First-Order Query Evaluation

Martin Raszyk

March 24, 2023

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 *Infinite*
imports *Main*
begin

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

lemma *arb_element*: *finite* $Y \Longrightarrow \exists\, x :: 'a.\ x \notin Y$
⟨*proof*⟩

lemma *arb_finite_subset*: *finite* $Y \Longrightarrow \exists\, X :: 'a\ set.\ Y \cap X = \{\} \wedge$ *finite* $X \wedge n \leq$ *card* X
⟨*proof*⟩

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

end

instance *nat* :: *infinite*
⟨*proof*⟩

end
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*: (⋀*n*. *n* ∈ *fv_fo_term_set t* ⟹ *σ n* = *σ′ n*) ⟹
eval_term σ t = *eval_term σ′ t*
⟨*proof*⟩

lemma *eval_terms_fv_fo_terms_set*: *σ* ⊙ *ts* = *σ′* ⊙ *ts* ⟹ *n* ∈ *fv_fo_terms_set ts* ⟹ *σ n* = *σ′ n*
⟨*proof*⟩

lemma *eval_terms_cong*: (⋀*n*. *n* ∈ *fv_fo_terms_set ts* ⟹ *σ n* = *σ′ n*) ⟹
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* ⇒ *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* (λ*m*. *n* ≠ *m*) (*fv_fo_fmla_list_rec φ*)
| *fv_fo_fmla_list_rec* (*Forall n φ*) = *filter* (λ*m*. *n* ≠ *m*) (*fv_fo_fmla_list_rec φ*)

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

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

lemma *finite_fv_fo_fmla*: *finite* (*fv_fo_fmla* φ)
⟨*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 ⟹ *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 ⟹ σ *n* = σ' *n*) ⟹

```
  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 = (λσ. map σ (fv_fo_fmla_list φ)) ` {σ. sat φ I σ}

end
theory Eval_FO
  imports Infinite FO
begin

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

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

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

lemma *eval_fmla_correct*:
fixes φ :: ('a :: *infinite*, 'b) *fo_fmla*
assumes *wf_fo_intp* φ *I*
shows *wf* φ *I* (*eval_fmla* φ *I*)
⟨*proof*⟩

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

lemma *eval_fmla_proj_sat*:
fixes φ :: ('a :: *infinite*, 'b) *fo_fmla*
assumes *wf_fo_intp* φ *I*
shows *rep* (*eval_fmla* φ *I*) = *proj_sat* φ *I*
⟨*proof*⟩

lemma *eval_sound*:
fixes φ :: ('a :: *infinite*, 'b) *fo_fmla*
assumes *eval* φ *I* = *Fin Z*
shows *Z* = *proj_sat* φ *I*
⟨*proof*⟩

lemma *eval_complete*:
fixes φ :: ('a :: *infinite*, 'b) *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*))
⟨*proof*⟩

end

lift_definition *mapping_antijoin* :: ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping* **is**
λ*m m*' *x*. *case m x of None* ⇒ *None* | *Some y* ⇒ (*case m*' *x of None* ⇒ *Some y* | *Some y*' ⇒ *None*)
⟨*proof*⟩

lemma *mapping_antijoin_code*[*code*]:
fixes $t :: ('a :: ccompare, 'b)\ mapping\_rbt$
shows *mapping_antijoin* (*RBT_Mapping t*) (*RBT_Mapping u*) =
(*case ID CCOMPARE*('*a*) *of None* ⇒ *Code.abort* (*STR* ''*mapping_antijoin RBT_Mapping*: *ccompare* = *None*'') (λ_. *mapping_antijoin* (*RBT_Mapping t*) (*RBT_Mapping u*))
| *Some* _ ⇒ *RBT_Mapping* (*diff t u*))
⟨*proof*⟩

end
theory *Cluster*
imports *Mapping_Code*
begin

lemma *these_Un*[*simp*]: *Option.these* (*A* ∪ *B*) = *Option.these A* ∪ *Option.these B*
⟨*proof*⟩

lemma *these_insert*[*simp*]: *Option.these* (*insert x A*) = (*case x of Some a* ⇒ *insert a* | *None* ⇒ *id*) (*Option.these A*)
⟨*proof*⟩

lemma *these_image_Un*[*simp*]: *Option.these* (*f* ‘ (*A* ∪ *B*)) = *Option.these* (*f* ‘ *A*) ∪ *Option.these* (*f* ‘ *B*)
⟨*proof*⟩

lemma *these_imageI*: *f x* = *Some y* ⟹ *x* ∈ *X* ⟹ *y* ∈ *Option.these* (*f* ‘ *X*)
⟨*proof*⟩

lift_definition *cluster* :: ('*b* ⇒ '*a option*) ⇒ '*b set* ⇒ ('*a*, '*b set*) *mapping* **is**

$\lambda f\ Y\ x.$ *if Some* $x \in f$ ‘ Y *then Some* $\{y \in Y.\ f\ y = \mathit{Some}\ x\}$ *else None* ⟨*proof*⟩

lemma *set_of_idx_cluster*: *set_of_idx* (*cluster* (*Some* ∘ f) X) = X
⟨*proof*⟩

lemma *lookup_cluster′*: *Mapping.lookup* (*cluster* (*Some* ∘ h) X) y = (*if* $y \notin h$ ‘ X *then None else Some* $\{x \in X.\ h\ x = y\}$)
⟨*proof*⟩

context *ord*
begin

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

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

definition *cluster_rbt* :: $('b \Rightarrow 'a\ \mathit{option}) \Rightarrow ('b, \mathit{unit})\ \mathit{rbt} \Rightarrow ('a, 'b\ \mathit{set})\ \mathit{rbt}$ **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* ⇒ $\{b\} \mid$ *Some* $Y \Rightarrow$ *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 ≡ (λb _ t. case f b of Some a ⇒ add_to_rbt_comp (a, b) t
| None ⇒ t)

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

context
  assumes c: comparator c
begin

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

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

end

end

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

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

end
theory Ailamazyan
  imports Eval_FO Cluster Mapping_Code
begin

fun SP :: ('a, 'b) fo_fmla ⇒ nat set where
  SP (Eqa (Var n) (Var n')) = (if n ≠ n' then {n, n'} else {})
| SP (Neg φ) = SP φ
```

| $SP\ (Conj\ \varphi\ \psi) = SP\ \varphi \cup SP\ \psi$
| $SP\ (Disj\ \varphi\ \psi) = SP\ \varphi \cup SP\ \psi$
| $SP\ (Exists\ n\ \varphi) = SP\ \varphi - \{n\}$
| $SP\ (Forall\ n\ \varphi) = SP\ \varphi - \{n\}$
| $SP\ \_ = \{\}$

lemma *SP_fv*: $SP\ \varphi \subseteq fv\_fo\_fmla\ \varphi$
⟨*proof*⟩

lemma *finite_SP*: $finite\ (SP\ \varphi)$
⟨*proof*⟩

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

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

lemma *SP_list_set*: $set\ (SP\_list\ \varphi) = SP\ \varphi$
⟨*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** $\cdot e\ 60$) **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** $\odot e\ 60$) **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**
$sp\_equiv\ \sigma\ \tau\ I \longleftrightarrow pairwise\ sp\_equiv\_pair\ ((\lambda n.\ (\sigma\ n, \tau\ n))\ `\ I)$

lemma *sp_equiv_mono*: $I \subseteq J \Longrightarrow sp\_equiv\ \sigma\ \tau\ J \Longrightarrow sp\_equiv\ \sigma\ \tau\ I$
⟨*proof*⟩

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

lemma *ad_agr_sets_comm*: $ad\_agr\_sets\ FV\ S\ X\ \sigma\ \tau \Longrightarrow ad\_agr\_sets\ FV\ S\ X\ \tau\ \sigma$
⟨*proof*⟩

lemma *ad_agr_sets_mono*: $X \subseteq Y \Longrightarrow ad\_agr\_sets\ FV\ S\ Y\ \sigma\ \tau \Longrightarrow ad\_agr\_sets\ FV\ S\ X\ \sigma\ \tau$
⟨*proof*⟩

lemma *ad_agr_sets_mono'*: $S \subseteq S' \Longrightarrow ad\_agr\_sets\ FV\ S'\ X\ \sigma\ \tau \Longrightarrow ad\_agr\_sets\ FV\ S\ X\ \sigma\ \tau$
⟨*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* σ *ns*) (*map* τ *ns*) $\longleftrightarrow$ *sp_equiv* σ τ (*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* σ *ns*) (*map* σ' *ns*)
shows *sp_equiv_list* (*map* σ *ms*) (*map* σ' *ms*)
⟨*proof*⟩

lemma *ad_agr_list_subset*: *set ms* $\subseteq$ *set ns* $\Longrightarrow$ *ad_agr_list X* (*map* σ *ns*) (*map* σ' *ns*) $\Longrightarrow$
ad_agr_list X (*map* σ *ms*) (*map* σ' *ms*)
⟨*proof*⟩

lemma *ad_agr_list_link*: *ad_agr_sets* (*set ns*) (*set ns*) *AD* σ τ $\longleftrightarrow$
ad_agr_list AD (*map* σ *ns*) (*map* τ *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 φ *X* σ τ $\longleftrightarrow$ *ad_agr_sets* (*fv_fo_fmla* φ) (*SP* φ) *X* σ τ

lemma *ad_agr_sets_restrict*:
ad_agr_sets (*set* (*fv_fo_fmla_list* φ)) (*set* (*fv_fo_fmla_list* φ)) *AD* σ τ $\Longrightarrow$ *ad_agr* φ *AD* σ τ
⟨*proof*⟩

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

lemma *ex_out*:
assumes *finite X*
shows $\exists\, k.\ k \notin X \wedge k <$ *Suc* (*card X*)
⟨*proof*⟩

lemma *extend_*τ:
assumes *ad_agr_sets* (*FV* $-$ {*n*}) (*S* $-$ {*n*}) *X* σ τ *S* $\subseteq$ *FV finite S* τ ` (*FV* $-$ {*n*}) $\subseteq$ *Z*
Inl ` *X* $\cup$ *Inr* ` {..<*max* 1 (*card* (*Inr* $-`$ $\tau$ ` (*S* $-$ {*n*})) + (*if n* $\in$ *S then* 1 *else* 0))} $\subseteq$ *Z*
shows $\exists\, k \in Z.$ *ad_agr_sets FV S X* (σ(*n* := *x*)) (τ(*n* := *k*))
⟨*proof*⟩

lemma *esat_Pred*:
assumes *ad_agr_sets FV S* ($\bigcup$(*set* ` *X*)) $\sigma$ $\tau$ *fv_fo_terms_set ts* $\subseteq$ *FV* $\sigma$ $\odot e$ *ts* $\in$ *map Inl* ` *X*
t $\in$ *set ts*
shows σ $\cdot e$ *t* = τ $\cdot e$ *t*
⟨*proof*⟩

lemma *sp_equiv_list_fv*:
assumes ($\bigwedge i.$ *i* $\in$ *fv_fo_terms_set ts* $\Longrightarrow$ *ad_equiv_pair X* (σ *i*, τ *i*))
$\bigcup$(*set_fo_term* ` *set ts*) $\subseteq$ *X sp_equiv* σ τ (*fv_fo_terms_set ts*)
shows *sp_equiv_list* (*map* ((·*e*) σ) *ts*) (*map* ((·*e*) τ) *ts*)
⟨*proof*⟩

lemma *esat_Pred_inf*:
assumes *fv_fo_terms_set ts* $\subseteq$ *FV fv_fo_terms_set ts* $\subseteq$ *S*
ad_agr_sets FV S AD σ τ *ad_agr_list AD* (σ $\odot e$ *ts*) *vs*
$\bigcup$(*set_fo_term* ` *set ts*) $\subseteq$ *AD*
shows *ad_agr_list AD* (τ $\odot e$ *ts*) *vs*
⟨*proof*⟩

type_synonym ('*a*, '*c*) *fo_t* = '*a set* $\times$ *nat* $\times$ ('*a* + '*c*) *table*

fun *esat* :: ('*a*, '*b*) *fo_fmla* $\Rightarrow$ ('*a table*, '*b*) *fo_intp* $\Rightarrow$ ('*a* + *nat*) *val* $\Rightarrow$ ('*a* + *nat*) *set* $\Rightarrow$ *bool* **where**
esat (*Pred r ts*) *I* σ *X* $\longleftrightarrow$ σ $\odot e$ *ts* $\in$ *map Inl* ` *I* (*r*, *length ts*)
| *esat* (*Bool b*) *I* σ *X* $\longleftrightarrow$ *b*
| *esat* (*Eqa t t'*) *I* σ *X* $\longleftrightarrow$ σ $\cdot e$ *t* = σ $\cdot e$ *t'*

| *esat* (*Neg* φ) $I\ \sigma\ X \longleftrightarrow \neg$ *esat* $\varphi\ I\ \sigma\ X$
| *esat* (*Conj* $\varphi\ \psi$) $I\ \sigma\ X \longleftrightarrow$ *esat* $\varphi\ I\ \sigma\ X \wedge$ *esat* $\psi\ I\ \sigma\ X$
| *esat* (*Disj* $\varphi\ \psi$) $I\ \sigma\ X \longleftrightarrow$ *esat* $\varphi\ I\ \sigma\ X \vee$ *esat* $\psi\ I\ \sigma\ X$
| *esat* (*Exists* $n\ \varphi$) $I\ \sigma\ X \longleftrightarrow (\exists\, x \in X.$ *esat* $\varphi\ I\ (\sigma(n := x))\ X)$
| *esat* (*Forall* $n\ \varphi$) $I\ \sigma\ X \longleftrightarrow (\forall\, x \in X.$ *esat* $\varphi\ I\ (\sigma(n := x))\ X)$

fun *sz_fmla* :: ('*a*, '*b*) *fo_fmla* ⇒ *nat* **where**
  sz_fmla (*Neg* φ) = *Suc* (*sz_fmla* φ)
| *sz_fmla* (*Conj* $\varphi\ \psi$) = *Suc* (*sz_fmla* φ + *sz_fmla* ψ)
| *sz_fmla* (*Disj* $\varphi\ \psi$) = *Suc* (*sz_fmla* φ + *sz_fmla* ψ)
| *sz_fmla* (*Exists* $n\ \varphi$) = *Suc* (*sz_fmla* φ)
| *sz_fmla* (*Forall* $n\ \varphi$) = *Suc* (*Suc* (*Suc* (*Suc* (*sz_fmla* φ))))
| *sz_fmla* _ = 0

lemma *sz_fmla_induct*[*case_names Pred Bool Eqa Neg Conj Disj Exists Forall*]:
  $(\bigwedge r\ ts.\ P$ (*Pred* $r\ ts$)) ⟹ $(\bigwedge b.\ P$ (*Bool* b)) ⟹
  $(\bigwedge t\ t'.\ P$ (*Eqa* $t\ t'$)) ⟹ $(\bigwedge \varphi.\ P\ \varphi \Longrightarrow P$ (*Neg* φ)) ⟹
  $(\bigwedge \varphi\ \psi.\ P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P$ (*Conj* $\varphi\ \psi$)) ⟹ $(\bigwedge \varphi\ \psi.\ P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P$ (*Disj* $\varphi\ \psi$)) ⟹
  $(\bigwedge n\ \varphi.\ P\ \varphi \Longrightarrow P$ (*Exists* $n\ \varphi$)) ⟹ $(\bigwedge n\ \varphi.\ P$ (*Exists* n (*Neg* φ)) ⟹ P (*Forall* $n\ \varphi$)) ⟹ $P\ \varphi$
⟨*proof*⟩

lemma *esat_fv_cong*: $(\bigwedge n.\ n \in$ *fv_fo_fmla* $\varphi \Longrightarrow \sigma\ n = \sigma'\ n) \Longrightarrow$ *esat* $\varphi\ I\ \sigma\ X \longleftrightarrow$ *esat* $\varphi\ I\ \sigma'\ X$
⟨*proof*⟩

fun *ad_terms* :: ('*a fo_term*) *list* ⇒ '*a set* **where**
  ad_terms $ts = \bigcup$(*set* (*map set_fo_term* ts))

fun *act_edom* :: ('*a*, '*b*) *fo_fmla* ⇒ ('*a table*, '*b*) *fo_intp* ⇒ '*a set* **where**
  act_edom (*Pred* $r\ ts$) I = *ad_terms* $ts \cup \bigcup$(*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* φ) 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* ⇒ '*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* $\varphi\ I \subseteq AD\ d\ \varphi \leq n$
    τ ‘ *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 σ :: *nat* ⇒ '*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* σ (*fv_fo_fmla_list* φ)) (*map* τ (*fv_fo_fmla_list* φ))
act_edom φ *I* $\subseteq$ *AD*
shows *esat* φ *I* σ *UNIV* $\longleftrightarrow$ *esat* φ *I* τ *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 φ :: ('*a* :: *infinite*, '*b*) *fo_fmla*
assumes *fin*: *wf_fo_intp* φ *I*
shows *sat* φ *I* σ $\longleftrightarrow$ *esat* φ *I* (*Inl* $\circ$ σ :: *nat* $\Rightarrow$ '*a* + *nat*) *UNIV*
⟨*proof*⟩

lemma *sat_ad_agr_list*:
fixes φ :: ('*a* :: *infinite*, '*b*) *fo_fmla*
and *J* :: (('*a*, *nat*) *fo_t*, '*b*) *fo_intp*
assumes *wf_fo_intp* φ *I*
ad_agr_list AD (*map* (*Inl* $\circ$ σ :: *nat* $\Rightarrow$ '*a* + *nat*) (*fv_fo_fmla_list* φ))
(*map* (*Inl* $\circ$ τ) (*fv_fo_fmla_list* φ)) *act_edom* φ *I* $\subseteq$ *AD*
shows *sat* φ *I* σ $\longleftrightarrow$ *sat* φ *I* τ
⟨*proof*⟩

definition *nfv* :: ('*a*, '*b*) *fo_fmla* $\Rightarrow$ *nat* **where**
nfv φ = *length* (*fv_fo_fmla_list* φ)

lemma *nfv_card*: *nfv* φ = *card* (*fv_fo_fmla* φ)
⟨*proof*⟩

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

lemma *filter_rremdups_filter*: *filter P* (*rremdups* (*filter Q xs*)) =
rremdups (*filter* (λ*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* ⇒ *bool* **where**
nats ns = (*ns* = [*0*..<*length ns*])

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

lemma *fo_nmlzd_all_AD*:
assumes *set xs* ⊆ *Inl* ‘ *AD*
shows *fo_nmlzd AD xs*
⟨*proof*⟩

lemma *card_Inr_vimage_le_length*: *card* (*Inr* −‘ *set xs*) ≤ *length xs*
⟨*proof*⟩

lemma *fo_nmlzd_set*:
assumes *fo_nmlzd AD xs*
shows *set xs* = *set xs* ∩ *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*) = ⋃((λ*as*. (λ*x*. *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* ⟹ *a* ∈ *set xs*
⟨*proof*⟩

lemma *pos_length*: *pos a xs* = *Some i* ⟹ *i* < *length xs*
⟨*proof*⟩

lemma *pos_sound*: *pos a xs* = *Some i* ⟹ *i* < *length xs* ∧ *xs* ! *i* = *a*
⟨*proof*⟩

lemma *pos_complete*: *pos a xs* = *None* ⟹ *a* ∉ *set xs*
⟨*proof*⟩

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

lemma *rem_nth_length*: *i* < *length xs* ⟹ *length* (*rem_nth i xs*) = *length xs* − *1*
⟨*proof*⟩

lemma *rem_nth_take_drop*: *i* < *length xs* ⟹ *rem_nth i xs* = *take i xs* @ *drop* (*Suc i*) *xs*
⟨*proof*⟩

lemma *rem_nth_sound*: *distinct xs* ⟹ *pos n xs* = *Some i* ⟹
rem_nth i (*map σ xs*) = *map σ* (*filter* ((≠) *n*) *xs*)
⟨*proof*⟩

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

lemma *add_nth_length*: *i* ≤ *length zs* ⟹ *length* (*add_nth i z zs*) = *Suc* (*length zs*)
⟨*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**
$merge\ []\ mys = mys$
| $merge\ nxs\ [] = nxs$
| $merge\ ((n, x)\ \#\ nxs)\ ((m, y)\ \#\ mys) =$
$(if\ n \leq m\ then\ (n, x)\ \#\ merge\ nxs\ ((m, y)\ \#\ mys)$
$else\ (m, y)\ \#\ merge\ ((n, x)\ \#\ nxs)\ mys)$

lemma *merge_Nil2*[*simp*]: $merge\ nxs\ [] = nxs$
⟨*proof*⟩

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

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

lemma *insort_aux_gt*: $\forall x{\in}set\ nxs.\ n \leq fst\ x \Longrightarrow \forall x{\in}set\ mys.\ m \leq fst\ x \Longrightarrow \neg\ n \leq m \Longrightarrow$
$insort\ n\ (sort\ (map\ fst\ nxs\ @\ m\ \#\ map\ fst\ mys)) =$
$m\ \#\ insort\ n\ (sort\ (map\ fst\ nxs\ @\ map\ fst\ mys))$
⟨*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* (($\leq$) *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 = (λ*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 (λ*n. n < i) ns)* $\subseteq$ *{..<i}* $\Longrightarrow$ *filter* (($\leq$) *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)}*
⟨*proof*⟩

lemma *fo_nmlz_rec_set_rev*: *set (fo_nmlz_rec i m AD xs)* $\subseteq$ *Inl ‘ AD* $\Longrightarrow$ *set xs* $\subseteq$ *Inl ‘ AD*
⟨*proof*⟩

lemma *fo_nmlz_rec_map*: *inj_on m (dom m)* $\Longrightarrow$ *ran m* $\subseteq$ *{..<i}* $\Longrightarrow$ $\exists$*m′. inj_on m′ (dom m′)* $\wedge$
(∀*n. m n* $\neq$ *None* $\longrightarrow$ *m′ n = m n)* $\wedge$ (∀(*x, y*) $\in$ *set (zip xs (fo_nmlz_rec i m AD xs)).*
(case x of Inl x′ $\Rightarrow$ *if x′* $\in$ *AD then x = y else* $\exists$*j. m′ (Inl x′) = Some j* $\wedge$ *y = Inr j*
| Inr n $\Rightarrow$ $\exists$*j. m′ (Inr n) = Some j* $\wedge$ *y = Inr j))*
⟨*proof*⟩

lemma *ad_agr_map*:
assumes *length xs = length ys inj_on m (dom m)*
$\bigwedge$*x y.* (*x, y*) $\in$ *set (zip xs ys)* $\Longrightarrow$ *(case x of Inl x′* $\Rightarrow$
if x′ $\in$ *AD then x = y else m x = Some y* $\wedge$ *(case y of Inl z* $\Rightarrow$ *z* $\notin$ *AD | Inr _* $\Rightarrow$ *True)*
| Inr n $\Rightarrow$ *m x = Some y* $\wedge$ *(case y of Inl z* $\Rightarrow$ *z* $\notin$ *AD | Inr _* $\Rightarrow$ *True))*
shows *ad_agr_list AD xs ys*
⟨*proof*⟩

lemma *fo_nmlz_rec_take*: *take n (fo_nmlz_rec i m AD xs) = fo_nmlz_rec i m AD (take n xs)*
⟨*proof*⟩

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

lemma *fo_nmlz_Nil*[*simp*]: *fo_nmlz AD* [] = []
⟨*proof*⟩

lemma *fo_nmlz_Cons*: *fo_nmlz AD* [*x*] =
(case x of Inl x $\Rightarrow$ *if x* $\in$ *AD then* [*Inl x*] *else* [*Inr 0*] | _ $\Rightarrow$ [*Inr 0*])
⟨*proof*⟩

lemma *fo_nmlz_Cons_Cons*: *fo_nmlz AD* [*x, x*] =
(case x of Inl x $\Rightarrow$ *if x* $\in$ *AD then* [*Inl x, Inl x*] *else* [*Inr 0, Inr 0*] | _ $\Rightarrow$ [*Inr 0, Inr 0*])
⟨*proof*⟩

lemma *fo_nmlz_sound*: *fo_nmlzd AD* (*fo_nmlz AD xs*)
⟨*proof*⟩

lemma *fo_nmlz_length*: *length* (*fo_nmlz AD xs*) = *length xs*
⟨*proof*⟩

lemma *fo_nmlz_map*: ∃ *τ*. *fo_nmlz AD* (*map σ ns*) = *map τ ns*
⟨*proof*⟩

lemma *card_set_minus*: *card* (*set xs* − *X*) ≤ *length xs*
⟨*proof*⟩

lemma *fo_nmlz_set*: *set* (*fo_nmlz AD xs*) =
set xs ∩ *Inl* ‘ *AD* ∪ *Inr* ‘ {..<*min* (*length xs*) (*card* (*set xs* − *Inl* ‘ *AD*))}
⟨*proof*⟩

lemma *fo_nmlz_set_rev*: *set* (*fo_nmlz AD xs*) ⊆ *Inl* ‘ *AD* ⟹ *set xs* ⊆ *Inl* ‘ *AD*
⟨*proof*⟩

lemma *inj_on_empty*: *inj_on Map.empty* (*dom Map.empty*) **and** *ran_empty_upto*: *ran Map.empty* ⊆ {..<*0*}
⟨*proof*⟩

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

lemma *fo_nmlzd_mono*: *Inl* −‘ *set xs* ⊆ *AD* ⟹ *fo_nmlzd AD′ xs* ⟹ *fo_nmlzd AD xs*
⟨*proof*⟩

lemma *fo_nmlz_idem*: *fo_nmlzd AD ys* ⟹ *fo_nmlz AD ys* = *ys*
⟨*proof*⟩

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

fun *nall_tuples_rec* :: *′a set* ⇒ *nat* ⇒ *nat* ⇒ (*′a* + *nat*) *table* **where**
nall_tuples_rec AD i 0 = {[]}
| *nall_tuples_rec AD i* (*Suc n*) = ⋃((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* ⟹ *Inl* −‘ *set vs* ⊆ *AD*
⟨*proof*⟩

lemma *nall_tuples_rec_length*: *xs* ∈ *nall_tuples_rec AD i n* ⟹ *length xs* = *n*
⟨*proof*⟩

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

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

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

lemma *nall_tuples_rec_fo_nmlz_rec_complete*:

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

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

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

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

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

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

lemma *list_induct2_rev*[*consumes 1*]: *length xs = length ys ⟹ (P [] []) ⟹*
(⋀x y xs ys. P xs ys ⟹ P (xs @ [x]) (ys @ [y])) ⟹ P xs ys
⟨*proof*⟩

lemma *ad_agr_list_fo_nmlzd*:
assumes *ad_agr_list AD vs vs' fo_nmlzd AD vs fo_nmlzd AD vs'*
shows *vs = vs'*
⟨*proof*⟩

lemma *fo_nmlz_eqI*:
assumes *ad_agr_list AD vs vs'*
shows *fo_nmlz AD vs = fo_nmlz AD vs'*
⟨*proof*⟩

lemma *fo_nmlz_eqD*:
assumes *fo_nmlz AD vs = fo_nmlz AD vs'*
shows *ad_agr_list AD vs vs'*
⟨*proof*⟩

lemma *fo_nmlz_eq*: *fo_nmlz AD vs = fo_nmlz AD vs' ⟷ ad_agr_list AD vs vs'*
⟨*proof*⟩

lemma *fo_nmlz_mono*:
assumes *AD ⊆ AD' Inl −‘ set xs ⊆ AD*
shows *fo_nmlz AD' xs = fo_nmlz AD xs*
⟨*proof*⟩

definition *proj_vals* :: *'c val set ⇒ nat list ⇒ 'c table* **where**
proj_vals R ns = (λτ. map τ ns) ‘ R

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

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

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

definition *ext_tuple* :: *'a set* ⇒ *nat list* ⇒ *nat list* ⇒
('a + nat) list ⇒ *('a + nat) list set* **where**
ext_tuple AD fv_sub fv_sub_comp as = (*if fv_sub_comp* = [] *then* {*as*}
else (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* ⟹
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*))
⟨*proof*⟩

lemma *map_map_of*: *length xs* = *length ys* ⟹ *distinct xs* ⟹
ys = *map* (*the* ∘ (*map_of* (*zip xs ys*))) *xs*
⟨*proof*⟩

lemma *id_map_empty*: *id_map 0* = *Map.empty*
⟨*proof*⟩

lemma *fo_nmlz_rec_shift*:
fixes *xs* :: *('a + nat) list*
shows *fo_nmlz_rec i* (*id_map i*) *AD xs* = *xs* ⟹
i' = *card* (*Inr* −‘ (*Inr* ‘ {..<*i*} ∪ *set* (*take n xs*))) ⟹ *n* ≤ *length xs* ⟹
fo_nmlz_rec i' (*id_map i'*) *AD* (*drop n xs*) = *drop n xs*
⟨*proof*⟩

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 ∩ 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
shows *ext_tuple_set AD fv_sub fv_sub_comp ass = fo_nmlz AD ‘ proj_vals R fv_all*
⟨*proof*⟩

lemma *proj_tuple_sound*:
assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
set fv_sub ∩ set fv_sub_comp = {} set fv_sub ∪ set fv_sub_comp = set fv_all
ass = fo_nmlz AD ‘ proj_vals R fv_sub
⋀σ τ. ad_agr_sets (set fv_sub) (set fv_sub) AD σ τ ⟹ σ ∈ R ⟷ τ ∈ R
fo_nmlz AD xs = xs length xs = length fv_all
fo_nmlz AD (proj_tuple fv_sub (zip fv_all xs)) ∈ ass
shows *xs ∈ fo_nmlz AD ‘ ⋃(ext_tuple AD fv_sub fv_sub_comp ‘ ass)*
⟨*proof*⟩

lemma *proj_tuple_correct*:
assumes *sorted_distinct fv_sub sorted_distinct fv_sub_comp sorted_distinct fv_all*
set fv_sub ∩ set fv_sub_comp = {} set fv_sub ∪ set fv_sub_comp = set fv_all
ass = fo_nmlz AD ‘ proj_vals R fv_sub
⋀σ τ. ad_agr_sets (set fv_sub) (set fv_sub) AD σ τ ⟹ σ ∈ R ⟷ τ ∈ R
fo_nmlz AD xs = xs length xs = length fv_all
shows *xs ∈ fo_nmlz AD ‘ ⋃(ext_tuple AD fv_sub fv_sub_comp ‘ ass) ⟷*
fo_nmlz AD (proj_tuple fv_sub (zip fv_all xs)) ∈ ass
⟨*proof*⟩

fun *unify_vals_terms* :: *('a + 'c) list ⇒ ('a fo_term) list ⇒ (nat ⇀ ('a + 'c)) ⇒ (nat ⇀ ('a + 'c)) option* **where**
unify_vals_terms [] [] σ = Some σ
| *unify_vals_terms (v # vs) ((Const c′) # ts) σ =*
(if v = Inl c′ then unify_vals_terms vs ts σ else None)
| *unify_vals_terms (v # vs) ((Var n) # ts) σ =*
(case σ n of Some x ⇒ (if v = x then unify_vals_terms vs ts σ else None)

```
    | None ⇒ unify_vals_terms vs ts (σ(n := Some v)))
| unify_vals_terms _ _ _ = None

lemma unify_vals_terms_extends: unify_vals_terms vs ts σ = Some σ' ⟹ extends_subst σ σ'
  ⟨proof⟩

lemma unify_vals_terms_sound: unify_vals_terms vs ts σ = Some σ' ⟹ (the ∘ σ') ⊙e ts = vs
  ⟨proof⟩

lemma unify_vals_terms_complete: σ'' ⊙e ts = vs ⟹ (⋀n. σ n ≠ None ⟹ σ n = Some (σ'' n)) ⟹
  ∃σ'. unify_vals_terms vs ts σ = Some σ'
  ⟨proof⟩

definition eval_table :: 'a fo_term list ⇒ ('a + 'c) table ⇒ ('a + 'c) table where
  eval_table ts X = (let fvs = fv_fo_terms_list ts in
    ⋃((λvs. case unify_vals_terms vs ts Map.empty of Some σ ⇒
      {map (the ∘ σ) fvs} | _ ⇒ {}) ` X))

lemma eval_table:
  fixes X :: ('a + 'c) table
  shows eval_table ts X = proj_vals {σ. σ ⊙e ts ∈ X} (fv_fo_terms_list ts)
⟨proof⟩

fun ad_agr_close_rec :: nat ⇒ (nat ⇀ 'a + nat) ⇒ 'a set ⇒
  ('a + nat) list ⇒ ('a + nat) list set where
  ad_agr_close_rec i m AD [] = {[]}
| ad_agr_close_rec i m AD (Inl x # xs) = (λxs. Inl x # xs) ` ad_agr_close_rec i m AD xs
| ad_agr_close_rec i m AD (Inr n # xs) = (case m n of None ⇒ ⋃((λx. (λxs. Inl x # xs) `
    ad_agr_close_rec i (m(n := Some (Inl x))) (AD − {x}) xs) ` AD) ∪
    (λxs. Inr i # xs) ` ad_agr_close_rec (Suc i) (m(n := Some (Inr i))) AD xs
  | Some v ⇒ (λxs. v # xs) ` ad_agr_close_rec i m AD xs)

lemma ad_agr_close_rec_length: ys ∈ ad_agr_close_rec i m AD xs ⟹ length xs = length ys
  ⟨proof⟩

lemma ad_agr_close_rec_sound: ys ∈ ad_agr_close_rec i m AD xs ⟹
  fo_nmlz_rec j (id_map j) X xs = xs ⟹ X ∩ AD = {} ⟹ X ∩ Y = {} ⟹ Y ∩ AD = {} ⟹
  inj_on m (dom m) ⟹ dom m = {..<j} ⟹ ran m ⊆ Inl ` Y ∪ Inr ` {..<i} ⟹ i ≤ j ⟹
  fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) ys = ys ∧
  (∃m'. inj_on m' (dom m') ∧ (∀n v. m n = Some v ⟶ m' (Inr n) = Some v) ∧
  (∀(x, y) ∈ set (zip xs ys). case x of Inl x' ⇒
      if x' ∈ X then x = y else m' x = Some y ∧ (case y of Inl z ⇒ z ∉ X | Inr x ⇒ True)
  | Inr n ⇒ m' x = Some y ∧ (case y of Inl z ⇒ z ∉ X | Inr x ⇒ True)))
⟨proof⟩

lemma ad_agr_close_rec_complete:
  fixes xs :: ('a + nat) list
  shows fo_nmlz_rec j (id_map j) X xs = xs ⟹
  X ∩ AD = {} ⟹ X ∩ Y = {} ⟹ Y ∩ AD = {} ⟹
  inj_on m (dom m) ⟹ dom m = {..<j} ⟹ ran m = Inl ` Y ∪ Inr ` {..<i} ⟹ i ≤ j ⟹
  (⋀n b. (Inr n, b) ∈ set (zip xs ys) ⟹ case m n of Some v ⇒ v = b | None ⇒ b ∉ ran m) ⟹
  fo_nmlz_rec i (id_map i) (X ∪ Y ∪ AD) ys = ys ⟹ ad_agr_list X xs ys ⟹
  ys ∈ ad_agr_close_rec i m AD xs
⟨proof⟩

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

lemma *ad_agr_close_sound*:
assumes $ys \in ad\_agr\_close\ Y\ xs\ fo\_nmlzd\ X\ xs\ X \cap Y = \{\}$
shows $fo\_nmlzd\ (X \cup Y)\ ys \wedge ad\_agr\_list\ X\ xs\ ys$
⟨*proof*⟩

lemma *ad_agr_close_complete*:
assumes $X \cap Y = \{\}\ fo\_nmlzd\ X\ xs\ fo\_nmlzd\ (X \cup Y)\ ys\ ad\_agr\_list\ X\ xs\ ys$
shows $ys \in ad\_agr\_close\ Y\ xs$
⟨*proof*⟩

lemma *ad_agr_close_empty*: $fo\_nmlzd\ X\ xs \Longrightarrow ad\_agr\_close\ \{\}\ xs = \{xs\}$
⟨*proof*⟩

lemma *ad_agr_close_set_correct*:
assumes $AD' \subseteq AD\ sorted\_distinct\ ns$
$\bigwedge \sigma\ \tau.\ ad\_agr\_sets\ (set\ ns)\ (set\ ns)\ AD'\ \sigma\ \tau \Longrightarrow \sigma \in R \longleftrightarrow \tau \in R$
shows $\bigcup (ad\_agr\_close\ (AD - AD')\ `\ fo\_nmlz\ AD'\ `\ proj\_vals\ R\ ns) = fo\_nmlz\ AD\ `\ proj\_vals\ R\ ns$
⟨*proof*⟩

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 $\bigcup (ad\_agr\_close\ (AD - AD')\ `\ fo\_nmlz\ AD'\ `\ proj\_fmla\ \varphi\ R) = fo\_nmlz\ AD\ `\ proj\_fmla\ \varphi\ R$
⟨*proof*⟩

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

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 $eval\_conj\_tuple\ AD\ ns\varphi\ ns\psi\ xs\ ys =$

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

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

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

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

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

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

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

fun eval_forall :: nat ⇒ nat list ⇒ ('a, nat) fo_t ⇒ ('a, nat) fo_t where
  eval_forall i ns (AD, _, X) = (case pos i ns of Some j ⇒
    let n = card AD in
    (AD, length ns − 1, Mapping.keys (Mapping.filter (λt Z. n + card (Inr −‘ set t) + 1 ≤ card Z)
      (cluster (Some ∘ (λts. fo_nmlz AD (rem_nth j ts))) X)))
    | None ⇒ (AD, length ns, X))

lemma combine_map2: assumes length ys = length xs length ys' = length xs'
```

distinct xs distinct xs′ set xs ∩ set xs′ = {}
shows $\exists f.\ ys = map\ f\ xs \wedge ys' = map\ f\ xs'$
⟨*proof*⟩

lemma *combine_map3*: **assumes** *length ys = length xs length ys′ = length xs′ length ys″ = length xs″*
distinct xs distinct xs′ distinct xs″ set xs ∩ set xs′ = {} set xs ∩ set xs″ = {} set xs′ ∩ set xs″ = {}
shows $\exists f.\ ys = map\ f\ xs \wedge ys' = map\ f\ xs' \wedge ys'' = map\ f\ xs''$
⟨*proof*⟩

lemma *distinct_set_zip*: *length nsx = length xs ⟹ distinct nsx ⟹*
(a, b) ∈ set (zip nsx xs) ⟹ (a, ba) ∈ set (zip nsx xs) ⟹ b = ba
⟨*proof*⟩

lemma *fo_nmlz_idem_isl*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow (case\ x\ of\ Inl\ z \Rightarrow z \in X \mid \_ \Rightarrow False)$
shows *fo_nmlz X xs = xs*
⟨*proof*⟩

lemma *set_zip_mapI*: *x ∈ set xs ⟹ (f x, g x) ∈ set (zip (map f xs) (map g xs))*
⟨*proof*⟩

lemma *ad_agr_list_fo_nmlzd_isl*:
assumes *ad_agr_list X (map f xs) (map g xs) fo_nmlzd X (map f xs) x ∈ set xs isl (f x)*
shows *f x = g x*
⟨*proof*⟩

lemma *eval_conj_tuple_close_empty2*:
assumes *fo_nmlzd X xs fo_nmlzd Y ys*
length nsx = length xs length nsy = length ys
sorted_distinct nsx sorted_distinct nsy
sorted_distinct ns set ns ⊆ set nsx ∩ set nsy
fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsy ys)) ∨
(proj_tuple ns (zip nsx xs) ≠ proj_tuple ns (zip nsy ys) ∧
(∀ x ∈ set (proj_tuple ns (zip nsx xs)). isl x) ∧ (∀ y ∈ set (proj_tuple ns (zip nsy ys)). isl y))
xs′ ∈ ad_agr_close ((X ∪ Y) − X) xs ys′ ∈ ad_agr_close ((X ∪ Y) − Y) ys
shows *eval_conj_tuple (X ∪ Y) nsx nsy xs′ ys′ = {}*
⟨*proof*⟩

lemma *eval_conj_tuple_close_empty*:
assumes *fo_nmlzd X xs fo_nmlzd Y ys*
length nsx = length xs length nsy = length ys
sorted_distinct nsx sorted_distinct nsy
ns = filter (λn. n ∈ set nsy) nsx
fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz (X ∩ Y) (proj_tuple ns (zip nsy ys))
xs′ ∈ ad_agr_close ((X ∪ Y) − X) xs ys′ ∈ ad_agr_close ((X ∪ Y) − Y) ys
shows *eval_conj_tuple (X ∪ Y) nsx nsy xs′ ys′ = {}*
⟨*proof*⟩

lemma *eval_conj_tuple_empty2*:
assumes *fo_nmlzd Z xs fo_nmlzd Z ys*
length nsx = length xs length nsy = length ys
sorted_distinct nsx sorted_distinct nsy
sorted_distinct ns set ns ⊆ set nsx ∩ set nsy
fo_nmlz Z (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz Z (proj_tuple ns (zip nsy ys)) ∨
(proj_tuple ns (zip nsx xs) ≠ proj_tuple ns (zip nsy ys) ∧
(∀ x ∈ set (proj_tuple ns (zip nsx xs)). isl x) ∧ (∀ y ∈ set (proj_tuple ns (zip nsy ys)). isl y))
shows *eval_conj_tuple Z nsx nsy xs ys = {}*
⟨*proof*⟩

lemma *eval_conj_tuple_empty*:
assumes *fo_nmlzd Z xs fo_nmlzd Z ys*
length nsx = length xs length nsy = length ys
sorted_distinct nsx sorted_distinct nsy
ns = filter (λn. n ∈ set nsy) nsx
fo_nmlz Z (proj_tuple ns (zip nsx xs)) ≠ fo_nmlz Z (proj_tuple ns (zip nsy ys))
shows *eval_conj_tuple Z nsx nsy xs ys = {}*
⟨*proof*⟩

lemma *nall_tuples_rec_filter*:
assumes *xs ∈ nall_tuples_rec AD n (length xs) ys = filter (λx. ¬isl x) xs*
shows *ys ∈ nall_tuples_rec {} n (length ys)*
⟨*proof*⟩

lemma *nall_tuples_rec_filter_rev*:
assumes *ys ∈ nall_tuples_rec {} n (length ys) ys = filter (λx. ¬isl x) xs*
Inl −‘ set xs ⊆ AD
shows *xs ∈ nall_tuples_rec AD n (length xs)*
⟨*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 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*: $xs \in X\varphi$ $ys \in X\psi$
and σxs_def: $xs = map\ \sigma xs\ ns\varphi\ map\ snd\ cys = map\ \sigma xs\ (map\ fst\ cys)$
$ys\psi = map\ \sigma xs\ nys$
and σys_def: $ys = map\ \sigma ys\ ns\psi\ map\ snd\ cxs = map\ \sigma ys\ (map\ fst\ cxs)$
$xs\varphi = map\ \sigma ys\ nxs$
and $fs\varphi\_def$: $fs\varphi = map\ \sigma xs\ ns\varphi'$
and $fs\psi\_def$: $fs\psi = map\ \sigma ys\ ns\psi'$
and $ys\psi\_def$: $map\ \sigma xs\ nys \in nall\_tuples\_rec\ \{\}$
$(card\ (Inr\ -`\ set\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys)))))\ (length\ nys)$
and *Inl_set_AD*: $Inl\ -`\ (set\ (map\ snd\ cxs) \cup set\ xs\varphi) \subseteq AD$
$Inl\ -`\ (set\ (map\ snd\ cys) \cup set\ ys\psi) \subseteq AD$
and *ad_agr*: $ad\_agr\_list\ AD\ (map\ \sigma ys\ (sort\ (ns\psi\ @\ ns\psi')))\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ ns\varphi')))$
shows
$map\ snd\ (merge\ (zip\ ns\varphi\ xs)\ (zip\ ns\varphi'\ fs\varphi)) =$
$map\ snd\ (merge\ (zip\ (sort\ (ns\varphi\ @\ map\ fst\ cys))\ (map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))))$
$(zip\ nys\ (map\ \sigma xs\ nys)))$ **and**
$map\ snd\ (merge\ (zip\ ns\varphi\ xs)\ cys) = map\ \sigma xs\ (sort\ (ns\varphi\ @\ map\ fst\ cys))$
$fs\varphi \in nall\_tuples\_rec\ AD\ (card\ (Inr\ -`\ set\ xs))\ (length\ ns\varphi')$
⟨*proof*⟩

lemma *eval_conj_set_correct*:
assumes $ns\varphi'\_def$: $ns\varphi' = filter\ (\lambda n.\ n \notin set\ ns\varphi)\ ns\psi$
and $ns\psi'\_def$: $ns\psi' = filter\ (\lambda n.\ n \notin set\ ns\psi)\ ns\varphi$
and $X\varphi\_def$: $X\varphi = fo\_nmlz\ AD\ `\ proj\_vals\ R\varphi\ ns\varphi$
and $X\psi\_def$: $X\psi = fo\_nmlz\ AD\ `\ proj\_vals\ R\psi\ ns\psi$
and *distinct*: $sorted\_distinct\ ns\varphi\ sorted\_distinct\ ns\psi$
shows $eval\_conj\_set\ AD\ ns\varphi\ X\varphi\ ns\psi\ X\psi = ext\_tuple\_set\ AD\ ns\varphi\ ns\varphi'\ X\varphi \cap ext\_tuple\_set\ AD\ ns\psi\ ns\psi'\ X\psi$
⟨*proof*⟩

lemma *esat_exists_not_fv*: $n \notin fv\_fo\_fmla\ \varphi \Longrightarrow X \neq \{\} \Longrightarrow$
$esat\ (Exists\ n\ \varphi)\ I\ \sigma\ X \longleftrightarrow esat\ \varphi\ I\ \sigma\ X$
⟨*proof*⟩

lemma *esat_forall_not_fv*: $n \notin fv\_fo\_fmla\ \varphi \Longrightarrow X \neq \{\} \Longrightarrow$
$esat\ (Forall\ n\ \varphi)\ I\ \sigma\ X \longleftrightarrow esat\ \varphi\ I\ \sigma\ X$
⟨*proof*⟩

lemma *proj_sat_vals*: $proj\_sat\ \varphi\ I =$
$proj\_vals\ \{\sigma.\ sat\ \varphi\ I\ \sigma\}\ (fv\_fo\_fmla\_list\ \varphi)$
⟨*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'*: $\bigcup(set\ (map\ set\_fo\_term\ ts)) \subseteq X \Longrightarrow$
$ad\_agr\_list\ X\ (map\ \sigma\ (fv\_fo\_terms\_list\ ts))\ (map\ \tau\ (fv\_fo\_terms\_list\ ts)) \Longrightarrow$
$ad\_agr\_list\ X\ (\sigma \odot e\ ts)\ (\tau \odot e\ ts)$
⟨*proof*⟩

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

fo_nmlz AD ‘ proj_vals Sφ (fv_fo_fmla_list ψ)
ad_agr_close_set (AD − ADφ) Xφ = fo_nmlz AD ‘ proj_vals Sφ (fv_fo_fmla_list φ)
⟨*proof*⟩

lemma *proj_ext_tuple*:
assumes *Sφ_def*: $S\varphi \equiv \{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\}$
and *AD_sub*: *act_edom φ I ⊆ AD*
and *Xφ_def*: $X\varphi = fo\_nmlz\ AD$ ‘ *proj_vals Sφ (fv_fo_fmla_list φ)*
and *nsφ'_def*: $ns\varphi' = filter\ (\lambda n.\ n \notin fv\_fo\_fmla\ \varphi)\ ns\psi$
and *sd_nsψ*: *sorted_distinct nsψ*
and *fv_Un*: $fv\_fo\_fmla\ \psi = fv\_fo\_fmla\ \varphi \cup set\ ns\psi$
and *Z_props*: $\bigwedge xs.\ xs \in Z \Longrightarrow fo\_nmlz\ AD\ xs = xs \wedge length\ xs = length\ (fv\_fo\_fmla\_list\ \psi)$
shows $Z \cap ext\_tuple\_set\ AD\ (fv\_fo\_fmla\_list\ \varphi)\ ns\varphi'\ X\varphi =$
$\{xs \in Z.\ fo\_nmlz\ AD\ (proj\_tuple\ (fv\_fo\_fmla\_list\ \varphi)\ (zip\ (fv\_fo\_fmla\_list\ \psi)\ xs)) \in X\varphi\}$
$Z - ext\_tuple\_set\ AD\ (fv\_fo\_fmla\_list\ \varphi)\ ns\varphi'\ X\varphi =$
$\{xs \in Z.\ fo\_nmlz\ AD\ (proj\_tuple\ (fv\_fo\_fmla\_list\ \varphi)\ (zip\ (fv\_fo\_fmla\_list\ \psi)\ xs)) \notin X\varphi\}$
⟨*proof*⟩

lemma *fo_nmlz_proj_sub*: $fo\_nmlz\ AD$ ‘ $proj\_fmla\ \varphi\ R \subseteq nall\_tuples\ AD\ (nfv\ \varphi)$
⟨*proof*⟩

lemma *fin_ad_agr_list_iff*:
fixes *AD* :: (′*a* :: *infinite*) *set*
assumes *finite AD* $\bigwedge vs.\ vs \in Z \Longrightarrow length\ vs = n$
$Z = \{ts.\ \exists ts' \in X.\ ad\_agr\_list\ AD\ (map\ Inl\ ts)\ ts'\}$
shows *finite Z* $\longleftrightarrow \bigcup (set$ ‘ $Z) \subseteq AD$
⟨*proof*⟩

lemma *proj_out_list*:
fixes *AD* :: (′*a* :: *infinite*) *set*
and σ :: *nat* ⇒ ′*a* + *nat*
and *ns* :: *nat list*
assumes *finite AD*
shows $\exists \tau.\ ad\_agr\_list\ AD\ (map\ \sigma\ ns)\ (map\ (Inl \circ \tau)\ ns) \wedge$
$(\forall j\ x.\ j \in set\ ns \longrightarrow \sigma\ j = Inl\ x \longrightarrow \tau\ j = x)$
⟨*proof*⟩

lemma *proj_out*:
fixes φ :: (′*a* :: *infinite*, ′*b*) *fo_fmla*
and *J* :: ((′*a*, *nat*) *fo_t*, ′*b*) *fo_intp*
assumes *wf_fo_intp φ I esat φ I σ UNIV*
shows $\exists \tau.\ esat\ \varphi\ I\ (Inl \circ \tau)\ UNIV \wedge (\forall i\ x.\ i \in fv\_fo\_fmla\ \varphi \wedge \sigma\ i = Inl\ x \longrightarrow \tau\ i = x) \wedge$
ad_agr_list (act_edom φ I) (map σ (fv_fo_fmla_list φ)) (map (Inl ∘ τ) (fv_fo_fmla_list φ))
⟨*proof*⟩

lemma *proj_fmla_esat_sat*:
fixes φ :: (′*a* :: *infinite*, ′*b*) *fo_fmla*
and *J* :: ((′*a*, *nat*) *fo_t*, ′*b*) *fo_intp*
assumes *wf*: *wf_fo_intp φ I*
shows *proj_fmla φ* $\{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\} \cap map\ Inl$ ‘ $UNIV =$
map Inl ‘ *proj_fmla φ* $\{\sigma.\ sat\ \varphi\ I\ \sigma\}$
⟨*proof*⟩

lemma *norm_proj_fmla_esat_sat*:
fixes φ :: (′*a* :: *infinite*, ′*b*) *fo_fmla*
assumes *wf_fo_intp φ I*
shows *fo_nmlz (act_edom φ I)* ‘ *proj_fmla φ* $\{\sigma.\ esat\ \varphi\ I\ \sigma\ UNIV\} =$
fo_nmlz (act_edom φ I) ‘ *map Inl* ‘ *proj_fmla φ* $\{\sigma.\ sat\ \varphi\ I\ \sigma\}$

⟨*proof*⟩

lemma *proj_sat_fmla*: *proj_sat* φ *I* = *proj_fmla* φ {σ. *sat* φ *I* σ}
⟨*proof*⟩

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

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

lemma *fo_rep_fin*:
assumes *fo_wf* φ *I* (*AD*, *n*, *X*) *fo_fin* (*AD*, *n*, *X*)
shows *fo_rep* (*AD*, *n*, *X*) = *map projl* ‘ *X*
⟨*proof*⟩

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

lemma *map_projl_Inl*: *map projl* (*map Inl xs*) = *xs*
⟨*proof*⟩

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* × *nat* ⇒ '*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*: ⋃(*set* (*map set_fo_term ts*)) ⊆ *AD* ⟹ *ad_agr_list AD* (σ ⊙*e ts*) *zs* ⟹
∃ τ. *zs* = τ ⊙*e ts*
⟨*proof*⟩

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

lemma *ad_agr_list_fv_list*: *ad_agr_list X* (σ ⊙*e ts*) (τ ⊙*e ts*) ⟹

ad_agr_list X (map σ (fv_fo_terms_list ts)) (map τ (fv_fo_terms_list ts))
⟨*proof*⟩

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

lemma *eval_eq*: **fixes** *I* :: *'b × nat ⇒ '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 ⊙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 :: infinite) set*
assumes *fo_wf φ I (AD, n, X)*
shows *X = fo_nmlz AD ‘ map Inl ‘ fo_rep (AD, n, X)*
⟨*proof*⟩

lemma *fo_wf_X*:
fixes *φ* :: *('a :: infinite, 'b) fo_fmla*
assumes *wf*: *fo_wf φ I (AD, n, X)*
shows *X = fo_nmlz AD ‘ proj_fmla φ {σ. esat φ I σ UNIV}*
⟨*proof*⟩

lemma *eval_neg*:
fixes *φ* :: *('a :: infinite, 'b) fo_fmla*
assumes *wf*: *fo_wf φ I t*
shows *fo_wf (Neg φ) I (eval_neg (fv_fo_fmla_list φ) t)*
⟨*proof*⟩

definition *cross_with f t t' = ⋃((λxs. ⋃(f xs ‘ t')) ‘ t)*

lemma *mapping_join_cross_with*:
assumes *⋀x x'. x ∈ t ⟹ x' ∈ t' ⟹ h x ≠ h' x' ⟹ f x x' = {}*
shows *set_of_idx (mapping_join (cross_with f) (cluster (Some ∘ h) t) (cluster (Some ∘ h') t')) = cross_with f t t'*
⟨*proof*⟩

lemma *fo_nmlzd_mono_sub*: *X ⊆ X' ⟹ fo_nmlzd X xs ⟹ fo_nmlzd X' xs*
⟨*proof*⟩

lemma *idx_join*:
assumes *Xφ_props*: *⋀vs. vs ∈ Xφ ⟹ fo_nmlzd AD vs ∧ length vs = length nsφ*
assumes *Xψ_props*: *⋀vs. vs ∈ Xψ ⟹ fo_nmlzd AD vs ∧ length vs = length nsψ*
assumes *sd_ns*: *sorted_distinct nsφ sorted_distinct nsψ*
assumes *ns_def*: *ns = filter (λn. n ∈ set nsψ) nsφ*
shows *idx_join AD ns nsφ Xφ nsψ Xψ = eval_conj_set AD nsφ Xφ nsψ Xψ*
⟨*proof*⟩

lemma *proj_fmla_conj_sub*:
assumes *AD_sub*: *act_edom* ψ *I* $\subseteq$ *AD*
shows *fo_nmlz AD* ‘ *proj_fmla* (*Conj* φ ψ) {σ. *esat* φ *I* σ *UNIV*} $\cap$
fo_nmlz AD ‘ *proj_fmla* (*Conj* φ ψ) {σ. *esat* ψ *I* σ *UNIV*} $\subseteq$
fo_nmlz AD ‘ *proj_fmla* (*Conj* φ ψ) {σ. *esat* (*Conj* φ ψ) *I* σ *UNIV*}
⟨*proof*⟩

lemma *eval_conj*:
fixes φ :: (′*a* :: *infinite*, ′*b*) *fo_fmla*
assumes *wf*: *fo_wf* φ *I* *t*φ *fo_wf* ψ *I* *t*ψ
shows *fo_wf* (*Conj* φ ψ) *I* (*eval_conj* (*fv_fo_fmla_list* φ) *t*φ (*fv_fo_fmla_list* ψ) *t*ψ)
⟨*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*′})) $\Longrightarrow$
set_of_idx (*Mapping.map_values f* (*cluster* (*Some* $\circ$ *h*) *X*)) = $\bigcup$((λ*x*. *f* (*h x*) {*x*}) ‘ *X*)
⟨*proof*⟩

lemma *fo_nmlz_twice*:
assumes *sorted_distinct ns sorted_distinct ns*′ *set ns* $\subseteq$ *set ns*′
shows *fo_nmlz AD* (*proj_tuple ns* (*zip ns*′ (*fo_nmlz AD* (*map* σ *ns*′)))) = *fo_nmlz AD* (*map* σ *ns*)
⟨*proof*⟩

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 φ :: (′*a* :: *infinite*, ′*b*) *fo_fmla*
assumes *wf*: *fo_wf* φ *I* *t*φ *fo_wf* ψ *I* *t*ψ
shows *fo_wf* (*Conj* φ (*Neg* ψ)) *I*
(*eval_ajoin* (*fv_fo_fmla_list* φ) *t*φ (*fv_fo_fmla_list* ψ) *t*ψ)
⟨*proof*⟩

lemma *eval_disj*:
fixes φ :: (′*a* :: *infinite*, ′*b*) *fo_fmla*
assumes *wf*: *fo_wf* φ *I* *t*φ *fo_wf* ψ *I* *t*ψ
shows *fo_wf* (*Disj* φ ψ) *I*
(*eval_disj* (*fv_fo_fmla_list* φ) *t*φ (*fv_fo_fmla_list* ψ) *t*ψ)
⟨*proof*⟩

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

fv_fo_fmla_list (Forall i φ) = fv_fo_fmla_list φ
⟨*proof*⟩

lemma *nfv_ex_all*:
assumes *Some*: *pos i (fv_fo_fmla_list φ) = Some j*
shows *nfv φ = Suc (nfv (Exists i φ)) nfv φ = Suc (nfv (Forall i φ))*
⟨*proof*⟩

lemma *fv_fo_fmla_list_exists*: *fv_fo_fmla_list (Exists n φ) = filter ((≠) n) (fv_fo_fmla_list φ)*
⟨*proof*⟩

lemma *eval_exists*:
fixes *φ* :: (*'a* :: *infinite*, *'b*) *fo_fmla*
assumes *wf*: *fo_wf φ I t*
shows *fo_wf (Exists i φ) I (eval_exists i (fv_fo_fmla_list φ) t)*
⟨*proof*⟩

lemma *fv_fo_fmla_list_forall*: *fv_fo_fmla_list (Forall n φ) = filter ((≠) n) (fv_fo_fmla_list φ)*
⟨*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*:
fo_nmlz AD xs = xs ⟹ set xs = set xs ∩ Inl ‘ AD ∪ Inr ‘ {..<card (Inr −‘ set xs)}
⟨*proof*⟩

lemma *ad_agr_list_take_drop*: *ad_agr_list AD xs ys ⟹*
ad_agr_list AD (take i xs @ drop (Suc i) xs) (take i ys @ drop (Suc i) ys)
⟨*proof*⟩

lemma *fo_nmlz_rem_nth_add_nth*:
assumes *fo_nmlz AD zs = zs i ≤ length zs*
shows *fo_nmlz AD (rem_nth i (fo_nmlz AD (add_nth i z zs))) = zs*
⟨*proof*⟩

lemma *ad_agr_list_add*:
assumes *ad_agr_list AD xs ys i ≤ length xs*
shows *∃ z′ ∈ Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set ys))} ∪ set ys.*
ad_agr_list AD (take i xs @ z # drop i xs) (take i ys @ z′ # drop i ys)
⟨*proof*⟩

lemma *add_nth_restrict*:
assumes *fo_nmlz AD zs = zs i ≤ length zs*
shows *∃ z′ ∈ Inl ‘ AD ∪ Inr ‘ {..<Suc (card (Inr −‘ set zs))}.*
fo_nmlz AD (add_nth i z zs) = fo_nmlz AD (add_nth i z′ zs)
⟨*proof*⟩

lemma *fo_nmlz_add_rem*:
assumes *i ≤ length zs*
shows *∃ z′. fo_nmlz AD (add_nth i z zs) = fo_nmlz AD (add_nth i z′ (fo_nmlz AD zs))*
⟨*proof*⟩

lemma *fo_nmlz_add_rem′*:
assumes *i ≤ length zs*
shows *∃ z′. fo_nmlz AD (add_nth i z (fo_nmlz AD zs)) = fo_nmlz AD (add_nth i z′ zs)*

⟨*proof*⟩

lemma *fo_nmlz_add_nth_rem_nth*:
assumes *fo_nmlz AD xs* = *xs i* < *length xs*
shows ∃ *z*. *fo_nmlz AD* (*add_nth i z* (*fo_nmlz AD* (*rem_nth i xs*))) = *xs*
⟨*proof*⟩

lemma *sp_equiv_list_almost_same*: *sp_equiv_list* (*xs* @ *v* # *ys*) (*xs* @ *w* # *ys*) ⟹
v ∈ *set xs* ∪ *set ys* ∨ *w* ∈ *set xs* ∪ *set ys* ⟹ *v* = *w*
⟨*proof*⟩

lemma *ad_agr_list_add_nth*:
assumes *i* ≤ *length zs ad_agr_list AD* (*add_nth i v zs*) (*add_nth i w zs*) *v* ≠ *w*
shows {*v*, *w*} ∩ (*Inl* ‘ *AD* ∪ *set zs*) = {}
⟨*proof*⟩

lemma *Inr_in_tuple*:
assumes *fo_nmlz AD zs* = *zs n* < *card* (*Inr* −‘ *set zs*)
shows *Inr n* ∈ *set zs*
⟨*proof*⟩

lemma *card_wit_sub*:
assumes *finite Z card Z* ≤ *card* {*ts* ∈ *X*. ∃ *z* ∈ *Z*. *ts* = *f z*}
shows *f* ‘ *Z* ⊆ *X*
⟨*proof*⟩

lemma *add_nth_iff_card*:
assumes (⋀*xs*. *xs* ∈ *X* ⟹ *fo_nmlz AD xs* = *xs*) (⋀*xs*. *xs* ∈ *X* ⟹ *i* < *length xs*)
fo_nmlz AD zs = *zs i* ≤ *length zs finite AD finite X*
shows (∀ *z*. *fo_nmlz AD* (*add_nth i z zs*) ∈ *X*) ⟷
Suc (*card AD* + *card* (*Inr* −‘ *set zs*)) ≤ *card* {*ts* ∈ *X*. ∃ *z*. *ts* = *fo_nmlz AD* (*add_nth i z zs*)}
⟨*proof*⟩

lemma *set_fo_nmlz_add_nth_rem_nth*:
assumes *j* < *length xs* ⋀*x*. *x* ∈ *X* ⟹ *fo_nmlz AD x* = *x*
⋀*x*. *x* ∈ *X* ⟹ *j* < *length x*
shows {*ts* ∈ *X*. ∃ *z*. *ts* = *fo_nmlz AD* (*add_nth j z* (*fo_nmlz AD* (*rem_nth j xs*)))} =
{*y* ∈ *X*. *fo_nmlz AD* (*rem_nth j y*) = *fo_nmlz AD* (*rem_nth j xs*)}
⟨*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 σ* ⟷ (*if wf_fo_intp φ I then*
(*case eval_fmla φ I of* (*AD*, *n*, *X*) ⇒
fo_nmlz (*act_edom φ I*) (*map* (*Inl* ∘ *σ*) (*fv_fo_fmla_list φ*)) ∈ *X*)
else sat φ I σ)
⟨*proof*⟩

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

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

fun *convert_db_rec* :: (*'a* × *'c list*) *list* ⇒ ((*'a* × *nat*), *'c list set*) *mapping* ⇒
((*'a* × *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 ⟹*
∃ *tss'*. *Mapping.lookup* (*convert_db_rec ktss m*) (*r*, *n*) = *Some tss'* ∧ *tss* ⊆ *tss'*
⟨*proof*⟩

lemma *convert_db_rec_sound*: (*r*, *ts*) ∈ *set ktss ⟹*
∃ *tss*. *Mapping.lookup* (*convert_db_rec ktss m*) (*r*, *length ts*) = *Some tss* ∧ *ts* ∈ *tss*
⟨*proof*⟩

lemma *convert_db_rec_complete*: *Mapping.lookup* (*convert_db_rec ktss m*) (*r*, *n*) = *Some tss' ⟹*
ts ∈ *tss' ⟹*

(*length ts* = *n* ∧ (*r*, *ts*) ∈ *set ktss*) ∨ (∃ *tss*. *Mapping.lookup m* (*r*, *n*) = *Some tss* ∧ *ts* ∈ *tss*)
⟨*proof*⟩

definition *convert_db* :: ('*a* × '*c list*) *list* ⇒ ('*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* ⇒ {} | *Some v* ⇒ *v*))

lemma *convert_db_correct*: (*ts* ∈ *convert_db ktss* (*r*, *n*) ⟶ *n* = *length ts*) ∧
((*r*, *ts*) ∈ *set ktss* ⟷ *ts* ∈ *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* ∈ *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.