_nth_take_drop*: $i \leq length\ zs \Longrightarrow add\_nth\ i\ v\ zs = take\ i\ zs\ @\ v\ \#\ drop\ i\ zs$
  ⟨*proof*⟩

lemma *add_nth_rem_nth_map*: $distinct\ xs \Longrightarrow pos\ n\ xs = Some\ i \Longrightarrow$
  $add\_nth\ i\ a\ (rem\_nth\ i\ (map\ \sigma\ xs)) = map\ (\sigma(n := a))\ xs$
  ⟨*proof*⟩

lemma *add_nth_rem_nth_self*: $i < length\ xs \Longrightarrow add\_nth\ i\ (xs\ !\ i)\ (rem\_nth\ i\ xs) = xs$
  ⟨*proof*⟩

lemma *rem_nth_add_nth*: $i \leq length\ zs \Longrightarrow rem\_nth\ i\ (add\_nth\ i\ z\ zs) = zs$
  ⟨*proof*⟩

fun *merge* :: $(nat \times 'a)\ list \Rightarrow (nat \times 'a)\ list \Rightarrow (nat \times 'a)\ list$ **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*
  ⟨*proof*⟩

lemma *merge_length*: $length\ (merge\ nxs\ mys) = length\ (map\ fst\ nxs\ @\ map\ fst\ mys)$
⟨*proof*⟩

lemma *insort_aux_le*: $\forall x{\in}set\ nxs.\ n \le fst\ x \Longrightarrow \forall x{\in}set\ mys.\ m \le fst\ x \Longrightarrow n \le m \Longrightarrow$
$insort\ n\ (sort\ (map\ fst\ nxs\ @\ m\ \#\ map\ fst\ mys)) = n\ \#\ sort\ (map\ fst\ nxs\ @\ m\ \#\ map\ fst\ mys)$
⟨*proof*⟩

lemma *insort_aux_gt*: $\forall x{\in}set\ nxs.\ n \le fst\ x \Longrightarrow \forall x{\in}set\ mys.\ m \le fst\ x \Longrightarrow \neg\ n \le m \Longrightarrow$
$insort\ n\ (sort\ (map\ fst\ nxs\ @\ m\ \#\ map\ fst\ mys)) =$
$m\ \#\ insort\ n\ (sort\ (map\ fst\ nxs\ @\ map\ fst\ mys))$
⟨*proof*⟩

lemma *map_fst_merge*: $sorted\_distinct\ (map\ fst\ nxs) \Longrightarrow sorted\_distinct\ (map\ fst\ mys) \Longrightarrow$
$map\ fst\ (merge\ nxs\ mys) = sort\ (map\ fst\ nxs\ @\ map\ fst\ mys)$
⟨*proof*⟩

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

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

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

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

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

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

lemma *fo_nmlz_rec_length*: $length\ (fo\_nmlz\_rec\ i\ m\ AD\ xs) = length\ xs$
⟨*proof*⟩

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

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

$\langle proof \rangle$

lemma *fo_nmlz_rec_set_rev*: $set\ (fo\_nmlz\_rec\ i\ m\ AD\ xs) \subseteq Inl\ `\ AD \Longrightarrow set\ xs \subseteq Inl\ `\ AD$
$\langle proof \rangle$

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

lemma *ad_agr_map*:
assumes $length\ xs = length\ ys\ inj\_on\ m\ (dom\ m)$
$\bigwedge x\ y.\ (x, y) \in set\ (zip\ xs\ ys) \Longrightarrow (case\ x\ of\ Inl\ x' \Rightarrow$
$if\ x' \in AD\ then\ x = y\ else\ m\ x = Some\ y \land (case\ y\ of\ Inl\ z \Rightarrow z \notin AD \mid Inr\ \_ \Rightarrow True)$
$\mid Inr\ n \Rightarrow m\ x = Some\ y \land (case\ y\ of\ Inl\ z \Rightarrow z \notin AD \mid Inr\ \_ \Rightarrow True))$
shows $ad\_agr\_list\ AD\ xs\ ys$
$\langle proof \rangle$

lemma *fo_nmlz_rec_take*: $take\ n\ (fo\_nmlz\_rec\ i\ m\ AD\ xs) = fo\_nmlz\_rec\ i\ m\ AD\ (take\ n\ xs)$
$\langle proof \rangle$

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

lemma *fo_nmlz_Nil*[*simp*]: $fo\_nmlz\ AD\ [] = []$
$\langle proof \rangle$

lemma *fo_nmlz_Cons*: $fo\_nmlz\ AD\ [x] =$
$(case\ x\ of\ Inl\ x \Rightarrow if\ x \in AD\ then\ [Inl\ x]\ else\ [Inr\ 0] \mid \_ \Rightarrow [Inr\ 0])$
$\langle proof \rangle$

lemma *fo_nmlz_Cons_Cons*: $fo\_nmlz\ AD\ [x, x] =$
$(case\ x\ of\ Inl\ x \Rightarrow if\ x \in AD\ then\ [Inl\ x, Inl\ x]\ else\ [Inr\ 0, Inr\ 0] \mid \_ \Rightarrow [Inr\ 0, Inr\ 0])$
$\langle proof \rangle$

lemma *fo_nmlz_sound*: $fo\_nmlzd\ AD\ (fo\_nmlz\ AD\ xs)$
$\langle proof \rangle$

lemma *fo_nmlz_length*: $length\ (fo\_nmlz\ AD\ xs) = length\ xs$
$\langle proof \rangle$

lemma *fo_nmlz_map*: $\exists \tau.\ fo\_nmlz\ AD\ (map\ \sigma\ ns) = map\ \tau\ ns$
$\langle proof \rangle$

lemma *card_set_minus*: $card\ (set\ xs - X) \leq length\ xs$
$\langle proof \rangle$

lemma *fo_nmlz_set*: $set\ (fo\_nmlz\ AD\ xs) =$
$set\ xs \cap Inl\ `\ AD \cup Inr\ `\ \{..<min\ (length\ xs)\ (card\ (set\ xs - Inl\ `\ AD))\}$
$\langle proof \rangle$

lemma *fo_nmlz_set_rev*: $set\ (fo\_nmlz\ AD\ xs) \subseteq Inl\ `\ AD \Longrightarrow set\ xs \subseteq Inl\ `\ AD$
$\langle proof \rangle$

lemma *inj_on_empty*: $inj\_on\ Map.empty\ (dom\ Map.empty)$ **and** *ran_empty_upto*: $ran\ Map.empty \subseteq$
$\{..<0\}$
$\langle proof \rangle$

lemma *fo_nmlz_ad_agr*: *ad_agr_list AD xs* (*fo_nmlz AD xs*)
⟨*proof*⟩

lemma *fo_nmlzd_mono*: $Inl -`$ *set xs* $\subseteq$ *AD* $\Longrightarrow$ *fo_nmlzd AD′ xs* $\Longrightarrow$ *fo_nmlzd AD xs*
⟨*proof*⟩

lemma *fo_nmlz_idem*: *fo_nmlzd AD ys* $\Longrightarrow$ *fo_nmlz AD ys* = *ys*
⟨*proof*⟩

lemma *fo_nmlz_take*: *take n* (*fo_nmlz AD xs*) = *fo_nmlz AD* (*take n xs*)
⟨*proof*⟩

fun *nall_tuples_rec* :: *′a set* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ (*′a* + *nat*) *table* **where**
nall_tuples_rec AD i 0 = {[]}
| *nall_tuples_rec AD i* (*Suc n*) = $\bigcup$((λ*as*. (λ*x*. *x* # *as*) ` (*Inl* ` *AD* ∪ *Inr* ` {..<*i*})) `
nall_tuples_rec AD i n) ∪ (λ*as*. *Inr i* # *as*) ` *nall_tuples_rec AD* (*Suc i*) *n*

lemma *nall_tuples_rec_Inl*: *vs* ∈ *nall_tuples_rec AD i n* $\Longrightarrow$ $Inl -`$ *set vs* $\subseteq$ *AD*
⟨*proof*⟩

lemma *nall_tuples_rec_length*: *xs* ∈ *nall_tuples_rec AD i n* $\Longrightarrow$ *length xs* = *n*
⟨*proof*⟩

lemma *fun_upd_id_map*: (*id_map i*)(*Inr i* ↦ *i*) = *id_map* (*Suc i*)
⟨*proof*⟩

lemma *id_mapD*: *id_map j* (*Inr i*) = *None* $\Longrightarrow$ *j* ≤ *i id_map j* (*Inr i*) = *Some x* $\Longrightarrow$ *i* < *j* ∧ *i* = *x*
⟨*proof*⟩

lemma *nall_tuples_rec_fo_nmlz_rec_sound*: *i* ≤ *j* $\Longrightarrow$ *xs* ∈ *nall_tuples_rec AD i n* $\Longrightarrow$
fo_nmlz_rec j (*id_map j*) *AD xs* = *xs*
⟨*proof*⟩

lemma *nall_tuples_rec_fo_nmlz_rec_complete*:
assumes *fo_nmlz_rec j* (*id_map j*) *AD xs* = *xs*
shows *xs* ∈ *nall_tuples_rec AD j* (*length xs*)
⟨*proof*⟩

lemma *nall_tuples_rec_fo_nmlz*: *xs* ∈ *nall_tuples_rec AD 0* (*length xs*) ⟷ *fo_nmlz AD xs* = *xs*
⟨*proof*⟩

lemma *fo_nmlzd_code*[*code*]: *fo_nmlzd AD xs* ⟷ *fo_nmlz AD xs* = *xs*
⟨*proof*⟩

lemma *nall_tuples_code*[*code*]: *nall_tuples AD n* = *nall_tuples_rec AD 0 n*
⟨*proof*⟩

lemma *exists_map*: *length xs* = *length ys* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ ∃*f*. *ys* = *map f xs*
⟨*proof*⟩

lemma *exists_fo_nmlzd*:
assumes *length xs* = *length ys distinct xs fo_nmlzd AD ys*
shows ∃*f*. *ys* = *fo_nmlz AD* (*map f xs*)
⟨*proof*⟩

lemma *list_induct2_rev*[*consumes 1*]: *length xs* = *length ys* $\Longrightarrow$ (*P* [] []) $\Longrightarrow$
($\bigwedge$*x y xs ys*. *P xs ys* $\Longrightarrow$ *P* (*xs* @ [*x*]) (*ys* @ [*y*])) $\Longrightarrow$ *P xs ys*

$\langle proof \rangle$

lemma *ad_agr_list_fo_nmlzd*:
  assumes *ad_agr_list AD vs vs' fo_nmlzd AD vs fo_nmlzd AD vs'*
  shows $vs = vs'$
  $\langle proof \rangle$

lemma *fo_nmlz_eqI*:
  assumes *ad_agr_list AD vs vs'*
  shows *fo_nmlz AD vs = fo_nmlz AD vs'*
  $\langle proof \rangle$

lemma *fo_nmlz_eqD*:
  assumes *fo_nmlz AD vs = fo_nmlz AD vs'*
  shows *ad_agr_list AD vs vs'*
  $\langle proof \rangle$

lemma *fo_nmlz_eq*: *fo_nmlz AD vs = fo_nmlz AD vs'* $\longleftrightarrow$ *ad_agr_list AD vs vs'*
  $\langle proof \rangle$

lemma *fo_nmlz_mono*:
  assumes $AD \subseteq AD'$ *Inl* $-$‘ *set xs* $\subseteq AD$
  shows *fo_nmlz AD' xs = fo_nmlz AD xs*
$\langle proof \rangle$

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

definition *proj_fmla* :: (*'a*, *'b*) *fo_fmla* $\Rightarrow$ *'c val set* $\Rightarrow$ *'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 = ($\forall x$. σ $x \neq$ *None* $\longrightarrow$ σ x = τ x)

definition *ext_tuple* :: *'a set* $\Rightarrow$ *nat list* $\Rightarrow$ *nat list* $\Rightarrow$
  (*'a* + *nat*) *list* $\Rightarrow$ (*'a* + *nat*) *list set* **where**
  ext_tuple AD fv_sub fv_sub_comp as = (*if fv_sub_comp* = [] *then* {*as*}
    else (λ*fs*. *map snd* (*merge* (*zip fv_sub as*) (*zip fv_sub_comp fs*))) ‘
    (*nall_tuples_rec AD* (*card* (*Inr* $-$‘ *set as*)) (*length fv_sub_comp*)))

lemma *ext_tuple_eq*: *length fv_sub* = *length as* $\Longrightarrow$
  ext_tuple AD fv_sub fv_sub_comp as =
  (λ*fs*. *map snd* (*merge* (*zip fv_sub as*) (*zip fv_sub_comp fs*))) ‘
  (*nall_tuples_rec AD* (*card* (*Inr* $-$‘ *set as*)) (*length fv_sub_comp*))
  $\langle proof \rangle$

lemma *map_map_of*: *length xs* = *length ys* $\Longrightarrow$ *distinct xs* $\Longrightarrow$
  ys = *map* (*the* $\circ$ (*map_of* (*zip xs ys*))) *xs*
  $\langle proof \rangle$

lemma *id_map_empty*: *id_map 0* = *Map.empty*
  $\langle proof \rangle$

lemma *fo_nmlz_rec_shift*:
  fixes *xs* :: (*'a* + *nat*) *list*
  shows *fo_nmlz_rec i* (*id_map i*) *AD xs* = *xs* $\Longrightarrow$
  i' = *card* (*Inr* $-$‘ (*Inr* ‘ {..<i} $\cup$ *set* (*take n xs*))) $\Longrightarrow$ $n \leq$ *length xs* $\Longrightarrow$

fo_nmlz_rec i′ (id_map i′) AD (drop n xs) = drop n xs
⟨*proof*⟩

fun *proj_tuple* :: *nat list* ⇒ (*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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *proj_tuple_length*:
assumes *sorted_distinct ns sorted_distinct ms set ns* ⊆ *set ms length ms* = *length xs*
shows *length* (*proj_tuple ns* (*zip ms xs*)) = *length ns*
⟨*proof*⟩

lemma *ext_tuple_sound*:
assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
set fv_sub ∩ *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*
xs ∈ *fo_nmlz AD* ‘ ⋃(*ext_tuple AD fv_sub fv_sub_comp* ‘ *ass*)
shows *fo_nmlz AD* (*proj_tuple fv_sub* (*zip fv_all xs*)) ∈ *ass*
xs ∈ *fo_nmlz AD* ‘ *proj_vals R fv_all*
⟨*proof*⟩

lemma *ext_tuple_complete*:
assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
set fv_sub ∩ *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*
xs = *fo_nmlz AD* (*map* σ *fv_all*) σ ∈ *R*
shows *xs* ∈ *fo_nmlz AD* ‘ ⋃(*ext_tuple AD fv_sub fv_sub_comp* ‘ *ass*)
⟨*proof*⟩

definition *ext_tuple_set AD ns ns′ X* = (*if ns′* = [] *then X else fo_nmlz AD* ‘ ⋃(*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*)
⟨*proof*⟩

lemma *ext_tuple_set_mono*: *A* ⊆ *B* ⟹ *ext_tuple_set AD ns ns′ A* ⊆ *ext_tuple_set AD ns ns′ B*
⟨*proof*⟩

lemma *ext_tuple_correct*:
  assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
    set fv_sub $\cap$ *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*⟩

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*⟩

lemma *proj_tuple_correct*:
  assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
    set fv_sub $\cap$ *set fv_sub_comp* = {} *set fv_sub* $\cup$ *set fv_sub_comp* = *set fv_all*
    ass = *fo_nmlz AD ‘ proj_vals R fv_sub*
    $\bigwedge\sigma\ \tau$. *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*
  shows *xs* $\in$ *fo_nmlz AD ‘* $\bigcup$(*ext_tuple AD fv_sub fv_sub_comp ‘ ass*) $\longleftrightarrow$
    fo_nmlz AD (*proj_tuple fv_sub* (*zip fv_all xs*)) $\in$ *ass*
  ⟨*proof*⟩

fun *unify_vals_terms* :: ($'a$ + $'c$) *list* $\Rightarrow$ ($'a$ *fo_term*) *list* $\Rightarrow$ (*nat* $\rightharpoonup$ ($'a$ + $'c$)) $\Rightarrow$
  (*nat* $\rightharpoonup$ ($'a$ + $'c$)) *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* $\Rightarrow$ (*if v* = *x then unify_vals_terms vs ts* σ *else None*)
    | *None* $\Rightarrow$ *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* $\sigma' \Longrightarrow$ *extends_subst* $\sigma\ \sigma'$
  ⟨*proof*⟩

lemma *unify_vals_terms_sound*: *unify_vals_terms vs ts* σ = *Some* $\sigma' \Longrightarrow$ (*the* $\circ\ \sigma'$) $\odot$*e ts* = *vs*
  ⟨*proof*⟩

lemma *unify_vals_terms_complete*: σ'' $\odot$*e ts* = *vs* $\Longrightarrow$ ($\bigwedge$*n*. σ *n* $\neq$ *None* $\Longrightarrow \sigma$ *n* = *Some* (σ'' *n*)) $\Longrightarrow$
  $\exists\sigma'$. *unify_vals_terms vs ts* σ = *Some* σ'
  ⟨*proof*⟩

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

lemma *eval_table*:
  fixes *X* :: ($'a$ + $'c$) *table*
  shows *eval_table ts X* = *proj_vals* {σ. σ $\odot$*e ts* $\in$ *X*} (*fv_fo_terms_list ts*)
⟨*proof*⟩

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

lemma $ad\_agr\_close\_rec\_length$: $ys \in ad\_agr\_close\_rec\ i\ m\ AD\ xs \Longrightarrow length\ xs = length\ ys$
$\langle proof \rangle$

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

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

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

lemma $ad\_agr\_close\_sound$:
assumes $ys \in ad\_agr\_close\ Y\ xs\ fo\_nmlzd\ X\ xs\ X \cap Y = \{\}$
shows $fo\_nmlzd\ (X \cup Y)\ ys \wedge ad\_agr\_list\ X\ xs\ ys$
$\langle proof \rangle$

lemma $ad\_agr\_close\_complete$:
assumes $X \cap Y = \{\}\ fo\_nmlzd\ X\ xs\ fo\_nmlzd\ (X \cup Y)\ ys\ ad\_agr\_list\ X\ xs\ ys$
shows $ys \in ad\_agr\_close\ Y\ xs$
$\langle proof \rangle$

lemma $ad\_agr\_close\_empty$: $fo\_nmlzd\ X\ xs \Longrightarrow ad\_agr\_close\ \{\}\ xs = \{xs\}$
$\langle proof \rangle$

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 $\bigcup(ad\_agr\_close\ (AD - AD')\ `\ fo\_nmlz\ AD'\ `\ proj\_vals\ R\ ns) = fo\_nmlz\ AD\ `\ proj\_vals\ R\ ns$
$\langle proof \rangle$

lemma $ad\_agr\_close\_correct$:
assumes $AD' \subseteq AD$
$\bigwedge \sigma\ \tau.\ ad\_agr\_sets\ (set\ (fv\_fo\_fmla\_list\ \varphi))\ (set\ (fv\_fo\_fmla\_list\ \varphi))\ AD'\ \sigma\ \tau \Longrightarrow$
$\sigma \in R \longleftrightarrow \tau \in R$

shows $\bigcup$(*ad_agr_close* (*AD* $-$ *AD'*) ‘ *fo_nmlz AD'* ‘ *proj_fmla* φ *R*) = *fo_nmlz AD* ‘ *proj_fmla* φ *R*
⟨*proof*⟩

definition *ad_agr_close_set AD X* = (*if Set.is_empty AD then X else* $\bigcup$(*ad_agr_close AD* ‘ *X*))

lemma *ad_agr_close_set_eq*: *Ball X* (*fo_nmlzd AD'*) $\Longrightarrow$ *ad_agr_close_set AD X* = $\bigcup$(*ad_agr_close AD* ‘ *X*)
⟨*proof*⟩

lemma *Ball_fo_nmlzd*: *Ball* (*fo_nmlz AD* ‘ *X*) (*fo_nmlzd AD*)
⟨*proof*⟩

lemmas *ad_agr_close_set_nmlz_eq* = *ad_agr_close_set_eq*[*OF Ball_fo_nmlzd*]

definition *eval_pred* :: ('*a fo_term*) *list* $\Rightarrow$ '*a table* $\Rightarrow$ ('*a*, '*c*) *fo_t* **where**
eval_pred ts X = (*let AD* = $\bigcup$(*set* (*map set_fo_term ts*)) $\cup$ $\bigcup$(*set* ‘ *X*) *in*
(*AD*, *length* (*fv_fo_terms_list ts*), *eval_table ts* (*map Inl* ‘ *X*)))

definition *eval_bool* :: *bool* $\Rightarrow$ ('*a*, '*c*) *fo_t* **where**
eval_bool b = (*if b then* ({}, *0*, {[]}) *else* ({}, *0*, {}))

definition *eval_eq* :: '*a fo_term* $\Rightarrow$ '*a fo_term* $\Rightarrow$ ('*a*, *nat*) *fo_t* **where**
eval_eq t t' = (*case t of Var n* $\Rightarrow$
(*case t' of Var n'* $\Rightarrow$
if n = *n' then* ({}, *1*, {[*Inr 0*]})
else ({}, *2*, {[*Inr 0*, *Inr 0*]})
| *Const c'* $\Rightarrow$ ({*c'*}, *1*, {[*Inl c'*]}))
| *Const c* $\Rightarrow$
(*case t' of Var n'* $\Rightarrow$ ({*c*}, *1*, {[*Inl c*]})
| *Const c'* $\Rightarrow$ *if c* = *c' then* ({*c*}, *0*, {[]}) *else* ({*c*, *c'*}, *0*, {})))

fun *eval_neg* :: *nat list* $\Rightarrow$ ('*a*, *nat*) *fo_t* $\Rightarrow$ ('*a*, *nat*) *fo_t* **where**
eval_neg ns (*AD*, _, *X*) = (*AD*, *length ns*, *nall_tuples AD* (*length ns*) $-$ *X*)

definition *eval_conj_tuple AD ns*φ *ns*ψ *xs ys* =
(*let cxs* = *filter* (λ(*n*, *x*). *n* $\notin$ *set ns*ψ $\wedge$ *isl x*) (*zip ns*φ *xs*);
nxs = *map fst* (*filter* (λ(*n*, *x*). *n* $\notin$ *set ns*ψ $\wedge$ $\neg$*isl x*) (*zip ns*φ *xs*));
cys = *filter* (λ(*n*, *y*). *n* $\notin$ *set ns*φ $\wedge$ *isl y*) (*zip ns*ψ *ys*);
nys = *map fst* (*filter* (λ(*n*, *y*). *n* $\notin$ *set ns*φ $\wedge$ $\neg$*isl y*) (*zip ns*ψ *ys*)) *in*
fo_nmlz AD ‘ *ext_tuple* {} (*sort* (*ns*φ @ *map fst cys*)) *nys* (*map snd* (*merge* (*zip ns*φ *xs*) *cys*)) $\cap$
fo_nmlz AD ‘ *ext_tuple* {} (*sort* (*ns*ψ @ *map fst cxs*)) *nxs* (*map snd* (*merge* (*zip ns*ψ *ys*) *cxs*)))

definition *eval_conj_set AD ns*φ *X*φ *ns*ψ *X*ψ = $\bigcup$((λ*xs*. $\bigcup$(*eval_conj_tuple AD ns*φ *ns*ψ *xs* ‘ *X*ψ)) ‘ *X*φ)

definition *idx_join AD ns ns*φ *X*φ *ns*ψ *X*ψ =
(*let idx*φ' = *cluster* (*Some* $\circ$ (λ*xs*. *fo_nmlz AD* (*proj_tuple ns* (*zip ns*φ *xs*)))) *X*φ;
idxψ' = *cluster* (*Some* $\circ$ (λ*ys*. *fo_nmlz AD* (*proj_tuple ns* (*zip ns*ψ *ys*)))) *X*ψ *in*
set_of_idx (*mapping_join* (λ*X*φ'' *X*ψ''. *eval_conj_set AD ns*φ *X*φ'' *ns*ψ *X*ψ'') *idx*φ' *idx*ψ'))

fun *eval_conj* :: *nat list* $\Rightarrow$ ('*a*, *nat*) *fo_t* $\Rightarrow$ *nat list* $\Rightarrow$ ('*a*, *nat*) *fo_t* $\Rightarrow$
('*a*, *nat*) *fo_t* **where**
eval_conj nsφ (*AD*φ, _, *X*φ) *ns*ψ (*AD*ψ, _, *X*ψ) = (*let AD* = *AD*φ $\cup$ *AD*ψ; *AD*$\Delta\varphi$ = *AD* $-$ *AD*φ; *AD*$\Delta\psi$ = *AD* $-$ *AD*ψ; *ns* = *filter* (λ*n*. *n* $\in$ *set ns*ψ) *ns*φ *in*
(*AD*, *card* (*set ns*φ $\cup$ *set ns*ψ), *idx_join AD ns ns*φ (*ad_agr_close_set AD*$\Delta\varphi$ *X*φ) *ns*ψ (*ad_agr_close_set AD*$\Delta\psi$ *X*ψ)))

fun *eval_ajoin* :: *nat list* $\Rightarrow$ ('*a*, *nat*) *fo_t* $\Rightarrow$ *nat list* $\Rightarrow$ ('*a*, *nat*) *fo_t* $\Rightarrow$

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

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

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

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

lemma $combine\_map2$: **assumes** $length\ ys = length\ xs\ length\ ys' = length\ xs'$
$distinct\ xs\ distinct\ xs'\ set\ xs \cap set\ xs' = \{\}$
shows $\exists f.\ ys = map\ f\ xs \wedge ys' = map\ f\ xs'$
$\langle proof \rangle$

lemma $combine\_map3$: **assumes** $length\ ys = length\ xs\ length\ ys' = length\ xs'\ length\ ys'' = length\ xs''$
$distinct\ xs\ distinct\ xs'\ distinct\ xs''\ set\ xs \cap set\ xs' = \{\}\ set\ xs \cap set\ xs'' = \{\}\ set\ xs' \cap set\ xs'' = \{\}$
shows $\exists f.\ ys = map\ f\ xs \wedge ys' = map\ f\ xs' \wedge ys'' = map\ f\ xs''$
$\langle proof \rangle$

lemma $distinct\_set\_zip$: $length\ nsx = length\ xs \Longrightarrow distinct\ nsx \Longrightarrow$
$(a,\ b) \in set\ (zip\ nsx\ xs) \Longrightarrow (a,\ ba) \in set\ (zip\ nsx\ xs) \Longrightarrow b = ba$
$\langle proof \rangle$

lemma $fo\_nmlz\_idem\_isl$:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow (case\ x\ of\ Inl\ z \Rightarrow z \in X\ |\ \_ \Rightarrow False)$
shows $fo\_nmlz\ X\ xs = xs$
$\langle proof \rangle$

lemma set_zip_mapI: $x \in set\ xs \Longrightarrow (f\ x,\ g\ x) \in set\ (zip\ (map\ f\ xs)\ (map\ g\ xs))$
$\langle proof \rangle$

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

shows $f\ x = g\ x$
$\langle proof \rangle$

lemma $eval\_conj\_tuple\_close\_empty2$:
assumes $fo\_nmlzd\ X\ xs$ $fo\_nmlzd\ Y\ ys$
$length\ nsx = length\ xs$ $length\ nsy = length\ ys$
$sorted\_distinct\ nsx$ $sorted\_distinct\ nsy$
$sorted\_distinct\ ns$ $set\ ns \subseteq set\ nsx \cap set\ nsy$
$fo\_nmlz\ (X \cap Y)\ (proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq fo\_nmlz\ (X \cap Y)\ (proj\_tuple\ ns\ (zip\ nsy\ ys)) \vee$
$(proj\_tuple\ ns\ (zip\ nsx\ xs) \neq proj\_tuple\ ns\ (zip\ nsy\ ys) \wedge$
$(\forall x \in set\ (proj\_tuple\ ns\ (zip\ nsx\ xs)).\ isl\ x) \wedge (\forall y \in set\ (proj\_tuple\ ns\ (zip\ nsy\ ys)).\ isl\ y))$
$xs' \in ad\_agr\_close\ ((X \cup Y) - X)\ xs$ $ys' \in ad\_agr\_close\ ((X \cup Y) - Y)\ ys$
shows $eval\_conj\_tuple\ (X \cup Y)\ nsx\ nsy\ xs'\ ys' = \{\}$
$\langle proof \rangle$

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

lemma $eval\_conj\_tuple\_empty2$:
assumes $fo\_nmlzd\ Z\ xs$ $fo\_nmlzd\ Z\ ys$
$length\ nsx = length\ xs$ $length\ nsy = length\ ys$
$sorted\_distinct\ nsx$ $sorted\_distinct\ nsy$
$sorted\_distinct\ ns$ $set\ ns \subseteq set\ nsx \cap set\ nsy$
$fo\_nmlz\ Z\ (proj\_tuple\ ns\ (zip\ nsx\ xs)) \neq fo\_nmlz\ Z\ (proj\_tuple\ ns\ (zip\ nsy\ ys)) \vee$
$(proj\_tuple\ ns\ (zip\ nsx\ xs) \neq proj\_tuple\ ns\ (zip\ nsy\ ys) \wedge$
$(\forall x \in set\ (proj\_tuple\ ns\ (zip\ nsx\ xs)).\ isl\ x) \wedge (\forall y \in set\ (proj\_tuple\ ns\ (zip\ nsy\ ys)).\ isl\ y))$
shows $eval\_conj\_tuple\ Z\ nsx\ nsy\ xs\ ys = \{\}$
$\langle proof \rangle$

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

lemma $nall\_tuples\_rec\_filter$:
assumes $xs \in nall\_tuples\_rec\ AD\ n\ (length\ xs)$ $ys = filter\ (\lambda x.\ \neg isl\ x)\ xs$
shows $ys \in nall\_tuples\_rec\ \{\}\ n\ (length\ ys)$
$\langle proof \rangle$

lemma $nall\_tuples\_rec\_filter\_rev$:
assumes $ys \in nall\_tuples\_rec\ \{\}\ n\ (length\ ys)$ $ys = filter\ (\lambda x.\ \neg isl\ x)\ xs$
$Inl\ -`\ set\ xs \subseteq AD$
shows $xs \in nall\_tuples\_rec\ AD\ n\ (length\ xs)$
$\langle proof \rangle$

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⟩

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

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 $X\varphi$_*def*: $X\varphi$ = *fo_nmlz AD* ‘ *proj_vals* $R\varphi$ *ns*φ
and $X\psi$_*def*: $X\psi$ = *fo_nmlz AD* ‘ *proj_vals* $R\psi$ *ns*ψ
and *distinct*: *sorted_distinct ns*φ *sorted_distinct ns*ψ
shows *eval_conj_set AD ns*φ $X\varphi$ *ns*ψ $X\psi$ = *ext_tuple_set AD ns*φ *ns*φ' $X\varphi \cap$ *ext_tuple_set AD ns*ψ *ns*ψ' $X\psi$
⟨*proof*⟩

lemma *esat_exists_not_fv*: $n \notin$ *fv_fo_fmla* $\varphi \Longrightarrow X \neq \{\} \Longrightarrow$
esat (*Exists n* φ) I σ $X \longleftrightarrow$ *esat* φ I σ X
⟨*proof*⟩

lemma *esat_forall_not_fv*: $n \notin$ *fv_fo_fmla* $\varphi \Longrightarrow X \neq \{\} \Longrightarrow$
esat (*Forall n* φ) I σ $X \longleftrightarrow$ *esat* φ I σ X
⟨*proof*⟩

lemma *proj_sat_vals*: *proj_sat* φ I =
proj_vals $\{\sigma.$ *sat* φ I $\sigma\}$ (*fv_fo_fmla_list* φ)
⟨*proof*⟩

lemma *fv_fo_fmla_list_Pred*: *remdups_adj* (*sort* (*fv_fo_terms_list ts*)) = *fv_fo_terms_list ts*
⟨*proof*⟩

lemma *ad_agr_list_fv_list'*: $\bigcup$(*set* (*map set_fo_term ts*)) $\subseteq X \Longrightarrow$
ad_agr_list X (*map* σ (*fv_fo_terms_list ts*)) (*map* τ (*fv_fo_terms_list ts*)) $\Longrightarrow$
ad_agr_list X (σ $\odot$*e ts*) (τ $\odot$*e ts*)
⟨*proof*⟩

lemma *ext_tuple_ad_agr_close*:
assumes $S\varphi$_*def*: $S\varphi \equiv \{\sigma.$ *esat* φ I σ *UNIV*$\}$
and *AD_sub*: *act_edom* φ $I \subseteq AD\varphi$ $AD\varphi \subseteq AD$
and $X\varphi$_*def*: $X\varphi$ = *fo_nmlz* $AD\varphi$ ‘ *proj_vals* $S\varphi$ (*fv_fo_fmla_list* φ)
and *ns*φ'_*def*: *ns*φ' = *filter* ($\lambda n.$ $n \notin$ *fv_fo_fmla* φ) *ns*ψ
and *sd_ns*ψ: *sorted_distinct ns*ψ
and *fv_Un*: *fv_fo_fmla* ψ = *fv_fo_fmla* $\varphi \cup$ *set ns*ψ
shows *ext_tuple_set AD* (*fv_fo_fmla_list* φ) *ns*φ' (*ad_agr_close_set* ($AD - AD\varphi$) $X\varphi$) =
fo_nmlz AD ‘ *proj_vals* $S\varphi$ (*fv_fo_fmla_list* ψ)
ad_agr_close_set ($AD - AD\varphi$) $X\varphi$ = *fo_nmlz AD* ‘ *proj_vals* $S\varphi$ (*fv_fo_fmla_list* φ)
⟨*proof*⟩

lemma *proj_ext_tuple*:
assumes $S\varphi$_*def*: $S\varphi \equiv \{\sigma.$ *esat* φ I σ *UNIV*$\}$
and *AD_sub*: *act_edom* φ $I \subseteq AD$
and $X\varphi$_*def*: $X\varphi$ = *fo_nmlz AD* ‘ *proj_vals* $S\varphi$ (*fv_fo_fmla_list* φ)
and *ns*φ'_*def*: *ns*φ' = *filter* ($\lambda n.$ $n \notin$ *fv_fo_fmla* φ) *ns*ψ
and *sd_ns*ψ: *sorted_distinct ns*ψ
and *fv_Un*: *fv_fo_fmla* ψ = *fv_fo_fmla* $\varphi \cup$ *set ns*ψ
and *Z_props*: $\bigwedge$*xs. xs* $\in Z \Longrightarrow$ *fo_nmlz AD xs* = *xs* $\wedge$ *length xs* = *length* (*fv_fo_fmla_list* ψ)
shows $Z \cap$ *ext_tuple_set AD* (*fv_fo_fmla_list* φ) *ns*φ' $X\varphi$ =
$\{$*xs* $\in Z.$ *fo_nmlz AD* (*proj_tuple* (*fv_fo_fmla_list* φ) (*zip* (*fv_fo_fmla_list* ψ) *xs*)) $\in X\varphi\}$
$Z -$ *ext_tuple_set AD* (*fv_fo_fmla_list* φ) *ns*φ' $X\varphi$ =
$\{$*xs* $\in Z.$ *fo_nmlz AD* (*proj_tuple* (*fv_fo_fmla_list* φ) (*zip* (*fv_fo_fmla_list* ψ) *xs*)) $\notin X\varphi\}$
⟨*proof*⟩

lemma *fo_nmlz_proj_sub*: *fo_nmlz AD* ‘ *proj_fmla* φ $R \subseteq$ *nall_tuples AD* (*nfv* φ)
⟨*proof*⟩

lemma *fin_ad_agr_list_iff*:
fixes *AD* :: ('*a* :: *infinite*) *set*

assumes $\mathit{finite}\ AD\ \bigwedge vs.\ vs \in Z \Longrightarrow \mathit{length}\ vs = n$
$Z = \{ts.\ \exists\, ts' \in X.\ \mathit{ad\_agr\_list}\ AD\ (\mathit{map}\ \mathit{Inl}\ ts)\ ts'\}$
shows $\mathit{finite}\ Z \longleftrightarrow \bigcup(\mathit{set}\ `\ Z) \subseteq AD$
$\langle \mathit{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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*⟩

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)
  ⟨*proof*⟩

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*⟩

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*⟩

lemma *ad_agr_list_eval*: $\bigcup$(*set* (*map set_fo_term ts*)) $\subseteq$ *AD* $\Longrightarrow$ *ad_agr_list AD* (σ $\odot e$ *ts*) *zs* $\Longrightarrow$
  $\exists\tau$. *zs* = τ $\odot e$ *ts*
⟨*proof*⟩

lemma *sp_equiv_list_fv_list*:
  assumes *sp_equiv_list* (σ $\odot e$ *ts*) (τ $\odot e$ *ts*)
  shows *sp_equiv_list* (*map* σ (*fv_fo_terms_list ts*)) (*map* τ (*fv_fo_terms_list ts*))
⟨*proof*⟩

lemma *ad_agr_list_fv_list*: *ad_agr_list* X (σ $\odot e$ *ts*) (τ $\odot e$ *ts*) $\Longrightarrow$
  ad_agr_list X (*map* σ (*fv_fo_terms_list ts*)) (*map* τ (*fv_fo_terms_list ts*))
  ⟨*proof*⟩

lemma *eval_bool*: *fo_wf* (*Bool b*) I (*eval_bool b*)
  ⟨*proof*⟩

lemma *eval_eq*: **fixes** I :: $'b \times nat \Rightarrow {'a}$ :: *infinite list set*
  shows *fo_wf* (*Eqa* t t') I (*eval_eq* t t')
⟨*proof*⟩

lemma *fv_fo_terms_list_Var*: *fv_fo_terms_list_rec* (*map Var ns*) = *ns*
  ⟨*proof*⟩

lemma *eval_eterms_map_Var*: σ $\odot e$ *map Var ns* = *map* σ *ns*
  ⟨*proof*⟩

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*⟩

lemma *fo_rep_norm*:

fixes $AD :: ('a :: \mathit{infinite})\ \mathit{set}$
assumes $\mathit{fo\_wf}\ \varphi\ I\ (AD,\ n,\ X)$
shows $X = \mathit{fo\_nmlz}\ AD\ `\ \mathit{map}\ \mathit{Inl}\ `\ \mathit{fo\_rep}\ (AD,\ n,\ X)$
⟨*proof*⟩

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

lemma *eval_neg*:
fixes $\varphi :: ('a :: \mathit{infinite},\ 'b)\ \mathit{fo\_fmla}$
assumes wf: $\mathit{fo\_wf}\ \varphi\ I\ t$
shows $\mathit{fo\_wf}\ (\mathit{Neg}\ \varphi)\ I\ (\mathit{eval\_neg}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t)$
⟨*proof*⟩

definition $\mathit{cross\_with}\ f\ t\ t' = \bigcup((\lambda xs.\ \bigcup(f\ xs\ `\ t'))\ `\ t)$

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 $\mathit{set\_of\_idx}\ (\mathit{mapping\_join}\ (\mathit{cross\_with}\ f)\ (\mathit{cluster}\ (\mathit{Some} \circ h)\ t)\ (\mathit{cluster}\ (\mathit{Some} \circ h')\ t')) = \mathit{cross\_with}\ f\ t\ t'$
⟨*proof*⟩

lemma *fo_nmlzd_mono_sub*: $X \subseteq X' \Longrightarrow \mathit{fo\_nmlzd}\ X\ xs \Longrightarrow \mathit{fo\_nmlzd}\ X'\ xs$
⟨*proof*⟩

lemma *idx_join*:
assumes $X\varphi\_\mathit{props}$: $\bigwedge vs.\ vs \in X\varphi \Longrightarrow \mathit{fo\_nmlzd}\ AD\ vs \wedge \mathit{length}\ vs = \mathit{length}\ ns\varphi$
assumes $X\psi\_\mathit{props}$: $\bigwedge vs.\ vs \in X\psi \Longrightarrow \mathit{fo\_nmlzd}\ AD\ vs \wedge \mathit{length}\ vs = \mathit{length}\ ns\psi$
assumes $\mathit{sd\_ns}$: $\mathit{sorted\_distinct}\ ns\varphi\ \mathit{sorted\_distinct}\ ns\psi$
assumes $\mathit{ns\_def}$: $ns = \mathit{filter}\ (\lambda n.\ n \in \mathit{set}\ ns\psi)\ ns\varphi$
shows $\mathit{idx\_join}\ AD\ ns\ ns\varphi\ X\varphi\ ns\psi\ X\psi = \mathit{eval\_conj\_set}\ AD\ ns\varphi\ X\varphi\ ns\psi\ X\psi$
⟨*proof*⟩

lemma *proj_fmla_conj_sub*:
assumes AD_sub: $\mathit{act\_edom}\ \psi\ I \subseteq AD$
shows $\mathit{fo\_nmlz}\ AD\ `\ \mathit{proj\_fmla}\ (\mathit{Conj}\ \varphi\ \psi)\ \{\sigma.\ \mathit{esat}\ \varphi\ I\ \sigma\ \mathit{UNIV}\} \cap$
$\mathit{fo\_nmlz}\ AD\ `\ \mathit{proj\_fmla}\ (\mathit{Conj}\ \varphi\ \psi)\ \{\sigma.\ \mathit{esat}\ \psi\ I\ \sigma\ \mathit{UNIV}\} \subseteq$
$\mathit{fo\_nmlz}\ AD\ `\ \mathit{proj\_fmla}\ (\mathit{Conj}\ \varphi\ \psi)\ \{\sigma.\ \mathit{esat}\ (\mathit{Conj}\ \varphi\ \psi)\ I\ \sigma\ \mathit{UNIV}\}$
⟨*proof*⟩

lemma *eval_conj*:
fixes $\varphi :: ('a :: \mathit{infinite},\ 'b)\ \mathit{fo\_fmla}$
assumes wf: $\mathit{fo\_wf}\ \varphi\ I\ t\varphi\ \mathit{fo\_wf}\ \psi\ I\ t\psi$
shows $\mathit{fo\_wf}\ (\mathit{Conj}\ \varphi\ \psi)\ I\ (\mathit{eval\_conj}\ (\mathit{fv\_fo\_fmla\_list}\ \varphi)\ t\varphi\ (\mathit{fv\_fo\_fmla\_list}\ \psi)\ t\psi)$
⟨*proof*⟩

lemma *map_values_cluster*: $(\bigwedge w\ z\ Z.\ Z \subseteq X \Longrightarrow z \in Z \Longrightarrow w \in f\ (h\ z)\ \{z\} \Longrightarrow w \in f\ (h\ z)\ Z) \Longrightarrow$
$(\bigwedge w\ z\ Z.\ Z \subseteq X \Longrightarrow z \in Z \Longrightarrow w \in f\ (h\ z)\ Z \Longrightarrow (\exists z' \in Z.\ w \in f\ (h\ z)\ \{z'\})) \Longrightarrow$
$\mathit{set\_of\_idx}\ (\mathit{Mapping.map\_values}\ f\ (\mathit{cluster}\ (\mathit{Some} \circ h)\ X)) = \bigcup((\lambda x.\ f\ (h\ x)\ \{x\})\ `\ X)$
⟨*proof*⟩

lemma *fo_nmlz_twice*:
assumes $\mathit{sorted\_distinct}\ ns\ \mathit{sorted\_distinct}\ ns'\ \mathit{set}\ ns \subseteq \mathit{set}\ ns'$
shows $\mathit{fo\_nmlz}\ AD\ (\mathit{proj\_tuple}\ ns\ (\mathit{zip}\ ns'\ (\mathit{fo\_nmlz}\ AD\ (\mathit{map}\ \sigma\ ns')))) = \mathit{fo\_nmlz}\ AD\ (\mathit{map}\ \sigma\ ns)$
⟨*proof*⟩

lemma *map_values_cong*:
assumes $\bigwedge x\ y.\ Mapping.lookup\ t\ x = Some\ y \Longrightarrow f\ x\ y = f'\ x\ y$
shows $Mapping.map\_values\ f\ t = Mapping.map\_values\ f'\ t$
⟨*proof*⟩

lemma *ad_agr_close_set_length*: $z \in ad\_agr\_close\_set\ AD\ X \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow length\ x = n) \Longrightarrow length\ z = n$
⟨*proof*⟩

lemma *ad_agr_close_set_sound*: $z \in ad\_agr\_close\_set\ (AD - AD')\ X \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow fo\_nmlzd\ AD'\ x) \Longrightarrow AD' \subseteq AD \Longrightarrow fo\_nmlzd\ AD\ z$
⟨*proof*⟩

lemma *ext_tuple_set_length*: $z \in ext\_tuple\_set\ AD\ ns\ ns'\ X \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow length\ x = length\ ns) \Longrightarrow length\ z = length\ ns + length\ ns'$
⟨*proof*⟩

lemma *eval_ajoin*:
fixes $\varphi :: ('a :: infinite, 'b)\ fo\_fmla$
assumes wf: $fo\_wf\ \varphi\ I\ t\varphi\ fo\_wf\ \psi\ I\ t\psi$
shows $fo\_wf\ (Conj\ \varphi\ (Neg\ \psi))\ I$
$(eval\_ajoin\ (fv\_fo\_fmla\_list\ \varphi)\ t\varphi\ (fv\_fo\_fmla\_list\ \psi)\ t\psi)$
⟨*proof*⟩

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

lemma *fv_ex_all*:
assumes $pos\ i\ (fv\_fo\_fmla\_list\ \varphi) = None$
shows $fv\_fo\_fmla\_list\ (Exists\ i\ \varphi) = fv\_fo\_fmla\_list\ \varphi$
$fv\_fo\_fmla\_list\ (Forall\ i\ \varphi) = fv\_fo\_fmla\_list\ \varphi$
⟨*proof*⟩

lemma *nfv_ex_all*:
assumes $Some$: $pos\ i\ (fv\_fo\_fmla\_list\ \varphi) = Some\ j$
shows $nfv\ \varphi = Suc\ (nfv\ (Exists\ i\ \varphi))\ nfv\ \varphi = Suc\ (nfv\ (Forall\ i\ \varphi))$
⟨*proof*⟩

lemma *fv_fo_fmla_list_exists*: $fv\_fo\_fmla\_list\ (Exists\ n\ \varphi) = filter\ ((\neq)\ n)\ (fv\_fo\_fmla\_list\ \varphi)$
⟨*proof*⟩

lemma *eval_exists*:
fixes $\varphi :: ('a :: infinite, 'b)\ fo\_fmla$
assumes wf: $fo\_wf\ \varphi\ I\ t$
shows $fo\_wf\ (Exists\ i\ \varphi)\ I\ (eval\_exists\ i\ (fv\_fo\_fmla\_list\ \varphi)\ t)$
⟨*proof*⟩

lemma *fv_fo_fmla_list_forall*: $fv\_fo\_fmla\_list\ (Forall\ n\ \varphi) = filter\ ((\neq)\ n)\ (fv\_fo\_fmla\_list\ \varphi)$
⟨*proof*⟩

lemma *pairwise_take_drop*:
assumes $pairwise\ P\ (set\ (zip\ xs\ ys))\ length\ xs = length\ ys$
shows $pairwise\ P\ (set\ (zip\ (take\ i\ xs\ @\ drop\ (Suc\ i)\ xs)\ (take\ i\ ys\ @\ drop\ (Suc\ i)\ ys)))$

⟨*proof*⟩

lemma *fo_nmlz_set_card*:
$\mathit{fo\_nmlz}\ AD\ xs = xs \Longrightarrow \mathit{set}\ xs = \mathit{set}\ xs \cap \mathit{Inl}\ `\ AD \cup \mathit{Inr}\ `\ \{..<\mathit{card}\ (\mathit{Inr}\ -`\ \mathit{set}\ xs)\}$
⟨*proof*⟩

lemma *ad_agr_list_take_drop*: $\mathit{ad\_agr\_list}\ AD\ xs\ ys \Longrightarrow$
$\mathit{ad\_agr\_list}\ AD\ (\mathit{take}\ i\ xs\ @\ \mathit{drop}\ (\mathit{Suc}\ i)\ xs)\ (\mathit{take}\ i\ ys\ @\ \mathit{drop}\ (\mathit{Suc}\ i)\ ys)$
⟨*proof*⟩

lemma *fo_nmlz_rem_nth_add_nth*:
assumes $\mathit{fo\_nmlz}\ AD\ zs = zs\ \ i \le \mathit{length}\ zs$
shows $\mathit{fo\_nmlz}\ AD\ (\mathit{rem\_nth}\ i\ (\mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z\ zs))) = zs$
⟨*proof*⟩

lemma *ad_agr_list_add*:
assumes $\mathit{ad\_agr\_list}\ AD\ xs\ ys\ \ i \le \mathit{length}\ xs$
shows $\exists z' \in \mathit{Inl}\ `\ AD \cup \mathit{Inr}\ `\ \{..<\mathit{Suc}\ (\mathit{card}\ (\mathit{Inr}\ -`\ \mathit{set}\ ys))\} \cup \mathit{set}\ ys.$
$\mathit{ad\_agr\_list}\ AD\ (\mathit{take}\ i\ xs\ @\ z\ \#\ \mathit{drop}\ i\ xs)\ (\mathit{take}\ i\ ys\ @\ z'\ \#\ \mathit{drop}\ i\ ys)$
⟨*proof*⟩

lemma *add_nth_restrict*:
assumes $\mathit{fo\_nmlz}\ AD\ zs = zs\ \ i \le \mathit{length}\ zs$
shows $\exists z' \in \mathit{Inl}\ `\ AD \cup \mathit{Inr}\ `\ \{..<\mathit{Suc}\ (\mathit{card}\ (\mathit{Inr}\ -`\ \mathit{set}\ zs))\}.$
$\mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z\ zs) = \mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z'\ zs)$
⟨*proof*⟩

lemma *fo_nmlz_add_rem*:
assumes $i \le \mathit{length}\ zs$
shows $\exists z'.\ \mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z\ zs) = \mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z'\ (\mathit{fo\_nmlz}\ AD\ zs))$
⟨*proof*⟩

lemma *fo_nmlz_add_rem'*:
assumes $i \le \mathit{length}\ zs$
shows $\exists z'.\ \mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z\ (\mathit{fo\_nmlz}\ AD\ zs)) = \mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z'\ zs)$
⟨*proof*⟩

lemma *fo_nmlz_add_nth_rem_nth*:
assumes $\mathit{fo\_nmlz}\ AD\ xs = xs\ \ i < \mathit{length}\ xs$
shows $\exists z.\ \mathit{fo\_nmlz}\ AD\ (\mathit{add\_nth}\ i\ z\ (\mathit{fo\_nmlz}\ AD\ (\mathit{rem\_nth}\ i\ xs))) = xs$
⟨*proof*⟩

lemma *sp_equiv_list_almost_same*: $\mathit{sp\_equiv\_list}\ (xs\ @\ v\ \#\ ys)\ (xs\ @\ w\ \#\ ys) \Longrightarrow$
$v \in \mathit{set}\ xs \cup \mathit{set}\ ys \vee w \in \mathit{set}\ xs \cup \mathit{set}\ ys \Longrightarrow v = w$
⟨*proof*⟩

lemma *ad_agr_list_add_nth*:
assumes $i \le \mathit{length}\ zs\ \ \mathit{ad\_agr\_list}\ AD\ (\mathit{add\_nth}\ i\ v\ zs)\ (\mathit{add\_nth}\ i\ w\ zs)\ \ v \ne w$
shows $\{v, w\} \cap (\mathit{Inl}\ `\ AD \cup \mathit{set}\ zs) = \{\}$
⟨*proof*⟩

lemma *Inr_in_tuple*:
assumes $\mathit{fo\_nmlz}\ AD\ zs = zs\ \ n < \mathit{card}\ (\mathit{Inr}\ -`\ \mathit{set}\ zs)$
shows $\mathit{Inr}\ n \in \mathit{set}\ zs$
⟨*proof*⟩

lemma *card_wit_sub*:
assumes $\mathit{finite}\ Z\ \ \mathit{card}\ Z \le \mathit{card}\ \{ts \in X.\ \exists z \in Z.\ ts = f\ z\}$

shows f ‘ $Z \subseteq X$
⟨*proof*⟩

lemma *add_nth_iff_card*:
assumes ($\bigwedge$*xs*. *xs* ∈ *X* ⟹ *fo_nmlz AD xs* = *xs*) ($\bigwedge$*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*⟩

lemma *set_fo_nmlz_add_nth_rem_nth*:
assumes *j* < *length xs* $\bigwedge$*x*. *x* ∈ *X* ⟹ *fo_nmlz AD x* = *x*
$\bigwedge$*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*)}
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *fo_rep*: *fo_wf* φ *I t* ⟹ *fo_rep t* = *proj_sat* φ *I*
⟨*proof*⟩

global_interpretation *Ailamazyan*: *eval_fo fo_wf eval_pred fo_rep fo_res*
eval_bool eval_eq eval_neg eval_conj eval_ajoin eval_disj
eval_exists eval_forall
defines *eval_fmla* = *Ailamazyan.eval_fmla*
and *eval* = *Ailamazyan.eval*
⟨*proof*⟩

definition *esat_UNIV* :: (′*a* :: *infinite*, ′*b*) *fo_fmla* ⇒ (′*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*⟩

lemma *sat_code*[*code*]:
  fixes φ :: (*'a* :: *infinite*, *'b*) *fo_fmla*
  shows *sat* φ *I* $\sigma \longleftrightarrow$ (*if wf_fo_intp* φ *I then*
  (*case eval_fmla* φ *I of* (*AD*, *n*, *X*) $\Rightarrow$
    fo_nmlz (*act_edom* φ *I*) (*map* (*Inl* $\circ$ σ) (*fv_fo_fmla_list* φ)) $\in$ *X*)
  else sat φ *I* σ)
  ⟨*proof*⟩

end
theory *Ailamazyan_Code*
  imports *HOL−Library.Code_Target_Nat Containers.Containers Ailamazyan*
begin

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

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

lemma *convert_db_rec_mono*: *Mapping.lookup m* (*r*, *n*) = *Some tss* $\Longrightarrow$
  $\exists$ *tss'*. *Mapping.lookup* (*convert_db_rec ktss m*) (*r*, *n*) = *Some tss'* $\wedge$ *tss* $\subseteq$ *tss'*
  ⟨*proof*⟩

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

lemma *convert_db_rec_complete*: *Mapping.lookup* (*convert_db_rec ktss m*) (*r*, *n*) = *Some tss'* $\Longrightarrow$
  ts $\in$ *tss'* $\Longrightarrow$
  (*length ts* = *n* $\wedge$ (*r*, *ts*) $\in$ *set ktss*) $\vee$ ($\exists$ *tss*. *Mapping.lookup m* (*r*, *n*) = *Some tss* $\wedge$ *ts* $\in$ *tss*)
  ⟨*proof*⟩

definition *convert_db* :: (*'a* $\times$ *'c list*) *list* $\Rightarrow$ (*'c table*, *'a*) *fo_intp* **where**
  convert_db ktss = (*let m* = *convert_db_rec ktss Mapping.empty in*
    (λ*x*. *case Mapping.lookup m x of None* $\Rightarrow$ {} | *Some v* $\Rightarrow$ *v*))

lemma *convert_db_correct*: (*ts* $\in$ *convert_db ktss* (*r*, *n*) $\longrightarrow$ *n* = *length ts*) $\wedge$
  ((*r*, *ts*) $\in$ *set ktss* $\longleftrightarrow$ *ts* $\in$ *convert_db ktss* (*r*, *length ts*))
  ⟨*proof*⟩

lemma *Inl_vimage_set_code*[*code_unfold*]: *Inl* −‘ *set as* = *set* (*List.map_filter* (*case_sum Some Map.empty*) *as*)
  ⟨*proof*⟩

lemma *Inr_vimage_set_code*[*code_unfold*]: *Inr* −‘ *set as* = *set* (*List.map_filter* (*case_sum Map.empty Some*) *as*)
  ⟨*proof*⟩

lemma *Inl_vimage_code*: *Inl* −‘ *as* = *projl* ‘ {*x* $\in$ *as*. *isl x*}
  ⟨*proof*⟩

lemmas *ad_pred_code*[*code*] = *ad_terms.simps*[*unfolded Inl_vimage_code*]
lemmas *fo_wf_code*[*code*] = *fo_wf.simps*[*unfolded Inl_vimage_code*]

definition *empty_J* :: ((*nat*, *nat*) *fo_t*, *String.literal*) *fo_intp* **where**
empty_J = (λ(_, *n*). *eval_pred* (*map Var* [*0*..<*n*]) {})

definition *eval_fin_nat* :: (*nat*, *String.literal*) *fo_fmla* ⇒ (*nat table*, *String.literal*) *fo_intp* ⇒ *nat eval_res* **where**
eval_fin_nat φ *I* = *eval* φ *I*

definition *sat_fin_nat* :: (*nat*, *String.literal*) *fo_fmla* ⇒ (*nat table*, *String.literal*) *fo_intp* ⇒ *nat val* ⇒ *bool* **where**
sat_fin_nat φ *I* = *sat* φ *I*

definition *convert_nat_db* :: (*String.literal* × *nat list*) *list* ⇒
(*nat table*, *String.literal*) *fo_intp* **where**
convert_nat_db = *convert_db*

definition *rbt_nat_fold* :: _ ⇒ *nat set_rbt* ⇒ _ ⇒ _ **where**
rbt_nat_fold = *RBT_Set2.fold*

definition *rbt_nat_list_fold* :: _ ⇒ (*nat list*) *set_rbt* ⇒ _ ⇒ _ **where**
rbt_nat_list_fold = *RBT_Set2.fold*

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

export_code *eval_fin_nat sat_fin_nat fv_fo_fmla_list convert_nat_db rbt_nat_fold rbt_nat_list_fold rbt_sum_list_fold Const Conj Inl Fin nat_of_integer integer_of_nat RBT_set*
in *OCaml* **module_name** *Eval_FO* **file_prefix** *verified*

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

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

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

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

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

$(String.implode\ ''Q'',\ [20,\ 42]),$
$(String.implode\ ''Q'',\ [30,\ 43])])$

end

References

[1] A. K. Ailamazyan, M. M. Gilula, A. P. Stolboushkin, and G. F. Schwartz. Reduction of a relational model with infinite domains to the case of finite domains. *Dokl. Akad. Nauk SSSR*, 286:308–311, 1986.

[2] A. Avron and Y. Hirshfeld. On first order database query languages. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 226–231. IEEE Computer Society, 1991.

[3] M. Y. Vardi. The complexity of relational query languages (extended abstract). In H. R. Lewis, B. B. Simons, W. A. Burkhard, and L. H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982.