Making Arbitrary Relational Calculus Queries Safe-Range

Martin Raszyk Dmitriy Traytel

March 17, 2025

Abstract

The relational calculus (RC), i.e., first-order logic with equality but without function symbols, is a concise, declarative database query language. In contrast to relational algebra or SQL, which are the traditional query languages of choice in the database community, RC queries can evaluate to an infinite relation. Moreover, even in cases where the evaluation result of an RC query would be finite it is not clear how to efficiently compute it. Safe-range RC is an interesting syntactic subclass of RC, because all safe-range queries evaluate to a finite result and it is well-known [1, §5.4] how to evaluate such queries by translating them to relational algebra. We formalize and prove correct our recent translation [2] of an arbitrary RC query into a pair of safe-range queries. Assuming an infinite domain, the two queries have the following meaning: The first is closed and characterizes the original query's relative safety, i.e., whether given a fixed database (interpretation of atomic predicates with finite relations), the original query evaluates to a finite relation. The second safe-range query is equivalent to the original query, if the latter is relatively safe.

The formalization uses the Refinement Framework to go from the non-deterministic algorithm described in the paper to a deterministic, executable query translation. Our executable query translation is a first step towards a verified tool that efficiently evaluates arbitrary RC queries. This very problem is also solved by the AFP entry Eval_FO with a theoretically incomparable but practically worse time complexity. (The latter is demonstrated by our empirical evaluation [2].)

Contents

1 Preliminaries

1.1 Iterated Function Update

abbreviation *fun_upds* (‹_[_ :=* _]› [*90*, *0*, *0*] *91*) **where**
f[*xs* :=* *ys*] ≡ *fold* (λ(*x*, *y*) *f*. *f*(*x* := *y*)) (*zip xs ys*) *f*

fun *restrict* **where**
restrict A (*x* # *xs*) (*y* # *ys*) = (*if x* ∈ *A then y* # *restrict* (*A* − {*x*}) *xs ys else restrict A xs ys*)
| *restrict A* _ _ = []

fun *extend* :: *nat set* ⇒ *nat list* ⇒ *'a list* ⇒ *'a list set* **where**
extend A (*x* # *xs*) *ys* = (*if x* ∈ *A*
then (⋃ *zs* ∈ *extend* (*A* − {*x*}) *xs* (*tl ys*). {*hd ys* # *zs*})
else (⋃ *z* ∈ *UNIV*. ⋃ *zs* ∈ *extend A xs ys*. {*z* # *zs*}))
| *extend A* _ _ = {[]}

fun *lookup* **where**
lookup (*x* # *xs*) (*y* # *ys*) *z* = (*if x* = *z then y else lookup xs ys z*)
| *lookup* _ _ _ = *undefined*

lemma *extend_nonempty*: *extend A xs ys* ≠ {}
by (*induct xs arbitrary*: *A ys*) *auto*

lemma *length_extend*: *zs* ∈ *extend A xs ys* ⟹ *length zs* = *length xs*
by (*induct xs arbitrary*: *A ys zs*) (*auto split*: *if_splits*)

lemma *ex_lookup_extend*: *x* ∉ *A* ⟹ *x* ∈ *set xs* ⟹ ∃ *zs* ∈ *extend A xs ys*. *lookup xs zs x* = *d*
proof (*induct xs arbitrary*: *A ys*)
case (*Cons a xs*)
from *Cons*(*1*)[*of A* − {*a*} *tl ys*] *Cons*(*1*)[*of A ys*] *Cons*(*2*−) **show** *?case*
by (*auto simp*: *ex_in_conv extend_nonempty*)
qed *simp*

lemma *restrict_extend*: *A* ⊆ *set xs* ⟹ *length ys* = *card A* ⟹ *zs* ∈ *extend A xs ys* ⟹ *restrict A xs zs* = *ys*
proof (*induct xs arbitrary*: *A ys zs*)
case (*Cons a xs*)
then have *finite A*
by (*elim finite_subset*) *auto*
with *Cons*(*1*)[*of A* − {*a*} *tl ys tl zs*] *Cons*(*1*)[*of A ys tl zs*] *Cons*(*2*−) **show** *?case*
by (*cases ys*) (*auto simp*: *subset_insert_iff split*: *if_splits*)
qed *simp*

lemma *fun_upds_notin*[*simp*]: *length xs* = *length ys* ⟹ *x* ∉ *set xs* ⟹ (*σ*[*xs* :=* *ys*]) *x* = *σ x*
by (*induct xs ys arbitrary*: *σ rule*: *list_induct2*) *auto*

lemma *fun_upds_twist*: *length xs* = *length ys* ⟹ *a* ∉ *set xs* ⟹ *σ*(*a* := *x*)[*xs* :=* *ys*] = (*σ*[*xs* :=* *ys*])(*a* := *x*)
by (*induct xs ys arbitrary*: *σ rule*: *list_induct2*) (*auto simp*: *fun_upd_twist*)

lemma *fun_upds_twist_apply*: *length xs* = *length ys* ⟹ *a* ∉ *set xs* ⟹ *a* ≠ *b* ⟹ (*σ*(*a* := *x*)[*xs* :=* *ys*]) *b* = (*σ*[*xs* :=* *ys*]) *b*
by (*induct xs ys arbitrary*: *σ rule*: *list_induct2*) (*auto simp*: *fun_upd_twist*)

lemma *fun_upds_extend*:
x ∈ *A* ⟹ *A* ⊆ *set xs* ⟹ *distinct xs* ⟹ *sorted xs* ⟹ *length ys* = *card A* ⟹ *zs* ∈ *extend A xs ys* ⟹
(*σ*[*xs* :=* *zs*]) *x* = (*σ*[*sorted_list_of_set A* :=* *ys*]) *x*
proof (*induct xs arbitrary*: *A ys zs σ*)
case (*Cons a xs*)
then have *fin*[*simp*]: *finite A*
by (*elim finite_subset*) *auto*
from *Cons*(*2*−) **have** *a* ∈ *A* ⟹ *Min A* = *a* **if** *a* ∈ *A*
by (*intro Min_eqI*) *auto*
with *Cons*(*2*) *fin* **have** ∗: *a* ∈ *A* ⟹ *sorted_list_of_set A* = *a* # *sorted_list_of_set* (*A* − {*a*})
by (*subst sorted_list_of_set_nonempty*) *auto*
show *?case*
using *Cons*(*1*)[*of A* − {*a*} *tl ys*] *Cons*(*1*)[*of A ys*] *Cons*(*2*−)
by (*cases ys*; *cases x* = *a*)
(*auto simp add*: *subset_insert_iff* ∗ *fun_upds_twist_apply length_extend simp del*: *fun_upd_apply split*: *if_splits*)
qed *simp*

lemma *fun_upds_map_self*: *σ*[*xs* :=* *map σ xs*] = *σ*
by (*induct xs arbitrary*: *σ*) *auto*

lemma *fun_upds_single*: *distinct xs* ⟹ *σ*[*xs* :=* *map* (*σ*(*y* := *d*)) *xs*] = (*if y* ∈ *set xs then σ*(*y* := *d*) *else σ*)
by (*induct xs arbitrary*: *σ*) (*auto simp*: *fun_upds_twist*)

1.2 Lists and Sets

lemma *find_index_less_size*: ∃ *x* ∈ *set xs*. *P x* ⟹ *find_index P xs* < *size xs*
by (*induct xs*) *auto*

lemma *index_less_size*: *x* ∈ *set xs* ⟹ *index xs x* < *size xs*
by (*simp add*: *index_def find_index_less_size*)

lemma *fun_upds_in*: *length xs* = *length ys* ⟹ *distinct xs* ⟹ *x* ∈ *set xs* ⟹ (*σ*[*xs* :=* *ys*]) *x* = *ys* ! *index xs x*
by (*induct xs ys arbitrary*: *σ rule*: *list_induct2*) *auto*

lemma *remove_nth_index*: *remove_nth* (*index ys y*) *ys* = *remove1 y ys*
by (*induct ys*) *auto*

lemma *index_remove_nth*: *distinct xs* ⟹ *x* ∈ *set xs* ⟹ *index* (*remove_nth i xs*) *x* = (*if index xs x* < *i then index xs x else if i* = *index xs x then length xs* − *1 else index xs x* − *1*)
by (*induct i xs rule*: *remove_nth.induct*) (*auto simp*: *not_less intro*!: *Suc_pred split*: *if_splits*)

lemma *insert_nth_nth_index*:
y ≠ *z* ⟹ *y* ∈ *set ys* ⟹ *z* ∈ *set ys* ⟹ *length ys* = *Suc* (*length xs*) ⟹ *distinct ys* ⟹
insert_nth (*index ys y*) *x xs* ! *index ys z* =

xs ! *index* (*remove1 y ys*) *z*
by (*subst nth_insert_nth*;
auto simp: *remove_nth_index*[*symmetric*] *index_remove_nth dest*: *index_less_size intro*!: *arg_cong*[*of* _ _ *nth xs*] *index_eqI*)

lemma *index_lt_index_remove*: *index xs x* < *index xs y* ⟹ *index xs x* = *index* (*remove1 y xs*) *x*
by (*induct xs*) *auto*

lemma *index_gt_index_remove*: *index xs x* > *index xs y* ⟹ *index xs x* = *Suc* (*index* (*remove1 y xs*) *x*)
proof (*induct xs*)
case (*Cons z xs*)
then show *?case*
by (*cases z* = *x*) *auto*
qed *simp*

lemma *lookup_map*[*simp*]: *x* ∈ *set xs* ⟹ *lookup xs* (*map f xs*) *x* = *f x*
by (*induct xs*) *auto*

lemma *in_set_remove_cases*: *P z* ⟹ (∀ *x* ∈ *set* (*remove1 z xs*). *P x*) ⟹ *x* ∈ *set xs* ⟹ *P x*
by (*cases x* = *z*) *auto*

lemma *insert_remove_id*: *x* ∈ *X* ⟹ *X* = *insert x* (*X* − {*x*})
by *auto*

lemma *infinite_surj*: *infinite A* ⟹ *A* ⊆ *f* ‘ *B* ⟹ *infinite B*
by (*elim contrapos_nn finite_surj*)

class *infinite* =
fixes *to_nat* :: *'a* ⇒ *nat*
assumes *surj_to_nat*: *surj to_nat*
begin

lemma *infinite_UNIV*: *infinite* (*UNIV* :: *'a set*)
using *surj_to_nat* **by** (*intro infinite_surj*[*of UNIV to_nat*]) *auto*

end

instantiation *nat* :: *infinite* **begin**
definition *to_nat_nat* :: *nat* ⇒ *nat* **where** *to_nat_nat* = *id*
instance by *standard* (*auto simp*: *to_nat_nat_def*)
end

instantiation *list* :: (*type*) *infinite* **begin**
definition *to_nat_list* :: *'a list* ⇒ *nat* **where** *to_nat_list* = *length*
instance by *standard* (*auto simp*: *image_iff to_nat_list_def intro*!: *exI*[*of* _ *replicate* _ _])
end

1.3 Equivalence Closure and Classes

definition *symcl* **where**
symcl r = {(*x*, *y*). (*x*, *y*) ∈ *r* ∨ (*y*, *x*) ∈ *r*}

definition *transymcl* **where**
transymcl r = *trancl* (*symcl r*)

lemma *symclp_symcl_eq*[*pred_set_conv*]: *symclp* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x y*. (*x*, *y*) ∈ *symcl r*)
by (*auto simp*: *symclp_def symcl_def fun_eq_iff*)

definition *classes Qeq* = *quotient* (*Field Qeq*) (*transymcl Qeq*)

lemma *Field_symcl*[*simp*]: *Field* (*symcl r*) = *Field r*
unfolding *symcl_def Field_def* **by** *auto*

lemma *Domain_symcl*[*simp*]: *Domain* (*symcl r*) = *Field r*
unfolding *symcl_def Field_def* **by** *auto*

lemma *Field_trancl*[*simp*]: *Field* (*trancl r*) = *Field r*
unfolding *Field_def* **by** *auto*

lemma *Field_transymcl*[*simp*]: *Field* (*transymcl r*) = *Field r*
unfolding *transymcl_def* **by** *auto*

lemma *eqclass_empty_iff*[*simp*]: r `` $\{x\} = \{\} \longleftrightarrow x \notin$ *Domain r*
by *auto*

lemma *sym_symcl*[*simp*]: *sym* (*symcl r*)
unfolding *symcl_def sym_def* **by** *auto*

lemma *in_symclI*:
$(a,b) \in r \Longrightarrow (a,b) \in$ *symcl r*
$(a,b) \in r \Longrightarrow (b,a) \in$ *symcl r*
by (*auto simp*: *symcl_def*)

lemma *sym_transymcl*: *sym* (*transymcl r*)
by (*simp add*: *sym_trancl transymcl_def*)

lemma *symcl_insert*:
symcl (*insert* (*x*, *y*) *Qeq*) = *insert* (*y*, *x*) (*insert* (*x*, *y*) (*symcl Qeq*))
by (*auto simp*: *symcl_def*)

lemma *equiv_transymcl*: *Equiv_Relations.equiv* (*Field Qeq*) (*transymcl Qeq*)
by (*auto simp*: *Equiv_Relations.equiv_def sym_trancl refl_on_def transymcl_def*
dest: *FieldI1 FieldI2 Field_def*[*THEN equalityD1*, *THEN set_mp*]
intro: *r_r_into_trancl*[*of x _ _ x* **for** *x*] *elim*!: *in_symclI*)

lemma *equiv_quotient_no_empty_class*: *Equiv_Relations.equiv A r* $\Longrightarrow \{\} \notin A$ // *r*
by (*auto simp*: *quotient_def refl_on_def sym_def Equiv_Relations.equiv_def*)

lemma *classes_cover*: $\bigcup$(*classes Qeq*) = *Field Qeq*
by (*simp add*: *Union_quotient classes_def equiv_transymcl*)

lemma *classes_disjoint*: *X* $\in$ *classes Qeq* $\Longrightarrow$ *Y* $\in$ *classes Qeq* $\Longrightarrow$ *X* = *Y* $\vee$ *X* $\cap$ *Y* = $\{\}$
using *quotient_disj*[*OF equiv_transymcl*]
by (*auto simp*: *classes_def*)

lemma *classes_nonempty*: $\{\} \notin$ *classes Qeq*
using *equiv_quotient_no_empty_class*[*OF equiv_transymcl*]
by (*auto simp*: *classes_def*)

definition *class x Qeq* = (*if* $\exists$ *X* $\in$ *classes Qeq*. *x* $\in$ *X then Some* (*THE X*. *X* $\in$ *classes Qeq* $\wedge$ *x* $\in$ *X*) *else None*)

lemma *class_Some_eq*: *class x Qeq* = *Some X* $\longleftrightarrow$ *X* $\in$ *classes Qeq* $\wedge$ *x* $\in$ *X*
unfolding *class_def*
by (*auto 0 3 dest*: *classes_disjoint del*: *conjI intro*!: *the_equality*[*of _ X*]
conjI[*of* ($\exists$ *X*$\in$*classes Qeq*. *x* $\in$ *X*)] *intro*: *theI*[**where** *P*=λ*X*. *X* $\in$ *classes Qeq* $\wedge$ *x* $\in$ *X*])

lemma *class_None_eq*: *class x Qeq = None* $\longleftrightarrow$ *x* $\notin$ *Field Qeq*
by (*simp add*: *class_def classes_cover*[*symmetric*] *split*: *if_splits*)

lemma *insert_Image_triv*: *x* $\notin$ *r* $\Longrightarrow$ *insert* (*x*, *y*) *Qeq* `` *r* = *Qeq* `` *r*
by *auto*

lemma *Un1_Image_triv*: *Domain B* $\cap$ *r* = {} $\Longrightarrow$ (*A* $\cup$ *B*) `` *r* = *A* `` *r*
by *auto*

lemma *Un2_Image_triv*: *Domain A* $\cap$ *r* = {} $\Longrightarrow$ (*A* $\cup$ *B*) `` *r* = *B* `` *r*
by *auto*

lemma *classes_empty*: *classes* {} = {}
unfolding *classes_def* **by** *auto*

lemma *ex_class*: *x* $\in$ *Field Qeq* $\Longrightarrow$ $\exists$ *X*. *class x Qeq* = *Some X* $\wedge$ *x* $\in$ *X*
by (*metis Union_iff class_Some_eq classes_cover*)

lemma *equivD*:
Equiv_Relations.equiv A r $\Longrightarrow$ *refl_on A r*
Equiv_Relations.equiv A r $\Longrightarrow$ *sym r*
Equiv_Relations.equiv A r $\Longrightarrow$ *trans r*
by (*blast elim*: *Equiv_Relations.equivE*)+

lemma *transymcl_into*:
(*x*, *y*) $\in$ *r* $\Longrightarrow$ (*x*, *y*) $\in$ *transymcl r*
(*x*, *y*) $\in$ *r* $\Longrightarrow$ (*y*, *x*) $\in$ *transymcl r*
unfolding *transymcl_def* **by** (*blast intro*: *in_symclI r_into_trancl'*)+

lemma *transymcl_self*:
(*x*, *y*) $\in$ *r* $\Longrightarrow$ (*x*, *x*) $\in$ *transymcl r*
(*x*, *y*) $\in$ *r* $\Longrightarrow$ (*y*, *y*) $\in$ *transymcl r*
unfolding *transymcl_def* **by** (*blast intro*: *in_symclI*(*1*) *in_symclI*(*2*) *r_r_into_trancl*)+

lemma *transymcl_trans*: (*x*, *y*) $\in$ *transymcl r* $\Longrightarrow$ (*y*, *z*) $\in$ *transymcl r* $\Longrightarrow$ (*x*, *z*) $\in$ *transymcl r*
using *equiv_transymcl*[*THEN equivD*(*3*), *THEN transD*] **.**

lemma *transymcl_sym*: (*x*, *y*) $\in$ *transymcl r* $\Longrightarrow$ (*y*, *x*) $\in$ *transymcl r*
using *equiv_transymcl*[*THEN equivD*(*2*), *THEN symD*] **.**

lemma *edge_same_class*: *X* $\in$ *classes Qeq* $\Longrightarrow$ (*a*, *b*) $\in$ *Qeq* $\Longrightarrow$ *a* $\in$ *X* $\longleftrightarrow$ *b* $\in$ *X*
unfolding *classes_def* **by** (*elim quotientE*) (*auto elim*!: *transymcl_trans transymcl_into*)

lemma *Field_transymcl_self*: *a* $\in$ *Field Qeq* $\Longrightarrow$ (*a*, *a*) $\in$ *transymcl Qeq*
by (*auto simp*: *Field_def transymcl_def*[*symmetric*] *transymcl_self*)

lemma *transymcl_insert*: *transymcl* (*insert* (*a*, *b*) *Qeq*) = *transymcl Qeq* $\cup$ {(*a*,*a*),(*b*,*b*)} $\cup$
((*transymcl Qeq* $\cup$ {(*a*, *a*), (*b*, *b*)}) *O* {(*a*, *b*), (*b*, *a*)} *O* (*transymcl Qeq* $\cup$ {(*a*, *a*), (*b*, *b*)}) $-$ *transymcl Qeq*)
by (*auto simp*: *relcomp_def relcompp_apply transymcl_def symcl_insert trancl_insert2 dest*: *trancl_trans*)

lemma *transymcl_insert_both_new*: *a* $\notin$ *Field Qeq* $\Longrightarrow$ *b* $\notin$ *Field Qeq* $\Longrightarrow$
transymcl (*insert* (*a*, *b*) *Qeq*) = *transymcl Qeq* $\cup$ {(*a*,*a*),(*b*,*b*),(*a*,*b*),(*b*,*a*)}
unfolding *transymcl_insert*
by (*auto dest*: *FieldI1 FieldI2*)

lemma *transymcl_insert_same_class*: (*x*, *y*) $\in$ *transymcl Qeq* $\Longrightarrow$ *transymcl* (*insert* (*x*, *y*) *Qeq*) =

```
transymcl Qeq
  by (auto 0 3 simp: transymcl_insert intro: transymcl_sym transymcl_trans)

lemma classes_insert: classes (insert (x, y) Qeq) =
  (case (class x Qeq, class y Qeq) of
    (Some X, Some Y) ⇒ if X = Y then classes Qeq else classes Qeq − {X, Y} ∪ {X ∪ Y}
  | (Some X, None) ⇒ classes Qeq − {X} ∪ {insert y X}
  | (None, Some Y) ⇒ classes Qeq − {Y} ∪ {insert x Y}
  | (None, None) ⇒ classes Qeq ∪ {{x,y}})
proof ((cases class x Qeq; cases class y Qeq), goal_cases NN NS SN SS)
  case NN
  then have classes (insert (x, y) Qeq) = classes Qeq ∪ {{x, y}}
    by (fastforce simp: class_None_eq classes_def transymcl_insert_both_new insert_Image_triv quotientI
        elim!: quotientE dest: FieldI1  intro: quotient_def[THEN Set.equalityD2, THEN set_mp] intro!: disjI1)
  with NN show ?case
    by auto
next
  case (NS Y)
  then have insert x Y = transymcl (insert (x, y) Qeq) `` {x}
    unfolding transymcl_insert using FieldI1[of x _ transymcl Qeq]
   relcompI[OF insertI1 relcompI[OF insertI1 insertI2[OF insertI2[OF transymcl_trans[OF transymcl_sym]]]],
      of _ y Qeq _ x x insert (y,y) (transymcl Qeq) {(y,x)} (x, x) (y, y)]
    by (auto simp: class_None_eq class_Some_eq classes_def
      dest: FieldI1 FieldI2 elim!: quotientE intro: transymcl_sym transymcl_trans)
  then have ∗: insert x Y ∈ classes (insert (x, y) Qeq)
    by (auto simp: class_None_eq class_Some_eq classes_def  intro!: quotientI)
  moreover from ∗ NS have Y ∉ classes (insert (x, y) Qeq)
    using classes_disjoint[of Y insert (x, y) Qeq insert x Y] classes_cover[of Qeq]
    by (auto simp: class_None_eq class_Some_eq)
  moreover {
    fix Z
    assume Z: Z ≠ Y Z ∈ classes Qeq
    then obtain z where z: z ∈ Field Qeq Z = transymcl Qeq `` {z}
      by (auto elim!: quotientE simp: classes_def)
    with NS Z have z ∈ Z z ≠ x z ≠ y (z, x) ∉ transymcl Qeq (z, y) ∉ transymcl Qeq
      using classes_disjoint[of Z Qeq Y] classes_nonempty[of Qeq]
      by (auto simp: class_None_eq class_Some_eq disjoint_iff Field_transymcl_self
        dest: FieldI2 intro: transymcl_trans)
    with NS Z ∗ have transymcl Qeq `` {z} = transymcl (insert (x, y) Qeq) `` {z}
      unfolding transymcl_insert
      by (intro trans[OF _ Un1_Image_triv[symmetric]]) (auto simp: class_None_eq class_Some_eq)
    with z have Z ∈ classes (insert (x, y) Qeq)
      by (auto simp: classes_def intro!: quotientI)
  }
  moreover {
    fix Z
    assume Z: Z ≠ insert x Y Z ∈ classes (insert (x, y) Qeq)
    then obtain z where z: z ∈ Field (insert (x, y) Qeq) Z = transymcl (insert (x, y) Qeq) `` {z}
      by (auto elim!: quotientE simp: classes_def)
   with NS Z ∗ have z ∈ Z z ≠ x z ≠ y (z, x) ∉ transymcl (insert (x, y) Qeq) (z, y) ∉ transymcl (insert (x, y) Qeq)
      using classes_disjoint[of Z insert (x, y) Qeq insert x Y] classes_nonempty[of insert (x, y) Qeq]
      by (auto simp: class_None_eq class_Some_eq Field_transymcl_self transymcl_into(2)
        intro: transymcl_trans)
    with NS Z ∗ have transymcl (insert (x, y) Qeq) `` {z} = transymcl Qeq `` {z}
      unfolding transymcl_insert
```

```
    by (intro trans[OF Un1_Image_triv]) (auto simp: class_None_eq class_Some_eq)
   with z ‹z ≠ x› ‹z ≠ y› have Z ∈ classes Qeq
    by (auto simp: classes_def intro!: quotientI)
  }
  ultimately have classes (insert (x, y) Qeq) = classes Qeq − {Y} ∪ {insert x Y}
   by blast
  with NS show ?case
   by auto
next
  case (SN X)
  then have insert y X = transymcl (insert (x, y) Qeq) `` {x}
   unfolding transymcl_insert using FieldI1[of x _ transymcl Qeq]
   by (auto simp: class_None_eq class_Some_eq classes_def
     dest: FieldI1 FieldI2 elim!: quotientE intro: transymcl_sym transymcl_trans)
  then have ∗: insert y X ∈ classes (insert (x, y) Qeq)
   by (auto simp: class_None_eq class_Some_eq classes_def  intro!: quotientI)
  moreover from ∗ SN have X ∉ classes (insert (x, y) Qeq)
   using classes_disjoint[of X insert (x, y) Qeq insert y X] classes_cover[of Qeq]
   by (auto simp: class_None_eq class_Some_eq)
  moreover {
   fix Z
   assume Z: Z ≠ X Z ∈ classes Qeq
   then obtain z where z: z ∈ Field Qeq Z = transymcl Qeq `` {z}
    by (auto elim!: quotientE simp: classes_def)
   with SN Z have z ∈ Z z ≠ x z ≠ y (z, x) ∉ transymcl Qeq (z, y) ∉ transymcl Qeq
    using classes_disjoint[of Z Qeq X] classes_nonempty[of Qeq]
    by (auto simp: class_None_eq class_Some_eq disjoint_iff Field_transymcl_self
      dest: FieldI2 intro: transymcl_trans)
   with SN Z ∗ have transymcl Qeq `` {z} = transymcl (insert (x, y) Qeq) `` {z}
    unfolding transymcl_insert
    by (intro trans[OF _ Un1_Image_triv[symmetric]]) (auto simp: class_None_eq class_Some_eq)
   with z have Z ∈ classes (insert (x, y) Qeq)
    by (auto simp: classes_def intro!: quotientI)
  }
  moreover {
   fix Z
   assume Z: Z ≠ insert y X Z ∈ classes (insert (x, y) Qeq)
   then obtain z where z: z ∈ Field (insert (x, y) Qeq) Z = transymcl (insert (x, y) Qeq) `` {z}
    by (auto elim!: quotientE simp: classes_def)
  with SN Z ∗ have z ∈ Z z ≠ x z ≠ y (z, x) ∉ transymcl (insert (x, y) Qeq) (z, y) ∉ transymcl (insert
(x, y) Qeq)
    using classes_disjoint[of Z insert (x, y) Qeq insert y X] classes_nonempty[of insert (x, y) Qeq]
    by (auto simp: class_None_eq class_Some_eq Field_transymcl_self transymcl_into(2)
      intro: transymcl_trans)
   with SN Z ∗ have transymcl (insert (x, y) Qeq) `` {z} = transymcl Qeq `` {z}
    unfolding transymcl_insert
    by (intro trans[OF Un1_Image_triv]) (auto simp: class_None_eq class_Some_eq)
   with z ‹z ≠ x› ‹z ≠ y› have Z ∈ classes Qeq
    by (auto simp: classes_def intro!: quotientI)
  }
  ultimately have classes (insert (x, y) Qeq) = classes Qeq − {X} ∪ {insert y X}
   by blast
  with SN show ?case
   by auto
next
  case (SS X Y)
  moreover from SS have XY: X ∈ classes Qeq Y ∈ classes Qeq x ∈ X y ∈ Y x ∈ Field Qeq y ∈ Field
Qeq
```

```
  using class_None_eq[of x Qeq] class_None_eq[of y Qeq] class_Some_eq[of x Qeq X] class_Some_eq[of
y Qeq Y]
    by auto
  moreover from XY have X = Y ⟹ classes (insert (x, y) Qeq) = classes Qeq
    unfolding classes_def
    by (subst transymcl_insert_same_class)
      (auto simp: classes_def insert_absorb elim!: quotientE intro: transymcl_sym transymcl_trans)
  moreover
  {
    assume neq: X ≠ Y
    from XY have X = transymcl Qeq `` {x} Y = transymcl Qeq `` {y}
      by (auto simp: classes_def elim!: quotientE intro: transymcl_sym transymcl_trans)
    with XY have XY_eq:
      X ∪ Y = transymcl (insert (x, y) Qeq) `` {x}
      X ∪ Y = transymcl (insert (x, y) Qeq) `` {y}
      unfolding transymcl_insert by auto
    then have ∗: X ∪ Y ∈ classes (insert (x, y) Qeq)
      by (auto simp: classes_def quotientI)
    moreover
    from ∗ XY neq have ∗∗: X ∉ classes (insert (x, y) Qeq) Y ∉ classes (insert (x, y) Qeq)
      using classes_disjoint[OF ∗, of X] classes_disjoint[OF ∗, of Y] classes_disjoint[of X Qeq Y]
      by auto
    moreover {
      fix Z
      assume Z: Z ≠ X Z ≠ Y Z ∈ classes Qeq
      then obtain z where z: z ∈ Field Qeq Z = transymcl Qeq `` {z}
        by (auto elim!: quotientE simp: classes_def)
      with XY Z have z ∈ Z z ≠ x z ≠ y (z, x) ∉ transymcl Qeq (z, y) ∉ transymcl Qeq
        using classes_disjoint[of Z Qeq X] classes_disjoint[of Z Qeq Y] classes_nonempty[of Qeq]
        by (auto simp: disjoint_iff Field_transymcl_self dest: FieldI2 intro: transymcl_trans)
      with XY Z ∗ have transymcl Qeq `` {z} = transymcl (insert (x, y) Qeq) `` {z}
        unfolding transymcl_insert
        by (intro trans[OF _ Un1_Image_triv[symmetric]]) (auto simp: class_None_eq class_Some_eq)
      with z have Z ∈ classes (insert (x, y) Qeq)
        by (auto simp: classes_def intro!: quotientI)
    }
    moreover {
      fix Z
      assume Z: Z ≠ X ∪ Y Z ∈ classes (insert (x, y) Qeq)
      then obtain z where z: z ∈ Field (insert (x, y) Qeq) Z = transymcl (insert (x, y) Qeq) `` {z}
        by (auto elim!: quotientE simp: classes_def)
      with XY Z neq XY_eq have z ∈ Z z ≠ x z ≠ y (z, x) ∉ transymcl (insert (x, y) Qeq) (z, y) ∉
transymcl (insert (x, y) Qeq)
        using classes_disjoint[OF ∗, of Z] classes_disjoint[of X Qeq Y]
        by (auto simp: Field_transymcl_self)
      with XY Z ∗ have transymcl (insert (x, y) Qeq) `` {z} = transymcl Qeq `` {z}
        unfolding transymcl_insert
        by (intro trans[OF Un1_Image_triv]) (auto simp: class_None_eq class_Some_eq)
      with z ‹z ≠ x› ‹z ≠ y› have Z ∈ classes Qeq
        by (auto simp: classes_def intro!: quotientI)
    }
    ultimately have classes (insert (x, y) Qeq) = classes Qeq − {X, Y} ∪ {X ∪ Y}
      by blast
  }
  ultimately show ?case
    by auto
qed
```

lemma *classes_intersect_find_not_None*:
  assumes $\forall\, V{\in}classes\ (set\ xys).\ V \cap A \neq \{\}\ xys \neq []$
  shows *find* $(\lambda(x,\ y).\ x \in A \vee y \in A)\ xys \neq None$
proof −
  from *assms(2)* **obtain** *x y* **where** $(x,\ y) \in set\ xys$ **by** (*cases xys*) *auto*
  with *assms(1)* **obtain** *X* **where** *x*: *class x* (*set xys*) = *Some X X* ∩ *A* ≠ {}
    using *ex_class*[*of x set xys*]
    by (*auto simp*: *class_Some_eq Field_def*)
  then obtain *a* **where** *a* ∈ *A a* ∈ *X*
    by *blast*
  with *x* **have** (*a*, *x*) ∈ *transymcl* (*set xys*)
    using *equiv_class_eq*[*OF equiv_transymcl*, *of* _ _ *set xys*]
    by (*fastforce simp*: *class_Some_eq classes_def elim*!: *quotientE*)
  then obtain *b* **where** (*a*, *b*) ∈ *symcl* (*set xys*)
    by (*auto simp*: *transymcl_def elim*: *converse_tranclE*)
  with ‹*a* ∈ *A*› **show** *?thesis*
    by (*auto simp*: *find_None_iff symcl_def*)
qed

2 Relational Calculus

2.1 First-order Terms

datatype *'a term* = *Const 'a* | *Var nat*

type_synonym *'a val* = *nat* ⇒ *'a*

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

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

definition *fv_terms_set* :: (*'a term*) *list* ⇒ *nat set* **where**
  fv_terms_set ts = ⋃(*set* (*map fv_term_set ts*))

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

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

lemma *finite_set_term*: *finite* (*set_term t*)
  by (*cases t*) *auto*

lemma *finite_fv_term_set*: *finite* (*fv_term_set t*)
  by (*cases t*) *auto*

lemma *fv_term_setD*: *n* ∈ *fv_term_set t* ⟹ *t* = *Var n*
  by (*cases t*) *auto*

lemma *fv_term_set_cong*: *fv_term_set t* = *fv_term_set* (*map_term f t*)
  by (*cases t*) *auto*

lemma *fv_terms_setI*: *Var m* ∈ *set ts* ⟹ *m* ∈ *fv_terms_set ts*
  by (*induction ts*) (*auto simp*: *fv_terms_set_def*)

lemma *fv_terms_setD*: *m* ∈ *fv_terms_set ts* ⟹ *Var m* ∈ *set ts*
  by (*induction ts*) (*auto simp*: *fv_terms_set_def dest*: *fv_term_setD*)

lemma *finite_fv_terms_set*: *finite* (*fv_terms_set ts*)
  by (*auto simp*: *fv_terms_set_def finite_fv_term_set*)

lemma *fv_terms_set_cong*: *fv_terms_set ts* = *fv_terms_set* (*map* (*map_term f*) *ts*)
  using *fv_term_set_cong*
  by (*induction ts*) (*fastforce simp*: *fv_terms_set_def*)+

lemma *eval_term_cong*: (⋀*n*. *n* ∈ *fv_term_set t* ⟹ *σ n* = *σ*′ *n*) ⟹
  eval_term σ t = *eval_term σ*′ *t*
  by (*cases t*) *auto*

lemma *eval_terms_fv_terms_set*: *σ* ⊙ *ts* = *σ*′ ⊙ *ts* ⟹ *n* ∈ *fv_terms_set ts* ⟹ *σ n* = *σ*′ *n*
proof (*induction ts*)
  case (*Cons t ts*)
  then show *?case*
    by (*cases t*) (*auto simp*: *eval_terms_def fv_terms_set_def*)
qed (*auto simp*: *eval_terms_def fv_terms_set_def*)

lemma *eval_terms_cong*: (⋀*n*. *n* ∈ *fv_terms_set ts* ⟹ *σ n* = *σ*′ *n*) ⟹
  eval_terms σ ts = *eval_terms σ*′ *ts*
  by (*auto simp*: *eval_terms_def fv_terms_set_def intro*: *eval_term_cong*)

2.2 Relational Calculus Syntax and Semantics

datatype (*discs_sels*) ('*a*, '*b*) *fmla* =
  Pred '*b* ('*a term*) *list*
| *Bool bool*
| *Eq nat* '*a term*
| *Neg* ('*a*, '*b*) *fmla*
| *Conj* ('*a*, '*b*) *fmla* ('*a*, '*b*) *fmla*
| *Disj* ('*a*, '*b*) *fmla* ('*a*, '*b*) *fmla*
| *Exists nat* ('*a*, '*b*) *fmla*

derive *linorder term*
derive *linorder fmla*

fun *fv* :: ('*a*, '*b*) *fmla* ⇒ *nat set* **where**
  fv (*Pred* _ *ts*) = *fv_terms_set ts*
| *fv* (*Bool b*) = {}
| *fv* (*Eq x t*′) = {*x*} ∪ *fv_term_set t*′
| *fv* (*Neg φ*) = *fv φ*
| *fv* (*Conj φ ψ*) = *fv φ* ∪ *fv ψ*
| *fv* (*Disj φ ψ*) = *fv φ* ∪ *fv ψ*
| *fv* (*Exists z φ*) = *fv φ* − {*z*}

definition *exists* **where** *exists x Q* = (*if x* ∈ *fv Q then Exists x Q else Q*)
abbreviation *Forall x Q* ≡ *Neg* (*Exists x* (*Neg Q*))
abbreviation *forall x Q* ≡ *Neg* (*exists x* (*Neg Q*))
abbreviation *Impl Q1 Q2* ≡ *Disj* (*Neg Q1*) *Q2*

definition *EXISTS xs Q* = *fold Exists xs Q*

abbreviation *close* **where**
  close Q ≡ *EXISTS* (*sorted_list_of_set* (*fv Q*)) *Q*

lemma *fv_exists*[*simp*]: *fv* (*exists x Q*) = *fv Q* − {*x*}
  by (*auto simp*: *exists_def*)

lemma *fv_EXISTS*: *fv* (*EXISTS xs Q*) = *fv Q* − *set xs*
  by (*induct xs arbitrary*: *Q*) (*auto simp*: *EXISTS_def*)

lemma *exists_Exists*: *x* ∈ *fv Q* ⟹ *exists x Q* = *Exists x Q*
  by (*auto simp*: *exists_def*)

lemma *is_Bool_exists*[*simp*]: *is_Bool* (*exists x Q*) = *is_Bool Q*
  by (*auto simp*: *exists_def is_Bool_def*)

lemma *finite_fv*[*simp*]: *finite* (*fv φ*)
  by (*induction φ rule*: *fv.induct*)
    (*auto simp*: *finite_fv_term_set finite_fv_terms_set*)

lemma *fv_close*[*simp*]: *fv* (*close Q*) = {}
  by (*subst fv_EXISTS*) *auto*

type_synonym *'a table* = (*'a list*) *set*
type_synonym (*'a*, *'b*) *intp* = *'b* × *nat* ⇒ *'a table*

definition *adom* :: (*'a*, *'b*) *intp* ⇒ *'a set* **where**
  adom I = (⋃ *rn*. ⋃ *xs* ∈ *I rn*. *set xs*)

fun *sat* :: (*'a*, *'b*) *fmla* ⇒ (*'a*, *'b*) *intp* ⇒ *'a val* ⇒ *bool* **where**
  sat (*Pred r ts*) *I σ* ⟷ *σ* ⊙ *ts* ∈ *I* (*r*, *length ts*)
| *sat* (*Bool b*) *I σ* ⟷ *b*
| *sat* (*Eq x t'*) *I σ* ⟷ *σ x* = *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 z φ*) *I σ* ⟷ (∃ *x*. *sat φ I* (*σ*(*z* := *x*)))

lemma *sat_fv_cong*: (⋀*n*. *n* ∈ *fv φ* ⟹ *σ n* = *σ' n*) ⟹
  sat φ I σ ⟷ *sat φ I σ'*
proof (*induction φ arbitrary*: *σ σ'*)
  case (*Neg φ*)
  show *?case*
    using *Neg*(*1*)[*of σ σ'*] *Neg*(*2*)
    by *auto*
next
  case (*Conj φ ψ*)
  show *?case*
    using *Conj*(*1*,*2*)[*of σ σ'*] *Conj*(*3*)
    by *auto*
next
  case (*Disj φ ψ*)
  show *?case*
    using *Disj*(*1*,*2*)[*of σ σ'*] *Disj*(*3*)
    by *auto*
next
  case (*Exists n φ*)
  have ⋀*x*. *sat φ I* (*σ*(*n* := *x*)) = *sat φ I* (*σ'*(*n* := *x*))
    using *Exists*(*2*)

by (*auto intro*!: *Exists*(*1*))
then show *?case*
by *simp*
qed (*auto cong*: *eval_terms_cong eval_term_cong*)

lemma *sat_fun_upd*: $n \notin fv\ Q \Longrightarrow sat\ Q\ I\ (\sigma(n := z)) = sat\ Q\ I\ \sigma$
by (*rule sat_fv_cong*) *auto*

lemma *sat_exists*[*simp*]: $sat\ (exists\ n\ Q)\ I\ \sigma = (\exists x.\ sat\ Q\ I\ (\sigma(n := x)))$
by (*auto simp add*: *exists_def sat_fun_upd*)

abbreviation *eq* (**infix** ‹≈› *80*) **where**
$x \approx y \equiv Eq\ x\ (Var\ y)$

definition *equiv* (**infix** ‹≜› *100*) **where**
$Q1 \triangleq Q2 = (\forall I\ \sigma.\ finite\ (adom\ I) \longrightarrow sat\ Q1\ I\ \sigma \longleftrightarrow sat\ Q2\ I\ \sigma)$

lemma *equiv_refl*[*iff*]: $Q \triangleq Q$
unfolding *equiv_def* **by** *auto*

lemma *equiv_sym*[*sym*]: $Q1 \triangleq Q2 \Longrightarrow Q2 \triangleq Q1$
unfolding *equiv_def* **by** *auto*

lemma *equiv_trans*[*trans*]: $Q1 \triangleq Q2 \Longrightarrow Q2 \triangleq Q3 \Longrightarrow Q1 \triangleq Q3$
unfolding *equiv_def* **by** *auto*

lemma *equiv_Neg_cong*[*simp*]: $Q \triangleq Q' \Longrightarrow Neg\ Q \triangleq Neg\ Q'$
unfolding *equiv_def* **by** *auto*

lemma *equiv_Conj_cong*[*simp*]: $Q1 \triangleq Q1' \Longrightarrow Q2 \triangleq Q2' \Longrightarrow Conj\ Q1\ Q2 \triangleq Conj\ Q1'\ Q2'$
unfolding *equiv_def* **by** *auto*

lemma *equiv_Disj_cong*[*simp*]: $Q1 \triangleq Q1' \Longrightarrow Q2 \triangleq Q2' \Longrightarrow Disj\ Q1\ Q2 \triangleq Disj\ Q1'\ Q2'$
unfolding *equiv_def* **by** *auto*

lemma *equiv_Exists_cong*[*simp*]: $Q \triangleq Q' \Longrightarrow Exists\ x\ Q \triangleq Exists\ x\ Q'$
unfolding *equiv_def* **by** *auto*

lemma *equiv_Exists_exists_cong*[*simp*]: $Q \triangleq Q' \Longrightarrow Exists\ x\ Q \triangleq exists\ x\ Q'$
unfolding *equiv_def* **by** *auto*

lemma *equiv_Exists_Disj*: $Exists\ x\ (Disj\ Q1\ Q2) \triangleq Disj\ (Exists\ x\ Q1)\ (Exists\ x\ Q2)$
unfolding *equiv_def* **by** *auto*

lemma *equiv_Disj_Assoc*: $Disj\ (Disj\ Q1\ Q2)\ Q3 \triangleq Disj\ Q1\ (Disj\ Q2\ Q3)$
unfolding *equiv_def* **by** *auto*

lemma *foldr_Disj_equiv_cong*[*simp*]:
$list\_all2\ (\triangleq)\ xs\ ys \Longrightarrow b \triangleq c \Longrightarrow foldr\ Disj\ xs\ b \triangleq foldr\ Disj\ ys\ c$
by (*induct xs ys arbitrary*: *b c rule*: *list.rel_induct*) *auto*

lemma *Exists_nonfree_equiv*: $x \notin fv\ Q \Longrightarrow Exists\ x\ Q \triangleq Q$
unfolding *equiv_def sat.simps*
by (*metis exists_def sat_exists*)

2.3 Constant Propagation

fun *cp* **where**

```
  cp (Eq x t) = (case t of Var y ⇒ if x = y then Bool True else x ≈ y | _ ⇒ Eq x t)
| cp (Neg Q) = (let Q' = cp Q in if is_Bool Q' then Bool (¬ un_Bool Q') else Neg Q')
| cp (Conj Q1 Q2) =
    (let Q1' = cp Q1; Q2' = cp Q2 in
    if is_Bool Q1' then if un_Bool Q1' then Q2' else Bool False
    else if is_Bool Q2' then if un_Bool Q2' then Q1' else Bool False
    else Conj Q1' Q2')
| cp (Disj Q1 Q2) =
    (let Q1' = cp Q1; Q2' = cp Q2 in
    if is_Bool Q1' then if un_Bool Q1' then Bool True else Q2'
    else if is_Bool Q2' then if un_Bool Q2' then Bool True else Q1'
    else Disj Q1' Q2')
| cp (Exists x Q) = exists x (cp Q)
| cp Q = Q

lemma fv_cp: fv (cp Q) ⊆ fv Q
  by (induct Q) (auto simp: Let_def split: fmla.splits term.splits)

lemma cp_exists[simp]: cp (exists x Q) = exists x (cp Q)
  by (auto simp: exists_def fv_cp[THEN set_mp])

fun nocp where
  nocp (Bool b) = False
| nocp (Pred p ts) = True
| nocp (Eq x t) = (t ≠ Var x)
| nocp (Neg Q) = nocp Q
| nocp (Conj Q1 Q2) = (nocp Q1 ∧ nocp Q2)
| nocp (Disj Q1 Q2) = (nocp Q1 ∧ nocp Q2)
| nocp (Exists x Q) = (x ∈ fv Q ∧ nocp Q)

lemma nocp_exists[simp]: nocp (exists x Q) = nocp Q
  unfolding exists_def by auto

lemma nocp_cp_triv: nocp Q ⟹ cp Q = Q
  by (induct Q) (auto simp: exists_def is_Bool_def split: fmla.splits term.splits)

lemma is_Bool_cp_triv: is_Bool Q ⟹ cp Q = Q
  by (auto simp: is_Bool_def)

lemma nocp_cp_or_is_Bool: nocp (cp Q) ∨ is_Bool (cp Q)
  by (induct Q) (auto simp: Let_def split: fmla.splits term.splits)

lemma cp_idem[simp]: cp (cp Q) = cp Q
  using is_Bool_cp_triv nocp_cp_triv nocp_cp_or_is_Bool by blast

lemma sat_cp[simp]: sat (cp Q) I σ = sat Q I σ
  by (induct Q arbitrary: σ) (auto 0 0 simp: Let_def is_Bool_def split: term.splits fmla.splits)

lemma equiv_cp_cong[simp]: Q ≜ Q' ⟹ cp Q ≜ cp Q'
  by (auto simp: equiv_def)

lemma equiv_cp[simp]: cp Q ≜ Q
  by (auto simp: equiv_def)

definition cpropagated where cpropagated Q = (nocp Q ∨ is_Bool Q)

lemma cpropagated_cp[simp]: cpropagated (cp Q)
  by (auto simp: cpropagated_def nocp_cp_or_is_Bool)
```

lemma *nocp_cpropagated*[*simp*]: *nocp Q* $\Longrightarrow$ *cpropagated Q*
by (*auto simp*: *cpropagated_def*)

lemma *cpropagated_cp_triv*: *cpropagated Q* $\Longrightarrow$ *cp Q* = *Q*
by (*auto simp*: *cpropagated_def nocp_cp_triv is_Bool_def*)

lemma *cpropagated_nocp*: *cpropagated Q* $\Longrightarrow$ *x* $\in$ *fv Q* $\Longrightarrow$ *nocp Q*
by (*auto simp*: *cpropagated_def is_Bool_def*)

lemma *cpropagated_simps*[*simp*]:
cpropagated (*Bool b*) $\longleftrightarrow$ *True*
cpropagated (*Pred p ts*) $\longleftrightarrow$ *True*
cpropagated (*Eq x t*) $\longleftrightarrow$ *t* $\neq$ *Var x*
cpropagated (*Neg Q*) $\longleftrightarrow$ *nocp Q*
cpropagated (*Conj Q1 Q2*) $\longleftrightarrow$ *nocp Q1* $\wedge$ *nocp Q2*
cpropagated (*Disj Q1 Q2*) $\longleftrightarrow$ *nocp Q1* $\wedge$ *nocp Q2*
cpropagated (*Exists x Q*) $\longleftrightarrow$ *x* $\in$ *fv Q* $\wedge$ *nocp Q*
by (*auto simp*: *cpropagated_def*)

2.4 Big Disjunction

fun *foldr1* **where**
foldr1 f (*x* # *xs*) *z* = *foldr f xs x*
| *foldr1 f* [] *z* = *z*

definition *DISJ* **where**
DISJ G = *foldr1 Disj* (*sorted_list_of_set G*) (*Bool False*)

lemma *sat_foldr_Disj*[*simp*]: *sat* (*foldr Disj xs Q*) *I* σ = ($\exists$ *Q* $\in$ *set xs* $\cup$ {*Q*}. *sat Q I* σ)
by (*induct xs arbitrary*: *Q*) *auto*

lemma *sat_foldr1_Disj*[*simp*]: *sat* (*foldr1 Disj xs Q*) *I* σ = (*if xs* = [] *then sat Q I* σ *else* $\exists$ *Q* $\in$ *set xs*. *sat Q I* σ)
by (*cases xs*) *auto*

lemma *sat_DISJ*[*simp*]: *finite G* $\Longrightarrow$ *sat* (*DISJ G*) *I* σ = ($\exists$ *Q* $\in$ *G*. *sat Q I* σ)
unfolding *DISJ_def* **by** *auto*

lemma *foldr_Disj_equiv*: *insert Q* (*set Qs*) = *insert Q′* (*set Qs′*) $\Longrightarrow$ *foldr Disj Qs Q* $\triangleq$ *foldr Disj Qs′ Q′*
by (*auto simp*: *equiv_def set_eq_iff*)

lemma *foldr1_Disj_equiv*: *set Qs* = *set Qs′* $\Longrightarrow$ *foldr1 Disj Qs* (*Bool False*) $\triangleq$ *foldr1 Disj Qs′* (*Bool False*)
by (*cases Qs*; *cases Qs′*) (*auto simp*: *foldr_Disj_equiv*)

lemma *foldr1_Disj_equiv_cong*[*simp*]:
list_all2 ($\triangleq$) *xs ys* $\Longrightarrow$ *b* $\triangleq$ *c* $\Longrightarrow$ *foldr1 Disj xs b* $\triangleq$ *foldr1 Disj ys c*
by (*erule list.rel_cases*) *auto*

lemma *Exists_foldr_Disj*:
Exists x (*foldr Disj xs b*) $\triangleq$ *foldr Disj* (*map* (*exists x*) *xs*) (*exists x b*)
by (*auto simp*: *equiv_def*)

lemma *Exists_foldr1_Disj*:
Exists x (*foldr1 Disj xs b*) $\triangleq$ *foldr1 Disj* (*map* (*exists x*) *xs*) (*exists x b*)
by (*auto simp*: *equiv_def*)

lemma *Exists_DISJ*:
finite $\mathcal{Q} \Longrightarrow$ *Exists x* (*DISJ* $\mathcal{Q}$) $\triangleq$ *DISJ* (*exists x* ‘ $\mathcal{Q}$)
unfolding *DISJ_def*
by (*rule equiv_trans*[*OF Exists_foldr1_Disj*])
(*auto simp*: *exists_def intro*!: *foldr1_Disj_equiv equiv_trans*[*OF _ equiv_sym*[*OF equiv_cp*]])

lemma *Exists_cp_DISJ*:
finite $\mathcal{Q} \Longrightarrow$ *Exists x* (*cp* (*DISJ* $\mathcal{Q}$)) $\triangleq$ *DISJ* (*exists x* ‘ $\mathcal{Q}$)
by (*rule equiv_trans*[*OF equiv_Exists_cong*[*OF equiv_cp*] *Exists_DISJ*])

lemma *Disj_empty*[*simp*]: *DISJ* {} = *Bool False*
unfolding *DISJ_def* **by** *auto*
lemma *Disj_single*[*simp*]: *DISJ* {*x*} = *x*
unfolding *DISJ_def* **by** *auto*

lemma *DISJ_insert*[*simp*]: *finite X* $\Longrightarrow$ *DISJ* (*insert x X*) $\triangleq$ *Disj x* (*DISJ X*)
by (*induct X arbitrary*: *x rule*: *finite_induct*) (*auto simp*: *equiv_def*)

lemma *DISJ_union*[*simp*]: *finite X* $\Longrightarrow$ *finite Y* $\Longrightarrow$ *DISJ* ($X \cup Y$) $\triangleq$ *Disj* (*DISJ X*) (*DISJ Y*)
by (*induct X rule*: *finite_induct*)
(*auto intro*!: *DISJ_insert*[*THEN equiv_trans*] *simp*: *equiv_def*)

lemma *DISJ_exists_pull_out*: *finite* $\mathcal{Q} \Longrightarrow Q \in \mathcal{Q} \Longrightarrow$
DISJ (*exists x* ‘ $\mathcal{Q}$) $\triangleq$ *Disj* (*Exists x Q*) (*DISJ* (*exists x* ‘ ($\mathcal{Q}$ − {*Q*})))
by (*auto simp*: *equiv_def*)

lemma *DISJ_push_in*: *finite* $\mathcal{Q} \Longrightarrow$ *Disj Q* (*DISJ* $\mathcal{Q}$) $\triangleq$ *DISJ* (*insert Q* $\mathcal{Q}$)
by (*auto simp*: *equiv_def*)

lemma *DISJ_insert_reorder*: *finite* $\mathcal{Q} \Longrightarrow$ *DISJ* (*insert* (*Disj Q1 Q2*) $\mathcal{Q}$) $\triangleq$ *DISJ* (*insert Q2* (*insert Q1* $\mathcal{Q}$))
by (*auto simp*: *equiv_def*)

lemma *DISJ_insert_reorder′*: *finite* $\mathcal{Q} \Longrightarrow$ *finite* $\mathcal{Q}' \Longrightarrow$ *DISJ* (*insert* (*Disj* (*DISJ* $\mathcal{Q}'$) *Q2*) $\mathcal{Q}$) $\triangleq$ *DISJ* (*insert Q2* ($\mathcal{Q}' \cup \mathcal{Q}$))
by (*auto simp*: *equiv_def*)

lemma *fv_foldr_Disj*[*simp*]: *fv* (*foldr Disj Qs Q*) = (*fv Q* ∪ ($\bigcup Q \in$ *set Qs*. *fv Q*))
by (*induct Qs*) *auto*

lemma *fv_foldr1_Disj*[*simp*]: *fv* (*foldr1 Disj Qs Q*) = (*if Qs* = [] *then fv Q else* ($\bigcup Q \in$ *set Qs*. *fv Q*))
by (*cases Qs*) *auto*

lemma *fv_DISJ*: *finite* $\mathcal{Q} \Longrightarrow$ *fv* (*DISJ* $\mathcal{Q}$) ⊆ ($\bigcup Q \in \mathcal{Q}$. *fv Q*)
by (*auto simp*: *DISJ_def dest*!: *fv_cp*[*THEN set_mp*] *split*: *if_splits*)

lemma *fv_DISJ_close*[*simp*]: *finite* $\mathcal{Q} \Longrightarrow$ *fv* (*DISJ* (*close* ‘ $\mathcal{Q}$)) = {}
by (*auto dest*!: *fv_DISJ*[*THEN set_mp*, *rotated 1*])

lemma *fv_cp_foldr_Disj*: ∀ *Q*∈*set Qs* ∪ {*Q*}. *cpropagated Q* ∧ *fv Q* = *A* $\Longrightarrow$ *fv* (*cp* (*foldr Disj Qs Q*)) = *A*
by (*induct Qs*) (*auto simp*: *cpropagated_cp_triv Let_def is_Bool_def*)

lemma *fv_cp_foldr1_Disj*: *cp* (*foldr1 Disj Qs* (*Bool False*)) ≠ *Bool False* $\Longrightarrow$
∀ *Q*∈*set Qs*. *cpropagated Q* ∧ *fv Q* = *A* $\Longrightarrow$
fv (*cp* (*foldr1 Disj Qs* (*Bool False*))) = *A*
by (*cases Qs*) (*auto simp*: *fv_cp_foldr_Disj*)

lemma *fv_cp_DISJ_eq*: *finite* $\mathcal{Q} \Longrightarrow$ *cp* (*DISJ* $\mathcal{Q}$) $\neq$ *Bool False* $\Longrightarrow \forall\, Q \in \mathcal{Q}$. *cpropagated* $Q \wedge$ *fv* $Q =$ $A \Longrightarrow$ *fv* (*cp* (*DISJ* $\mathcal{Q}$)) $= A$
by (*auto simp*: *DISJ_def fv_cp_foldr1_Disj*)

fun *sub* **where**
sub (*Bool t*) = {*Bool t*}
| *sub* (*Pred p ts*) = {*Pred p ts*}
| *sub* (*Eq x t*) = {*Eq x t*}
| *sub* (*Neg Q*) = *insert* (*Neg Q*) (*sub Q*)
| *sub* (*Conj Q1 Q2*) = *insert* (*Conj Q1 Q2*) (*sub Q1* ∪ *sub Q2*)
| *sub* (*Disj Q1 Q2*) = *insert* (*Disj Q1 Q2*) (*sub Q1* ∪ *sub Q2*)
| *sub* (*Exists z Q*) = *insert* (*Exists z Q*) (*sub Q*)

lemma *cpropagated_sub*: *cpropagated* $Q \Longrightarrow Q' \in$ *sub* $Q \Longrightarrow$ *cpropagated* Q'
by (*induct Q*) *auto*

lemma *Exists_in_sub_cp_foldr_Disj*:
Exists x $Q' \in$ *sub* (*cp* (*foldr Disj Qs Q*)) $\Longrightarrow$ *Exists x* $Q' \in$ *sub* (*cp Q*) $\vee$ ($\exists\, Q \in$ *set Qs*. *Exists x* $Q' \in$ *sub* (*cp Q*))
by (*induct Qs arbitrary*: *Q*) (*auto simp*: *Let_def split*: *if_splits*)

lemma *Exists_in_sub_cp_foldr1_Disj*:
Exists x $Q' \in$ *sub* (*cp* (*foldr1 Disj Qs Q*)) $\Longrightarrow$ *Qs* = [] $\wedge$ *Exists x* $Q' \in$ *sub* (*cp Q*) $\vee$ ($\exists\, Q \in$ *set Qs*. *Exists x* $Q' \in$ *sub* (*cp Q*))
by (*cases Qs*) (*auto simp*: *Exists_in_sub_cp_foldr_Disj*)

lemma *Exists_in_sub_cp_DISJ*: *Exists x* $Q' \in$ *sub* (*cp* (*DISJ* $\mathcal{Q}$)) $\Longrightarrow$ *finite* $\mathcal{Q} \Longrightarrow$ ($\exists\, Q \in \mathcal{Q}$. *Exists x* $Q' \in$ *sub* (*cp Q*))
unfolding *DISJ_def* **by** (*drule Exists_in_sub_cp_foldr1_Disj*) *auto*

lemma *Exists_in_sub_foldr_Disj*:
Exists x $Q' \in$ *sub* (*foldr Disj Qs Q*) $\Longrightarrow$ *Exists x* $Q' \in$ *sub Q* $\vee$ ($\exists\, Q \in$ *set Qs*. *Exists x* $Q' \in$ *sub Q*)
by (*induct Qs arbitrary*: *Q*) (*auto simp*: *Let_def split*: *if_splits*)

lemma *Exists_in_sub_foldr1_Disj*:
Exists x $Q' \in$ *sub* (*foldr1 Disj Qs Q*) $\Longrightarrow$ *Qs* = [] $\wedge$ *Exists x* $Q' \in$ *sub Q* $\vee$ ($\exists\, Q \in$ *set Qs*. *Exists x* Q' $\in$ *sub Q*)
by (*cases Qs*) (*auto simp*: *Exists_in_sub_foldr_Disj*)

lemma *Exists_in_sub_DISJ*: *Exists x* $Q' \in$ *sub* (*DISJ* $\mathcal{Q}$) $\Longrightarrow$ *finite* $\mathcal{Q} \Longrightarrow$ ($\exists\, Q \in \mathcal{Q}$. *Exists x* $Q' \in$ *sub Q*)
unfolding *DISJ_def* **by** (*drule Exists_in_sub_foldr1_Disj*) *auto*

2.5 Substitution

fun *subst_term* (‹_[_ →*t* _]› [*90*, *0*, *0*] *91*) **where**
Var z[*x* →*t y*] = *Var* (*if x* = *z then y else z*)
| *Const c*[*x* →*t y*] = *Const c*

abbreviation *substs_term* (‹_[_ →*t** _]› [*90*, *0*, *0*] *91*) **where**
t[*xs* →*t** *ys*] ≡ *fold* (λ(*x*, *y*) *t*. *t*[*x* →*t y*]) (*zip xs ys*) *t*

lemma *size_subst_term*[*simp*]: *size* (*t*[*x* →*t y*]) = *size t*
by (*cases t*) *auto*

lemma *fv_subst_term*[*simp*]: *fv_term_set* (*t*[*x* →*t y*]) =
(*if x* ∈ *fv_term_set t then insert y* (*fv_term_set t* − {*x*}) *else fv_term_set t*)

by (*cases t*) *auto*

definition *fresh2 x y Q* = *Suc* (*Max* (*insert x* (*insert y* (*fv Q*))))

function (*sequential*) *subst* :: ('*a*, '*b*) *fmla* ⇒ *nat* ⇒ *nat* ⇒ ('*a*, '*b*) *fmla* (‹_[_ → _]› [*90*, *0*, *0*] *91*)
where
Bool t[*x* → *y*] = *Bool t*
| *Pred p ts*[*x* → *y*] = *Pred p* (*map* (λ*t*. *t*[*x* →*t y*]) *ts*)
| *Eq z t*[*x* → *y*] = *Eq* (*if z* = *x then y else z*) (*t*[*x* →*t y*])
| *Neg Q*[*x* → *y*] = *Neg* (*Q*[*x* → *y*])
| *Conj Q1 Q2*[*x* → *y*] = *Conj* (*Q1*[*x* → *y*]) (*Q2*[*x* → *y*])
| *Disj Q1 Q2*[*x* → *y*] = *Disj* (*Q1*[*x* → *y*]) (*Q2*[*x* → *y*])
| *Exists z Q*[*x* → *y*] = (*if x* = *z then Exists x Q else*
if z = *y then let z*′ = *fresh2 x y Q in Exists z*′ (*Q*[*z* → *z*′][*x* → *y*]) *else Exists z* (*Q*[*x* → *y*]))
by *pat_completeness auto*

abbreviation *substs* (‹_[_ →* _]› [*90*, *0*, *0*] *91*) **where**
Q[*xs* →* *ys*] ≡ *fold* (λ(*x*, *y*) *Q*. *Q*[*x* → *y*]) (*zip xs ys*) *Q*

lemma *size_subst_p*[*simp*]: *subst_dom* (*Q*, *x*, *y*) ⟹ *size* (*Q*[*x* → *y*]) = *size Q*
by (*induct Q x y rule*: *subst.pinduct*) (*auto simp*: *subst.psimps o_def Let_def exists_def*)

termination by *lexicographic_order*

lemma *size_subst*[*simp*]: *size* (*Q*[*x* → *y*]) = *size Q*
by (*induct Q x y rule*: *subst.induct*) (*auto simp*: *o_def Let_def exists_def*)

lemma *fresh2_gt*:
x < *fresh2 x y Q*
y < *fresh2 x y Q*
z ∈ *fv Q* ⟹ *z* < *fresh2 x y Q*
unfolding *fresh2_def less_Suc_eq_le*
by (*auto simp*: *max_def Max_ge_iff*)

lemma *fresh2*:
x ≠ *fresh2 x y Q*
y ≠ *fresh2 x y Q*
fresh2 x y Q ∉ *fv Q*
using *fresh2_gt*(*1*)[*of x y Q*] *fresh2_gt*(*2*)[*of y x Q*] *fresh2_gt*(*3*)[*of fresh2 x y Q Q x y*]
by *auto*

lemma *fv_subst*:
fv (*Q*[*x* → *y*]) = (*if x* ∈ *fv Q then insert y* (*fv Q* − {*x*}) *else fv Q*)
by (*induct Q x y rule*: *subst.induct*)
(*auto simp*: *fv_terms_set_def Let_def fresh2 split*: *if_splits*)

lemma *subst_term_triv*: *x* ∉ *fv_term_set t* ⟹ *t*[*x* →*t y*] = *t*
by (*cases t*) *auto*

lemma *subst_exists*: *exists z Q*[*x* → *y*] = (*if z* ∈ *fv Q then if x* = *z then exists x Q else*
if z = *y then let z*′ = *fresh2 x y Q in exists z*′ (*Q*[*z* → *z*′][*x* → *y*]) *else exists z* (*Q*[*x* → *y*]) *else Q*[*x* → *y*])
by (*auto simp*: *exists_def Let_def fv_subst fresh2 dest*: *sym*)

lemma *eval_subst*[*simp*]: *σ* · *t*[*x* →*t y*] = *σ*(*x* := *σ y*) · *t*
by (*cases t*) *auto*

lemma *sat_subst*[*simp*]: *sat* (*Q*[*x* → *y*]) *I σ* = *sat Q I* (*σ*(*x* := *σ y*))

by (*induct Q x y arbitrary*: σ *rule*: *subst.induct*)
(*auto 0 3 simp*: *eval_terms_def o_def Let_def fun_upd_twist*[*symmetric*] *sat_fun_upd fresh2 dest*: *sym*)

lemma *substs_Bool*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ *Bool b*[*xs* $\rightarrow^*$ *ys*] = *Bool b*
by (*induct xs ys rule*: *list_induct2*) *auto*

lemma *substs_Neg*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ *Neg Q*[*xs* $\rightarrow^*$ *ys*] = *Neg* (*Q*[*xs* $\rightarrow^*$ *ys*])
by (*induct xs ys arbitrary*: *Q rule*: *list_induct2*) (*auto simp*: *Let_def*)

lemma *substs_Conj*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ *Conj Q1 Q2*[*xs* $\rightarrow^*$ *ys*] = *Conj* (*Q1*[*xs* $\rightarrow^*$ *ys*]) (*Q2*[*xs* $\rightarrow^*$ *ys*])
by (*induct xs ys arbitrary*: *Q1 Q2 rule*: *list_induct2*) *auto*

lemma *substs_Disj*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ *Disj Q1 Q2*[*xs* $\rightarrow^*$ *ys*] = *Disj* (*Q1*[*xs* $\rightarrow^*$ *ys*]) (*Q2*[*xs* $\rightarrow^*$ *ys*])
by (*induct xs ys arbitrary*: *Q1 Q2 rule*: *list_induct2*) *auto*

fun *substs_bd* **where**
substs_bd z (*x* # *xs*) (*y* # *ys*) *Q* = (*if x* = *z then substs_bd z xs ys Q else*
if z = *y then substs_bd* (*fresh2 x y Q*) *xs ys* (*Q*[*y* $\rightarrow$ *fresh2 x y Q*][*x* $\rightarrow$ *y*]) *else substs_bd z xs ys* (*Q*[*x* $\rightarrow$ *y*]))
| *substs_bd z* _ _ _ = *z*

fun *substs_src* **where**
substs_src z (*x* # *xs*) (*y* # *ys*) *Q* = (*if x* = *z then substs_src z xs ys Q else*
if z = *y then* [*y*, *x*] @ *substs_src* (*fresh2 x y Q*) *xs ys* (*Q*[*y* $\rightarrow$ *fresh2 x y Q*][*x* $\rightarrow$ *y*]) *else x* # *substs_src z xs ys* (*Q*[*x* $\rightarrow$ *y*]))
| *substs_src* _ _ _ _ = []

fun *substs_dst* **where**
substs_dst z (*x* # *xs*) (*y* # *ys*) *Q* = (*if x* = *z then substs_dst z xs ys Q else*
if z = *y then* [*fresh2 x y Q*, *y*] @ *substs_dst* (*fresh2 x y Q*) *xs ys* (*Q*[*y* $\rightarrow$ *fresh2 x y Q*][*x* $\rightarrow$ *y*]) *else y* # *substs_dst z xs ys* (*Q*[*x* $\rightarrow$ *y*]))
| *substs_dst* _ _ _ _ = []

lemma *length_substs*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ *length* (*substs_src z xs ys Q*) = *length* (*substs_dst z xs ys Q*)
by (*induct xs ys arbitrary*: *z Q rule*: *list_induct2*) *auto*

lemma *substs_Exists*[*simp*]: *length xs* = *length ys* $\Longrightarrow$
Exists z Q[*xs* $\rightarrow^*$ *ys*] = *Exists* (*substs_bd z xs ys Q*) (*Q*[*substs_src z xs ys Q* $\rightarrow^*$ *substs_dst z xs ys Q*])
by (*induct xs ys arbitrary*: *Q z rule*: *list_induct2*) (*auto simp*: *Let_def intro*: *exI*[*of* _ []])

fun *subst_var* **where**
subst_var (*x* # *xs*) (*y* # *ys*) *z* = (*if x* = *z then subst_var xs ys y else subst_var xs ys z*)
| *subst_var* _ _ *z* = *z*

lemma *substs_Eq*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ (*Eq x t*)[*xs* $\rightarrow^*$ *ys*] = *Eq* (*subst_var xs ys x*) (*t*[*xs* $\rightarrow t^*$ *ys*])
by (*induct xs ys arbitrary*: *x t rule*: *list_induct2*) *auto*

lemma *substs_term_Var*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ (*Var x*)[*xs* $\rightarrow t^*$ *ys*] = *Var* (*subst_var xs ys x*)
by (*induct xs ys arbitrary*: *x rule*: *list_induct2*) *auto*

lemma *substs_term_Const*[*simp*]: *length xs* = *length ys* $\Longrightarrow$ (*Const c*)[*xs* $\rightarrow t^*$ *ys*] = *Const c*
by (*induct xs ys rule*: *list_induct2*) *auto*

lemma *in_fv_substs*:
length xs = length ys ⟹ x ∈ fv Q ⟹ subst_var xs ys x ∈ fv (Q[xs → ys])*
by (*induct xs ys arbitrary*: *x Q rule*: *list_induct2*) (*auto simp*: *fv_subst*)

lemma *exists_cp_subst*: *x ≠ y ⟹ exists x (cp (Q[x → y])) = cp (Q[x → y])*
by (*auto simp*: *exists_def fv_subst dest*!: *set_mp*[*OF fv_cp*] *split*: *if_splits*)

2.6 Generated Variables

inductive *ap* **where**
Pred: *ap (Pred p ts)*
| *Eqc*: *ap (Eq x (Const c))*

inductive *gen* **where**
gen x (Bool False) {}
| *ap Q ⟹ x ∈ fv Q ⟹ gen x Q {Q}*
| *gen x Q G ⟹ gen x (Neg (Neg Q)) G*
| *gen x (Conj (Neg Q1) (Neg Q2)) G ⟹ gen x (Neg (Disj Q1 Q2)) G*
| *gen x (Disj (Neg Q1) (Neg Q2)) G ⟹ gen x (Neg (Conj Q1 Q2)) G*
| *gen x Q1 G1 ⟹ gen x Q2 G2 ⟹ gen x (Disj Q1 Q2) (G1 ∪ G2)*
| *gen x Q1 G ∨ gen x Q2 G ⟹ gen x (Conj Q1 Q2) G*
| *gen y Q G ⟹ gen x (Conj Q (x ≈ y)) ((λQ. cp (Q[y → x])) ‘ G)*
| *gen y Q G ⟹ gen x (Conj Q (y ≈ x)) ((λQ. cp (Q[y → x])) ‘ G)*
| *x ≠ y ⟹ gen x Q G ⟹ gen x (Exists y Q) (exists y ‘ G)*

inductive *gen′* **where**
gen′ x (Bool False) {}
| *ap Q ⟹ x ∈ fv Q ⟹ gen′ x Q {Q}*
| *gen′ x Q G ⟹ gen′ x (Neg (Neg Q)) G*
| *gen′ x (Conj (Neg Q1) (Neg Q2)) G ⟹ gen′ x (Neg (Disj Q1 Q2)) G*
| *gen′ x (Disj (Neg Q1) (Neg Q2)) G ⟹ gen′ x (Neg (Conj Q1 Q2)) G*
| *gen′ x Q1 G1 ⟹ gen′ x Q2 G2 ⟹ gen′ x (Disj Q1 Q2) (G1 ∪ G2)*
| *gen′ x Q1 G ∨ gen′ x Q2 G ⟹ gen′ x (Conj Q1 Q2) G*
| *gen′ y Q G ⟹ gen′ x (Conj Q (x ≈ y)) ((λQ. Q[y → x]) ‘ G)*
| *gen′ y Q G ⟹ gen′ x (Conj Q (y ≈ x)) ((λQ. Q[y → x]) ‘ G)*
| *x ≠ y ⟹ gen′ x Q G ⟹ gen′ x (Exists y Q) (exists y ‘ G)*

inductive *qp* **where**
ap: *ap Q ⟹ qp Q*
| *exists*: *qp Q ⟹ qp (exists x Q)*

lemma *qp_Exists*: *qp Q ⟹ x ∈ fv Q ⟹ qp (Exists x Q)*
by (*metis qp.exists exists_def*)

lemma *qp_ExistsE*: *qp (Exists x Q) ⟹ (qp Q ⟹ x ∈ fv Q ⟹ R) ⟹ R*
by (*induct Exists x Q rule*: *qp.induct*) (*auto elim*!: *ap.cases simp*: *exists_def split*: *if_splits*)

fun *qp_impl* **where**
qp_impl (Eq x (Const c)) = True
| *qp_impl (Pred x ts) = True*
| *qp_impl (Exists x Q) = (x ∈ fv Q ∧ qp Q)*
| *qp_impl _ = False*

lemma *qp_imp_qp_impl*: *qp Q ⟹ qp_impl Q*
by (*induct Q rule*: *qp.induct*) (*auto elim*!: *ap.cases simp*: *exists_def*)

lemma *qp_impl_imp_qp*: *qp_impl Q ⟹ qp Q*
by (*induct Q rule*: *qp_impl.induct*) (*auto intro*: *ap.intros qp_Exists qp.ap*)

```
lemma qp_code[code]: qp Q = qp_impl Q
  using qp_imp_qp_impl qp_impl_imp_qp by blast

lemma ap_cp: ap Q ⟹ ap (cp Q)
  by (induct Q rule: ap.induct) (auto intro: ap.intros)

lemma qp_cp: qp Q ⟹ qp (cp Q)
  by (induct Q rule: qp.induct) (auto intro: qp.intros ap_cp)

lemma ap_substs: ap Q ⟹ length xs = length ys ⟹ ap (Q[xs →* ys])
proof (induct Q arbitrary: xs ys rule: ap.induct)
  case (Pred p ts)
  then show ?case
    by (induct xs ys arbitrary: ts rule: list_induct2) (auto intro!: ap.intros)
next
  case (Eqc x c)
  then show ?case
    by (induct xs ys arbitrary: x rule: list_induct2) (auto intro!: ap.intros)
qed

lemma ap_subst': ap (Q[x → y]) ⟹ ap Q
proof (induct Q[x → y] arbitrary: Q rule: ap.induct)
  case (Pred p ts)
  then show ?case
    by (cases Q) (auto simp: Let_def split: if_splits intro: ap.intros)
next
  case (Eqc x c)
  then show ?case
  proof (cases Q)
    case (Eq x t)
    with Eqc show ?thesis
      by (cases t) (auto intro: ap.intros)
  qed (auto simp: Let_def split: if_splits)
qed

lemma qp_substs: qp Q ⟹ length xs = length ys ⟹ qp (Q[xs →* ys])
proof (induct Q arbitrary: xs ys rule: qp.induct)
  case (ap Q)
  then show ?case
    by (rule qp.ap[OF ap_substs])
next
  case (exists Q z)
  from exists(3,1,2) show ?case
  proof (induct xs ys arbitrary: Q z rule: list_induct2)
    case Nil
    then show ?case
      by (auto intro: qp.intros)
  next
    case (Cons x xs y ys)
    have [simp]: Q[x → y][xs →* ys] = Q[x # xs →* y # ys] for Q :: ('a, 'b) fmla and x y xs ys
      by auto
    have IH1[simp]: qp (Q[x → y]) for x y
      using Cons(4)[of [x] [y]] by auto
    have IH2[simp]: qp (Q[x → y][a → b]) for x y a b
      using Cons(4)[of [x, a] [y, b]] by auto
    note zip_Cons_Cons[simp del]
    show ?case
```

unfolding *zip_Cons_Cons fold.simps prod.case o_apply subst_exists* **using** *Cons(1,3)*
by (*auto simp*: *Let_def intro*!: *qp.intros(2) Cons(2,4)*)
qed
qed

lemma *qp_subst*: *qp Q* ⟹ *qp* (*Q*[*x* → *y*])
using *qp_substs*[*of Q* [*x*] [*y*]] **by** *auto*

lemma *qp_Neg*[*dest*]: *qp* (*Neg Q*) ⟹ *False*
by (*rule qp.induct*[**where** *P* = λ*Q'*. *Q'* = *Neg Q* ⟶ *False*, *THEN mp*]) (*auto elim*!: *ap.cases simp*: *exists_def*)

lemma *qp_Disj*[*dest*]: *qp* (*Disj Q1 Q2*) ⟹ *False*
by (*rule qp.induct*[**where** *P* = λ*Q*. *Q* = *Disj Q1 Q2* ⟶ *False*, *THEN mp*]) (*auto elim*!: *ap.cases simp*: *exists_def*)

lemma *qp_Conj*[*dest*]: *qp* (*Conj Q1 Q2*) ⟹ *False*
by (*rule qp.induct*[**where** *P* = λ*Q*. *Q* = *Conj Q1 Q2* ⟶ *False*, *THEN mp*]) (*auto elim*!: *ap.cases simp*: *exists_def*)

lemma *qp_eq*[*dest*]: *qp* (*x* ≈ *y*) ⟹ *False*
by (*rule qp.induct*[**where** *P* = λ*Q*. (∃ *x y*. *Q* = *x* ≈ *y*) ⟶ *False*, *THEN mp*]) (*auto elim*!: *ap.cases simp*: *exists_def*)

lemma *qp_subst'*: *qp* (*Q*[*x* → *y*]) ⟹ *qp Q*
proof (*induct Q x y rule*: *subst.induct*)
case (*3 z t x y*)
then show *?case*
by (*cases t*) (*auto intro*!: *ap Eqc split*: *if_splits*)
qed (*auto 0 3 simp*: *qp_Exists fv_subst Let_def fresh2 Pred ap dest*: *sym elim*!: *qp_ExistsE split*: *if_splits*)

lemma *qp_subst_eq*[*simp*]: *qp* (*Q*[*x* → *y*]) = *qp Q*
using *qp_subst qp_subst'* **by** *blast*

lemma *gen_qp*: *gen x Q G* ⟹ *Qqp* ∈ *G* ⟹ *qp Qqp*
by (*induct x Q G arbitrary*: *Qqp rule*: *gen.induct*) (*auto intro*: *qp.intros ap.intros qp_cp*)

lemma *gen'_qp*: *gen' x Q G* ⟹ *Qqp* ∈ *G* ⟹ *qp Qqp*
by (*induct x Q G arbitrary*: *Qqp rule*: *gen'.induct*) (*auto intro*: *qp.intros ap.intros*)

lemma *ap_cp_triv*: *ap Q* ⟹ *cp Q* = *Q*
by (*induct Q rule*: *ap.induct*) *auto*

lemma *qp_cp_triv*: *qp Q* ⟹ *cp Q* = *Q*
by (*induct Q rule*: *qp.induct*) (*auto simp*: *ap_cp_triv*)

lemma *ap_cp_subst_triv*: *ap Q* ⟹ *cp* (*Q*[*x* → *y*]) = *Q*[*x* → *y*]
by (*induct Q rule*: *ap.induct*) *auto*

lemma *qp_cp_subst_triv*: *qp Q* ⟹ *cp* (*Q*[*x* → *y*]) = *Q*[*x* → *y*]
by (*induct Q rule*: *qp.induct*)
(*auto simp*: *exists_def qp_cp_triv Let_def fv_subst fresh2 ap_cp_subst_triv dest*: *sym*)

lemma *gen_nocp_intros*:
gen y Q G ⟹ *gen x* (*Conj Q* (*x* ≈ *y*)) ((λ*Q*. *Q*[*y* → *x*]) ‘ *G*)
gen y Q G ⟹ *gen x* (*Conj Q* (*y* ≈ *x*)) ((λ*Q*. *Q*[*y* → *x*]) ‘ *G*)
by (*metis* (*no_types*, *lifting*) *gen.intros*(*8*) *gen_qp image_cong qp_cp_subst_triv*,
metis (*no_types*, *lifting*) *gen.intros*(*9*) *gen_qp image_cong qp_cp_subst_triv*)

lemma *gen′_cp_intros*:
gen′ y Q G $\Longrightarrow$ *gen′ x* (*Conj Q* (*x* $\approx$ *y*)) ((λ*Q*. *cp* (*Q*[*y* $\rightarrow$ *x*])) ‘ *G*)
gen′ y Q G $\Longrightarrow$ *gen′ x* (*Conj Q* (*y* $\approx$ *x*)) ((λ*Q*. *cp* (*Q*[*y* $\rightarrow$ *x*])) ‘ *G*)
by (*metis* (*no_types*, *lifting*) *gen′.intros*(*8*) *gen′_qp image_cong qp_cp_subst_triv*,
metis (*no_types*, *lifting*) *gen′.intros*(*9*) *gen′_qp image_cong qp_cp_subst_triv*)

lemma *gen′_gen*: *gen′ x Q G* $\Longrightarrow$ *gen x Q G*
by (*induct x Q G rule*: *gen′.induct*) (*auto intro*!: *gen.intros gen_nocp_intros*)

lemma *gen_gen′*: *gen x Q G* $\Longrightarrow$ *gen′ x Q G*
by (*induct x Q G rule*: *gen.induct*) (*auto intro*!: *gen′.intros gen′_cp_intros*)

lemma *gen_eq_gen′*: *gen* = *gen′*
using *gen′_gen gen_gen′* **by** *blast*

lemmas *gen_induct*[*consumes 1*] = *gen′.induct*[*folded gen_eq_gen′*]

abbreviation *Gen* **where** *Gen x Q* ≡ (∃ *G*. *gen x Q G*)

lemma *qp_Gen*: *qp Q* $\Longrightarrow$ *x* ∈ *fv Q* $\Longrightarrow$ *Gen x Q*
by (*induct Q rule*: *qp.induct*) (*force simp*: *exists_def intro*: *gen.intros*)+

lemma *qp_gen*: *qp Q* $\Longrightarrow$ *x* ∈ *fv Q* $\Longrightarrow$ *gen x Q* {*Q*}
by (*induct Q rule*: *qp.induct*)
(*force simp*: *exists_def intro*: *gen.intros dest*: *gen.intros*(*10*))+

lemma *gen_foldr_Disj*:
list_all2 (*gen x*) *Qs Gs* $\Longrightarrow$ *gen x Q G* $\Longrightarrow$ *GG* = *G* ∪ (⋃ *G* ∈ *set Gs*. *G*) $\Longrightarrow$
gen x (*foldr Disj Qs Q*) *GG*
proof (*induct Qs Gs arbitrary*: *Q G GG rule*: *list.rel_induct*)
case (*Cons Q′ Qs G′ Gs*)
then have *GG*: *GG* = *G′* ∪ (*G* ∪ (⋃ *G* ∈ *set Gs*. *G*))
by *auto*
from *Cons*(*1*,*3*−) **show** *?case*
unfolding *foldr.simps o_apply GG*
by (*intro gen.intros Cons*(*2*)[*OF* _ *refl*]) *auto*
qed *simp*

lemma *gen_foldr1_Disj*:
list_all2 (*gen x*) *Qs Gs* $\Longrightarrow$ *gen x Q G* $\Longrightarrow$ *GG* = (*if Qs* = [] *then G else* (⋃ *G* ∈ *set Gs*. *G*)) $\Longrightarrow$
gen x (*foldr1 Disj Qs Q*) *GG*
by (*erule list.rel_cases*) (*auto simp*: *gen_foldr_Disj*)

lemma *gen_Bool_True*[*simp*]: *gen x* (*Bool True*) *G* = *False*
by (*auto elim*: *gen.cases*)

lemma *gen_Bool_False*[*simp*]: *gen x* (*Bool False*) *G* = (*G* = {})
by (*auto elim*: *gen.cases intro*: *gen.intros*)

lemma *gen_Gen_cp*: *gen x Q G* $\Longrightarrow$ *Gen x* (*cp Q*)
by (*induct x Q G rule*: *gen_induct*)
(*auto split*: *if_splits simp*: *Let_def ap_cp_triv is_Bool_def exists_def intro*: *gen.intros*)

lemma *Gen_cp*: *Gen x Q* $\Longrightarrow$ *Gen x* (*cp Q*)
by (*metis gen_Gen_cp*)

lemma *Gen_DISJ*: *finite* 𝒬 $\Longrightarrow$ ∀ *Q* ∈ 𝒬. *qp Q* ∧ *x* ∈ *fv Q* $\Longrightarrow$ *Gen x* (*DISJ* 𝒬)

unfolding *DISJ_def*
by (*rule exI gen_foldr1_Disj*[**where** *Gs*=*map* (λ*Q*. {*Q*}) (*sorted_list_of_set* $\mathcal{Q}$) **and** *G*={}])+
(*auto simp*: *list.rel_map qp_cp_triv qp_gen gen.intros intro*!: *list.rel_refl_strong*)

lemma *Gen_cp_DISJ*: *finite* $\mathcal{Q} \Longrightarrow \forall\, Q \in \mathcal{Q}.\ qp\ Q \wedge x \in fv\ Q \Longrightarrow Gen\ x\ (cp\ (DISJ\ \mathcal{Q}))$
by (*rule Gen_cp Gen_DISJ*)+

lemma *gen_Pred*[*simp*]:
gen z (*Pred p ts*) *G* $\longleftrightarrow$ *z* $\in$ *fv_terms_set ts* $\wedge$ *G* = {*Pred p ts*}
by (*auto elim*: *gen.cases intro*: *gen.intros ap.intros*)

lemma *gen_Eq*[*simp*]:
gen z (*Eq a t*) *G* $\longleftrightarrow$ *z* = *a* $\wedge$ ($\exists$ *c*. *t* = *Const c* $\wedge$ *G* = {*Eq a t*})
by (*auto elim*: *gen.cases elim*!: *ap.cases intro*: *gen.intros ap.intros*)

lemma *gen_empty_cp*: *gen z Q G* $\Longrightarrow$ *G* = {} $\Longrightarrow$ *cp Q* = *Bool False*
by (*induct z Q G rule*: *gen_induct*)
(*fastforce simp*: *Let_def exists_def split*: *if_splits*)+

inductive *genempty* **where**
genempty (*Bool False*)
| *genempty Q* $\Longrightarrow$ *genempty* (*Neg* (*Neg Q*))
| *genempty* (*Conj* (*Neg Q1*) (*Neg Q2*)) $\Longrightarrow$ *genempty* (*Neg* (*Disj Q1 Q2*))
| *genempty* (*Disj* (*Neg Q1*) (*Neg Q2*)) $\Longrightarrow$ *genempty* (*Neg* (*Conj Q1 Q2*))
| *genempty Q1* $\Longrightarrow$ *genempty Q2* $\Longrightarrow$ *genempty* (*Disj Q1 Q2*)
| *genempty Q1* $\vee$ *genempty Q2* $\Longrightarrow$ *genempty* (*Conj Q1 Q2*)
| *genempty Q* $\Longrightarrow$ *genempty* (*Conj Q* (*x* $\approx$ *y*))
| *genempty Q* $\Longrightarrow$ *genempty* (*Conj Q* (*y* $\approx$ *x*))
| *genempty Q* $\Longrightarrow$ *genempty* (*Exists y Q*)

lemma *gen_genempty*: *gen z Q G* $\Longrightarrow$ *G* = {} $\Longrightarrow$ *genempty Q*
by (*induct z Q G rule*: *gen.induct*) (*auto intro*: *genempty.intros*)

lemma *genempty_substs*: *genempty Q* $\Longrightarrow$ *length xs* = *length ys* $\Longrightarrow$ *genempty* (*Q*[*xs* $\rightarrow^*$ *ys*])
by (*induct Q arbitrary*: *xs ys rule*: *genempty.induct*) (*auto intro*: *genempty.intros*)

lemma *genempty_substs_Exists*: *genempty Q* $\Longrightarrow$ *length xs* = *length ys* $\Longrightarrow$ *genempty* (*Exists y Q*[*xs* $\rightarrow^*$ *ys*])
by (*auto intro*!: *genempty.intros genempty_substs*)

lemma *genempty_cp*: *genempty Q* $\Longrightarrow$ *cp Q* = *Bool False*
by (*induct Q rule*: *genempty.induct*)
(*auto simp*: *Let_def exists_def split*: *if_splits*)

lemma *gen_empty_cp_substs*:
gen x Q {} $\Longrightarrow$ *length xs* = *length ys* $\Longrightarrow$ *cp* (*Q*[*xs* $\rightarrow^*$ *ys*]) = *Bool False*
by (*rule genempty_cp*[*OF genempty_substs*[*OF gen_genempty*[*OF _ refl*]]])

lemma *gen_empty_cp_substs_Exists*:
gen x Q {} $\Longrightarrow$ *length xs* = *length ys* $\Longrightarrow$ *cp* (*Exists y Q*[*xs* $\rightarrow^*$ *ys*]) = *Bool False*
by (*rule genempty_cp*[*OF genempty_substs_Exists*[*OF gen_genempty*[*OF _ refl*]]])

lemma *gen_Gen_substs_Exists*:
length xs = *length ys* $\Longrightarrow$ *x* $\neq$ *y* $\Longrightarrow$ *x* $\in$ *fv Q* $\Longrightarrow$
($\bigwedge$*xs ys*. *length xs* = *length ys* $\Longrightarrow$ *Gen* (*subst_var xs ys x*) (*cp* (*Q*[*xs* $\rightarrow^*$ *ys*]))) $\Longrightarrow$
Gen (*subst_var xs ys x*) (*cp* (*Exists y Q*[*xs* $\rightarrow^*$ *ys*]))
proof (*induct xs ys arbitrary*: *y x Q rule*: *list_induct2*)
case *Nil*

```
  from Nil(1) Nil(3)[of [] []] show ?case
    by (auto simp: exists_def intro: gen.intros)
next
  case (Cons xx xs yy ys)
  have Gen (subst_var xs ys yy) (cp (Q[[y,x]@xs →* [fresh2 x y Q,yy]@ys]))
    if length xs = length ys and x ≠ y for xs ys
    using Cons(5)[of [y,x]@xs [fresh2 x y Q,yy]@ys] that Cons.prems by auto
  moreover have Gen (subst_var xs ys x) (cp (Q[[yy,xx]@xs →* [fresh2 xx yy Q,yy]@ys]))
    if length xs = length ys x ≠ yy x ≠ xx for xs ys
    using Cons(5)[of [yy,xx]@xs [fresh2 xx yy Q,yy]@ys] that Cons.prems by auto
  moreover have Gen (subst_var xs ys yy) (cp (Q[[x]@xs →* [yy]@ys]))
    if length xs = length ys and x = xx for xs ys
    using Cons(5)[of [x]@xs [yy]@ys] that Cons.prems by auto
  moreover have Gen (subst_var xs ys x) (cp (Q[[xx]@xs →* [yy]@ys]))
    if length xs = length ys and x ≠ xx for xs ys
    using Cons(5)[of [xx]@xs [yy]@ys] that Cons.prems by auto
  ultimately show ?case using Cons
    by (auto simp: Let_def fresh2 fv_subst intro: Cons(2) simp del: substs_Exists split: if_splits)
qed

lemma gen_fv:
  gen x Q G ⟹ Qqp ∈ G ⟹ x ∈ fv Qqp ∧ fv Qqp ⊆ fv Q
  by (induct x Q G arbitrary: Qqp rule: gen_induct)
    (force simp: fv_subst dest: fv_cp[THEN set_mp])+

lemma gen_sat:
  fixes x :: nat
  shows gen x Q G ⟹ sat Q I σ ⟹ ∃ Qqp ∈ G. sat Qqp I σ
  by (induct x Q G arbitrary: σ rule: gen_induct)
    (auto 6 0 simp: fun_upd_idem intro: UnI1 UnI2)
```

2.7 Variable Erasure

```
fun erase :: ('a, 'b) fmla ⇒ nat ⇒ ('a, 'b) fmla (infix ‹⊥› 65) where
  Bool t ⊥ x = Bool t
| Pred p ts ⊥ x = (if x ∈ fv_terms_set ts then Bool False else Pred p ts)
| Eq z t ⊥ x = (if t = Var z then Bool True else
    if x = z ∨ x ∈ fv_term_set t then Bool False else Eq z t)
| Neg Q ⊥ x = Neg (Q ⊥ x)
| Conj Q1 Q2 ⊥ x = Conj (Q1 ⊥ x) (Q2 ⊥ x)
| Disj Q1 Q2 ⊥ x = Disj (Q1 ⊥ x) (Q2 ⊥ x)
| Exists z Q ⊥ x = (if x = z then Exists x Q else Exists z (Q ⊥ x))

lemma fv_erase: fv (Q ⊥ x) ⊆ fv Q − {x}
  by (induct Q) auto

lemma ap_cp_erase: ap Q ⟹ x ∈ fv Q ⟹ cp (Q ⊥ x) = Bool False
  by (induct Q rule: ap.induct) auto

lemma qp_cp_erase: qp Q ⟹ x ∈ fv Q ⟹ cp (Q ⊥ x) = Bool False
  by (induct Q rule: qp.induct) (auto simp: exists_def ap_cp_erase split: if_splits)

lemma sat_erase: sat (Q ⊥ x) I (σ(x := z)) = sat (Q ⊥ x) I σ
  by (rule sat_fun_upd) (auto dest: fv_erase[THEN set_mp])

lemma exists_cp_erase: exists x (cp (Q ⊥ x)) = cp (Q ⊥ x)
  by (auto simp: exists_def dest: set_mp[OF fv_cp] set_mp[OF fv_erase])
```

lemma *gen_cp_erase*:
fixes x :: *nat*
shows *gen x Q G* $\Longrightarrow$ *Qqp* $\in$ *G* $\Longrightarrow$ *cp* (*Qqp* $\bot$ *x*) = *Bool False*
by (*metis gen_qp qp_cp_erase gen_fv*)

2.8 Generated Variables and Substitutions

lemma *gen_Gen_cp_substs*: *gen z Q G* $\Longrightarrow$ *length xs* = *length ys* $\Longrightarrow$
Gen (*subst_var xs ys z*) (*cp* (*Q*[*xs* $\rightarrow^*$ *ys*]))
proof (*induct z Q G arbitrary*: *xs ys rule*: *gen_induct*)
case (*2 Q x*)
show *?case*
by (*subst ap_cp_triv*) (*rule exI gen.intros*(*2*) *ap_substs 2 in_fv_substs*)+
next
case (*3 x Q G*)
then show *?case*
by (*fastforce simp*: *Let_def intro*: *gen.intros*)
next
case (*4 x Q1 Q2 G*)
from *4*(*2*)[*of xs ys*] *4*(*1*,*3*) **show** *?case*
by (*auto simp*: *Let_def is_Bool_def intro*!: *gen.intros*(*4*) *split*: *if_splits*)
next
case (*5 x Q1 Q2 G*)
from *5*(*2*)[*of xs ys*] *5*(*1*,*3*) **show** *?case*
by (*auto simp*: *Let_def is_Bool_def intro*!: *gen.intros*(*5*) *split*: *if_splits*)
next
case (*6 x Q1 G1 Q2 G2*)
from *6*(*2*,*4*)[*of xs ys*] *6*(*1*,*3*,*5*) **show** *?case*
by (*auto simp*: *Let_def is_Bool_def intro*!: *gen.intros*(*6*) *split*: *if_splits*)
next
case (*7 x Q1 G Q2*)
from *7*(*1*) **show** *?case*
proof (*elim disjE conjE*, *goal_cases L R*)
case *L*
from *L*(*1*) *L*(*2*)[*rule_format*, *of xs ys*] *7*(*2*) **show** *?case*
by (*auto simp*: *Let_def is_Bool_def intro*!: *gen.intros*(*7*) *split*: *if_splits*)
next
case *R*
from *R*(*1*) *R*(*2*)[*rule_format*, *of xs ys*] *7*(*2*) **show** *?case*
by (*auto simp*: *Let_def is_Bool_def intro*!: *gen.intros*(*7*) *split*: *if_splits*)
qed
next
case (*8 y Q G x*)
from *8*(*2*)[*of xs ys*] *8*(*1*,*3*) **show** *?case*
by (*auto simp*: *Let_def is_Bool_def intro*!: *gen.intros*(*8*) *split*: *if_splits*)
next
case (*9 y Q G x*)
from *9*(*2*)[*of xs ys*] *9*(*1*,*3*) **show** *?case*
by (*auto simp*: *Let_def is_Bool_def intro*!: *gen.intros*(*9*) *split*: *if_splits*)
next
case (*10 x y Q G*)
show *?case*
proof (*cases x* $\in$ *fv Q*)
case *True*
with *10*(*4*,*1*) **show** *?thesis* **using** *10*(*3*)
by (*rule gen_Gen_substs_Exists*)
next
case *False*

```
    with 10(2) have G = {}
      by (auto dest: gen_fv)
    with 10(2,4) have cp (Q[xs →* ys]) = Bool False
      by (auto intro!: gen_empty_cp_substs[of x])
    with 10(2,4) have cp (Exists y Q[xs →* ys]) = Bool False unfolding ‹G = {}›
      by (intro gen_empty_cp_substs_Exists)
    then show ?thesis
      by auto
  qed
qed (fastforce simp: Let_def is_Bool_def intro!: gen.intros split: if_splits)+

lemma Gen_cp_substs: Gen z Q ⟹ length xs = length ys ⟹ Gen (subst_var xs ys z) (cp (Q[xs →*
ys]))
  by (blast intro: gen_Gen_cp_substs)

lemma Gen_cp_subst: Gen z Q ⟹ z ≠ x ⟹ Gen z (cp (Q[x → y]))
  using Gen_cp_substs[of z Q [x] [y]] by auto

lemma substs_bd_fv: length xs = length ys ⟹ substs_bd z xs ys Q ∈ fv (Q[substs_src z xs ys Q →*
substs_dst z xs ys Q]) ⟹ z ∈ fv Q
proof (induct xs ys arbitrary: z Q rule: list_induct2)
  case (Cons x xs y ys)
  from Cons(1,3) show ?case
    by (auto 0 4 simp: fv_subst fresh2 dest: Cons(2) sym split: if_splits)
qed simp

lemma Gen_substs_bd: length xs = length ys ⟹
  (⋀xs ys. length xs = length ys ⟹ Gen (subst_var xs ys z) (cp (Qz[xs →* ys]))) ⟹
  Gen (substs_bd z xs ys Qz) (cp (Qz[substs_src z xs ys Qz →* substs_dst z xs ys Qz]))
proof (induct xs ys arbitrary: z Qz rule: list_induct2)
  case Nil
  from Nil(1)[of [] []] show ?case
    by simp
next
  case (Cons x xs y ys)
  have Gen (subst_var xs ys (fresh2 x y Qz)) (cp (Qz[y → fresh2 x y Qz][x → y][xs →* ys]))
    if length xs = length ys z = y for xs ys
    using that Cons(3)[of [y,x]@xs [fresh2 x y Qz,y]@ys]
    by (auto simp: fresh2)
  moreover have Gen (subst_var xs ys z) (cp (Qz[x → y][xs →* ys]))
    if length xs = length ys x ≠ z for xs ys
    using that Cons(3)[of [x]@xs [y]@ys]
    by (auto simp: fresh2)
  ultimately show ?case using Cons(1,3)
    by (auto intro!: Cons(2))
qed
```

2.9 Safe-Range Queries

```
definition nongens where
  nongens Q = {x ∈ fv Q. ¬ Gen x Q}

abbreviation rrf where
  rrf Q ≡ nongens Q = {}

definition rrb where
  rrb Q = (∀ y Qy. Exists y Qy ∈ sub Q ⟶ Gen y Qy)
```

lemma *rrb_simps*[*simp*]:
  rrb (*Bool b*) = *True*
  rrb (*Pred p ts*) = *True*
  rrb (*Eq x t*) = *True*
  rrb (*Neg Q*) = *rrb Q*
  rrb (*Disj Q1 Q2*) = (*rrb Q1* ∧ *rrb Q2*)
  rrb (*Conj Q1 Q2*) = (*rrb Q1* ∧ *rrb Q2*)
  rrb (*Exists y Qy*) = (*Gen y Qy* ∧ *rrb Qy*)
  rrb (*exists y Qy*) = ((*y* ∈ *fv Qy* ⟶ *Gen y Qy*) ∧ *rrb Qy*)
  by (*auto simp*: *rrb_def exists_def*)

lemma *ap_rrb*[*simp*]: *ap Q* ⟹ *rrb Q*
  by (*cases Q rule*: *ap.cases*) *auto*

lemma *qp_rrb*[*simp*]: *qp Q* ⟹ *rrb Q*
  by (*induct Q rule*: *qp.induct*) (*auto simp*: *qp_Gen*)

lemma *rrb_cp*: *rrb Q* ⟹ *rrb* (*cp Q*)
  by (*induct Q rule*: *cp.induct*)
    (*auto split*: *term.splits simp*: *Let_def exists_def Gen_cp dest*!: *fv_cp*[*THEN set_mp*])

lemma *gen_Gen_erase*: *gen x Q G* ⟹ *Gen x* (*Q* ⊥ *z*)
  by (*induct x Q G rule*: *gen_induct*)
    (*auto 0 4 intro*: *gen.intros qp.intros ap.intros elim*!: *ap.cases*)

lemma *Gen_erase*: *Gen x Q* ⟹ *Gen x* (*Q* ⊥ *z*)
  by (*metis gen_Gen_erase*)

lemma *rrb_erase*: *rrb Q* ⟹ *rrb* (*Q* ⊥ *x*)
  by (*induct Q x rule*: *erase.induct*)
    (*auto split*: *term.splits simp*: *Let_def exists_def Gen_erase dest*!: *fv_cp*[*THEN set_mp*])

lemma *rrb_DISJ*[*simp*]: *finite* 𝒬 ⟹ (∀ *Q* ∈ 𝒬. *rrb Q*) ⟹ *rrb* (*DISJ* 𝒬)
  by (*auto simp*: *rrb_def dest*!: *Exists_in_sub_DISJ*)

lemma *rrb_cp_substs*: *rrb Q* ⟹ *length xs* = *length ys* ⟹ *rrb* (*cp* (*Q*[*xs* →* *ys*]))
proof (*induct size Q arbitrary*: *Q xs ys rule*: *less_induct*)
  case *less*
  then show *?case*
  proof (*cases Q*)
    case (*Exists z Qz*)
    from *less*(*2*,*3*) **show** *?thesis*
      unfolding *Exists substs_Exists*[*OF less*(*3*)] *cp.simps rrb_simps*
      by (*intro conjI impI less*(*1*) *Gen_substs_bd Gen_cp_substs*) (*simp_all add*: *Exists*)
  qed (*auto simp*: *Let_def ap_cp ap_substs ap.intros split*: *term.splits*)
qed

lemma *rrb_cp_subst*: *rrb Q* ⟹ *rrb* (*cp* (*Q*[*x* → *y*]))
  using *rrb_cp_substs*[*of Q* [*x*] [*y*]]
  by *auto*

definition *sr Q* = (*rrf Q* ∧ *rrb Q*)

lemma *nongens_cp*: *nongens* (*cp Q*) ⊆ *nongens Q*
  unfolding *nongens_def* **by** (*auto dest*: *gen_Gen_cp fv_cp*[*THEN set_mp*])

lemma *sr_Disj*: *fv Q1* = *fv Q2* ⟹ *sr* (*Disj Q1 Q2*) = (*sr Q1* ∧ *sr Q2*)
  by (*auto 0 4 simp*: *sr_def nongens_def elim*!: *ap.cases elim*: *gen.cases intro*: *gen.intros*)

lemma *sr_foldr_Disj*: $\forall Q' \in set\ Qs.\ fv\ Q' = fv\ Q \implies sr\ (foldr\ Disj\ Qs\ Q) \longleftrightarrow (\forall Q \in set\ Qs.\ sr\ Q) \wedge sr\ Q$
by (*induct Qs*) (*auto simp*: *sr_Disj*)

lemma *sr_foldr1_Disj*: $\forall Q' \in set\ Qs.\ fv\ Q' = X \implies sr\ (foldr1\ Disj\ Qs\ Q) \longleftrightarrow (if\ Qs = []\ then\ sr\ Q\ else\ (\forall Q \in set\ Qs.\ sr\ Q))$
by (*cases Qs*) (*auto simp*: *sr_foldr_Disj*)

lemma *sr_False*[*simp*]: *sr* (*Bool False*)
by (*auto simp*: *sr_def nongens_def*)

lemma *sr_cp*: *sr Q* $\implies$ *sr* (*cp Q*)
by (*auto simp*: *rrb_cp sr_def dest*: *nongens_cp*[*THEN set_mp*])

lemma *sr_DISJ*: *finite* $\mathcal{Q} \implies \forall Q' \in \mathcal{Q}.\ fv\ Q' = X \implies (\forall Q \in \mathcal{Q}.\ sr\ Q) \implies sr\ (DISJ\ \mathcal{Q})$
by (*auto simp*: *DISJ_def sr_foldr1_Disj*[*of* _ *X*] *sr_cp*)

lemma *sr_Conj_eq*: *sr Q* $\implies x \in fv\ Q \vee y \in fv\ Q \implies$ *sr* (*Conj Q* ($x \approx y$))
by (*auto simp*: *sr_def nongens_def intro*: *gen.intros*)

2.10 Simplification

locale *simplification* =
fixes *simp* :: ('*a*::{*infinite*, *linorder*}, '*b* :: *linorder*) *fmla* $\Rightarrow$ ('*a*, '*b*) *fmla*
and *simplified* :: ('*a*, '*b*) *fmla* $\Rightarrow$ *bool*
assumes *sat_simp*: *sat* (*simp Q*) *I* σ = *sat Q I* σ
and *fv_simp*: *fv* (*simp Q*) $\subseteq$ *fv Q*
and *rrb_simp*: *rrb Q* $\implies$ *rrb* (*simp Q*)
and *gen_Gen_simp*: *gen x Q G* $\implies$ *Gen x* (*simp Q*)
and *fv_simp_Disj_same*: *fv* (*simp Q1*) = *X* $\implies$ *fv* (*simp Q2*) = *X* $\implies$ *fv* (*simp* (*Disj Q1 Q2*)) = *X*
and *simp_False*: *simp* (*Bool False*) = *Bool False*
and *simplified_sub*: *simplified Q* $\implies Q' \in$ *sub Q* $\implies$ *simplified Q'*
and *simplified_Conj_eq*: *simplified Q* $\implies x \neq y \implies x \in fv\ Q \vee y \in fv\ Q \implies$ *simplified* (*Conj Q* ($x \approx y$))
and *simplified_fv_simp*: *simplified Q* $\implies$ *fv* (*simp Q*) = *fv Q*
and *simplified_simp*: *simplified* (*simp Q*)
and *simplified_cp*: *simplified* (*cp Q*)
begin

lemma *Gen_simp*: *Gen x Q* $\implies$ *Gen x* (*simp Q*)
by (*metis gen_Gen_simp*)

lemma *nongens_simp*: *nongens* (*simp Q*) $\subseteq$ *nongens Q*
using *Gen_simp* **by** (*auto simp*: *nongens_def dest*!: *fv_simp*[*THEN set_mp*])

lemma *sr_simp*: *sr Q* $\implies$ *sr* (*simp Q*)
by (*auto simp*: *rrb_simp sr_def dest*: *nongens_simp*[*THEN set_mp*])

lemma *equiv_simp_cong*: *Q* $\triangleq$ *Q'* $\implies$ *simp Q* $\triangleq$ *simp Q'*
by (*auto simp*: *equiv_def sat_simp*)

lemma *equiv_simp*: *simp Q* $\triangleq$ *Q*
by (*auto simp*: *equiv_def sat_simp*)

lemma *fv_simp_foldr_Disj*: $\forall Q \in set\ Qs \cup \{Q\}.\ simplified\ Q \wedge fv\ Q = A \implies$
fv (*simp* (*foldr Disj Qs Q*)) = *A*

by (*induct Qs*) (*auto simp*: *Let_def is_Bool_def simplified_fv_simp fv_simp_Disj_same*)

lemma *fv_simp_foldr1_Disj*: *simp* (*foldr1 Disj Qs* (*Bool False*)) ≠ *Bool False* ⟹
∀ *Q*∈*set Qs*. *simplified Q* ∧ *fv Q* = *A* ⟹
fv (*simp* (*foldr1 Disj Qs* (*Bool False*))) = *A*
by (*cases Qs*) (*auto simp*: *fv_simp_foldr_Disj simp_False*)

lemma *fv_simp_DISJ_eq*:
finite 𝒬 ⟹ *simp* (*DISJ* 𝒬) ≠ *Bool False* ⟹ ∀ *Q* ∈ 𝒬. *simplified Q* ∧ *fv Q* = *A* ⟹ *fv* (*simp* (*DISJ* 𝒬)) = *A*
by (*auto simp*: *DISJ_def fv_simp_foldr1_Disj*)

end

2.11 Covered Variables

inductive *cov* **where**
Eq_self: *cov x* (*x* ≈ *x*) {}
| *nonfree*: *x* ∉ *fv Q* ⟹ *cov x Q* {}
| *EqL*: *x* ≠ *y* ⟹ *cov x* (*x* ≈ *y*) {*x* ≈ *y*}
| *EqR*: *x* ≠ *y* ⟹ *cov x* (*y* ≈ *x*) {*x* ≈ *y*}
| *ap*: *ap Q* ⟹ *x* ∈ *fv Q* ⟹ *cov x Q* {*Q*}
| *Neg*: *cov x Q G* ⟹ *cov x* (*Neg Q*) *G*
| *Disj*: *cov x Q1 G1* ⟹ *cov x Q2 G2* ⟹ *cov x* (*Disj Q1 Q2*) (*G1* ∪ *G2*)
| *DisjL*: *cov x Q1 G* ⟹ *cp* (*Q1* ⊥ *x*) = *Bool True* ⟹ *cov x* (*Disj Q1 Q2*) *G*
| *DisjR*: *cov x Q2 G* ⟹ *cp* (*Q2* ⊥ *x*) = *Bool True* ⟹ *cov x* (*Disj Q1 Q2*) *G*
| *Conj*: *cov x Q1 G1* ⟹ *cov x Q2 G2* ⟹ *cov x* (*Conj Q1 Q2*) (*G1* ∪ *G2*)
| *ConjL*: *cov x Q1 G* ⟹ *cp* (*Q1* ⊥ *x*) = *Bool False* ⟹ *cov x* (*Conj Q1 Q2*) *G*
| *ConjR*: *cov x Q2 G* ⟹ *cp* (*Q2* ⊥ *x*) = *Bool False* ⟹ *cov x* (*Conj Q1 Q2*) *G*
| *Exists*: *x* ≠ *y* ⟹ *cov x Q G* ⟹ *x* ≈ *y* ∉ *G* ⟹ *cov x* (*Exists y Q*) (*exists y* ‘ *G*)
| *Exists_gen*: *x* ≠ *y* ⟹ *cov x Q G* ⟹ *gen y Q Gy* ⟹ *cov x* (*Exists y Q*) ((*exists y* ‘ (*G* − {*x* ≈ *y*})) ∪ ((λ*Q*. *cp* (*Q*[*y* → *x*])) ‘ *Gy*))

inductive *cov′* **where**
Eq_self: *cov′ x* (*x* ≈ *x*) {}
| *nonfree*: *x* ∉ *fv Q* ⟹ *cov′ x Q* {}
| *EqL*: *x* ≠ *y* ⟹ *cov′ x* (*x* ≈ *y*) {*x* ≈ *y*}
| *EqR*: *x* ≠ *y* ⟹ *cov′ x* (*y* ≈ *x*) {*x* ≈ *y*}
| *ap*: *ap Q* ⟹ *x* ∈ *fv Q* ⟹ *cov′ x Q* {*Q*}
| *Neg*: *cov′ x Q G* ⟹ *cov′ x* (*Neg Q*) *G*
| *Disj*: *cov′ x Q1 G1* ⟹ *cov′ x Q2 G2* ⟹ *cov′ x* (*Disj Q1 Q2*) (*G1* ∪ *G2*)
| *DisjL*: *cov′ x Q1 G* ⟹ *cp* (*Q1* ⊥ *x*) = *Bool True* ⟹ *cov′ x* (*Disj Q1 Q2*) *G*
| *DisjR*: *cov′ x Q2 G* ⟹ *cp* (*Q2* ⊥ *x*) = *Bool True* ⟹ *cov′ x* (*Disj Q1 Q2*) *G*
| *Conj*: *cov′ x Q1 G1* ⟹ *cov′ x Q2 G2* ⟹ *cov′ x* (*Conj Q1 Q2*) (*G1* ∪ *G2*)
| *ConjL*: *cov′ x Q1 G* ⟹ *cp* (*Q1* ⊥ *x*) = *Bool False* ⟹ *cov′ x* (*Conj Q1 Q2*) *G*
| *ConjR*: *cov′ x Q2 G* ⟹ *cp* (*Q2* ⊥ *x*) = *Bool False* ⟹ *cov′ x* (*Conj Q1 Q2*) *G*
| *Exists*: *x* ≠ *y* ⟹ *cov′ x Q G* ⟹ *x* ≈ *y* ∉ *G* ⟹ *cov′ x* (*Exists y Q*) (*exists y* ‘ *G*)
| *Exists_gen*: *x* ≠ *y* ⟹ *cov′ x Q G* ⟹ *gen y Q Gy* ⟹ *cov′ x* (*Exists y Q*) ((*exists y* ‘ (*G* − {*x* ≈ *y*})) ∪ ((λ*Q*. *Q*[*y* → *x*]) ‘ *Gy*))

lemma *cov_nocp_intros*:
x ≠ *y* ⟹ *cov x Q G* ⟹ *gen y Q Gy* ⟹ *cov x* (*Exists y Q*) ((*exists y* ‘ (*G* − {*x* ≈ *y*})) ∪ ((λ*Q*. *Q*[*y* → *x*]) ‘ *Gy*))
by (*metis* (*no_types*, *lifting*) *cov.Exists_gen gen_qp image_cong qp_cp_subst_triv*)

lemma *cov′_cp_intros*:
x ≠ *y* ⟹ *cov′ x Q G* ⟹ *gen y Q Gy* ⟹ *cov′ x* (*Exists y Q*) ((*exists y* ‘ (*G* − {*x* ≈ *y*})) ∪ ((λ*Q*. *cp* (*Q*[*y* → *x*])) ‘ *Gy*))

by (*metis* (*no_types*, *lifting*) *cov'.Exists_gen gen_qp image_cong qp_cp_subst_triv*)

lemma *cov'_cov*: *cov' x Q G* $\Longrightarrow$ *cov x Q G*
by (*induct x Q G rule*: *cov'.induct*) (*force intro*: *cov.intros cov_nocp_intros*)+

lemma *cov_cov'*: *cov x Q G* $\Longrightarrow$ *cov' x Q G*
by (*induct x Q G rule*: *cov.induct*) (*force intro*: *cov'.intros cov'_cp_intros*)+

lemma *cov_eq_cov'*: *cov* = *cov'*
using *cov'_cov cov_cov'* **by** *blast*

lemmas *cov_induct*[*consumes 1*, *case_names Eq_self nonfree EqL EqR ap Neg Disj DisjL DisjR Conj ConjL ConjR Exists Exists_gen*] =
cov'.induct[*folded cov_eq_cov'*]

lemma *ex_cov*: *rrb Q* $\Longrightarrow$ *x* $\in$ *fv Q* $\Longrightarrow$ $\exists$ *G*. *cov x Q G*
proof (*induct Q*)
case (*Eq z t*)
then show *?case*
by (*cases t*) (*auto 6 0 intro*: *cov.intros ap.intros*)
next
case (*Exists z Q*)
then obtain *G Gz* **where** *cov x Q G gen z Q Gz x* $\neq$ *z*
by *force*
then show *?case*
by (*cases x* $\approx$ *z* $\in$ *G*) (*auto intro*: *cov.intros*)
qed (*auto intro*: *cov.intros ap.intros*)

definition *qps* **where**
qps G = {*Q* $\in$ *G*. *qp Q*}

lemma *qps_qp*: *Q* $\in$ *qps G* $\Longrightarrow$ *qp Q*
by (*auto simp*: *qps_def*)

lemma *qps_in*: *Q* $\in$ *qps G* $\Longrightarrow$ *Q* $\in$ *G*
by (*auto simp*: *qps_def*)

lemma *qps_empty*[*simp*]: *qps* {} = {}
by (*auto simp*: *qps_def*)

lemma *qps_insert*: *qps* (*insert Q Qs*) = (*if qp Q then insert Q* (*qps Qs*) *else qps Qs*)
by (*auto simp*: *qps_def*)

lemma *qps_union*[*simp*]: *qps* (*X* $\cup$ *Y*) = *qps X* $\cup$ *qps Y*
by (*auto simp*: *qps_def*)

lemma *finite_qps*[*simp*]: *finite G* $\Longrightarrow$ *finite* (*qps G*)
by (*auto simp*: *qps_def*)

lemma *qps_exists*[*simp*]: *x* $\neq$ *y* $\Longrightarrow$ *qps* (*exists y* ‘ *G*) = *exists y* ‘ *qps G*
by (*auto simp*: *qps_def image_iff exists_def qp_Exists elim*: *qp_ExistsE*)

lemma *qps_subst*[*simp*]: *qps* ((*λQ*. *Q*[*x* $\rightarrow$ *y*]) ‘ *G*) = (*λQ*. *Q*[*x* $\rightarrow$ *y*]) ‘ *qps G*
by (*auto simp*: *qps_def image_iff exists_def*)

lemma *qps_minus*[*simp*]: *qps* (*G* − {*x* $\approx$ *y*}) = *qps G*
by (*auto simp*: *qps_def*)

lemma *gen_qps*[*simp*]: *gen x Q G* $\Longrightarrow$ *qps G* = *G*
by (*auto dest*: *gen_qp simp*: *qps_def*)

lemma *qps_rrb*[*simp*]: *Q* $\in$ *qps G* $\Longrightarrow$ *rrb Q*
by (*auto simp*: *qps_def*)

definition *eqs* **where**
eqs x G = {*y*. *x* $\neq$ *y* $\wedge$ *x* $\approx$ *y* $\in$ *G*}

lemma *eqs_in*: *y* $\in$ *eqs x G* $\Longrightarrow$ *x* $\approx$ *y* $\in$ *G*
by (*auto simp*: *eqs_def*)

lemma *eqs_noteq*: *y* $\in$ *eqs x Q* $\Longrightarrow$ *x* $\neq$ *y*
unfolding *eqs_def* **by** *auto*

lemma *eqs_empty*[*simp*]: *eqs x* {} = {}
by (*auto simp*: *eqs_def*)

lemma *eqs_union*[*simp*]: *eqs x* (*X* $\cup$ *Y*) = *eqs x X* $\cup$ *eqs x Y*
by (*auto simp*: *eqs_def*)

lemma *finite_eqs*[*simp*]: *finite G* $\Longrightarrow$ *finite* (*eqs x G*)
by (*force simp*: *eqs_def image_iff elim*!: *finite_surj*[**where** *f* = λ*Q*. *SOME y*. *Q* = *x* $\approx$ *y*])

lemma *eqs_exists*[*simp*]: *x* $\neq$ *y* $\Longrightarrow$ *eqs x* (*exists y* ‘ *G*) = *eqs x G* − {*y*}
by (*auto simp*: *eqs_def exists_def image_iff*)

lemma *notin_eqs*[*simp*]: *x* $\approx$ *y* $\notin$ *G* $\Longrightarrow$ *y* $\notin$ *eqs x G*
by (*auto simp*: *eqs_def*)

lemma *eqs_minus*[*simp*]: *eqs x* (*G* − {*x* $\approx$ *y*}) = *eqs x G* − {*y*}
by (*auto simp*: *eqs_def*)

lemma *Var_eq_subst_iff*: *Var z* = *t*[*x* $\rightarrow$*t y*] $\longleftrightarrow$ (*if z* = *x then x* = *y* $\wedge$ *t* = *Var x else*
if z = *y then t* = *Var x* $\vee$ *t* = *Var y else t* = *Var z*)
by (*cases t*) *auto*

lemma *Eq_eq_subst_iff*: *y* $\approx$ *z* = *Q*[*x* $\rightarrow$ *y*] $\longleftrightarrow$ (*if z* = *x then x* = *y* $\wedge$ *Q* = *x* $\approx$ *x else*
Q = *x* $\approx$ *z* $\vee$ *Q* = *y* $\approx$ *z* $\vee$ (*z* = *y* $\wedge$ *Q* $\in$ {*x* $\approx$ *x*, *y* $\approx$ *y*, *y* $\approx$ *x*}))
by (*cases Q*) (*auto simp*: *Let_def Var_eq_subst_iff split*: *if_splits*)

lemma *eqs_subst*[*simp*]: *x* $\neq$ *y* $\Longrightarrow$ *eqs y* ((*λQ*. *Q*[*x* $\rightarrow$ *y*]) ‘ *G*) = (*eqs y G* − {*x*}) $\cup$ (*eqs x G* − {*y*})
by (*auto simp*: *eqs_def image_iff exists_def Eq_eq_subst_iff*)

lemma *gen_eqs*[*simp*]: *gen x Q G* $\Longrightarrow$ *eqs z G* = {}
by (*auto dest*: *gen_qp simp*: *eqs_def*)

lemma *eqs_insert*: *eqs x* (*insert Q Qs*) = (*case Q of z* $\approx$ *y* $\Rightarrow$
if z = *x* $\wedge$ *z* $\neq$ *y then insert y* (*eqs x Qs*) *else eqs x Qs* | _ $\Rightarrow$ *eqs x Qs*)
by (*auto simp*: *eqs_def split*: *fmla.splits term.splits*)

lemma *eqs_insert*′: *y* $\neq$ *x* $\Longrightarrow$ *eqs x* (*insert* (*x* $\approx$ *y*) *Qs*) = *insert y* (*eqs x Qs*)
by (*auto simp*: *eqs_def split*: *fmla.splits term.splits*)

lemma *eqs_code*[*code*]: *eqs x G* = (*λeq*. *case eq of y* $\approx$ *z* $\Rightarrow$ *z*) ‘ (*Set.filter* (*λeq*. *case eq of y* $\approx$ *z* $\Rightarrow$ *x* = *y* $\wedge$ *x* $\neq$ *z* | _ => *False*) *G*)
by (*auto simp*: *eqs_def image_iff Set.filter_def split*: *term.splits fmla.splits*)

lemma *gen_finite*[*simp*]: *gen x Q G* $\Longrightarrow$ *finite G*
by (*induct x Q G rule*: *gen_induct*) *auto*

lemma *cov_finite*[*simp*]: *cov x Q G* $\Longrightarrow$ *finite G*
by (*induct x Q G rule*: *cov.induct*) *auto*

lemma *gen_sat_erase*: *gen y Q Gy* $\Longrightarrow$ *sat* (*Q* $\perp$ *x*) *I* σ $\Longrightarrow$ $\exists$ *Q*$\in$*Gy*. *sat Q I* σ
by (*induct y Q Gy arbitrary*: σ *rule*: *gen_induct*)
(*force elim*!: *ap.cases dest*: *sym gen_sat split*: *if_splits*)+

lemma *cov_sat_erase*: *cov x Q G* $\Longrightarrow$
sat (*Neg* (*Disj* (*DISJ* (*qps G*)) (*DISJ* ((*λy*. *x* $\approx$ *y*) ' *eqs x G*)))) *I* σ $\Longrightarrow$
sat Q I σ $\longleftrightarrow$ *sat* (*cp* (*Q* $\perp$ *x*)) *I* σ
unfolding *sat_cp*
proof (*induct x Q G arbitrary*: σ *rule*: *cov_induct*)
case (*Eq_self x*)
then show *?case*
by *auto*
next
case (*nonfree x Q*)
from *nonfree*(*1*) **show** *?case*
by (*induct Q arbitrary*: σ) *auto*
next
case (*EqL x y*)
then show *?case*
by (*auto simp*: *eqs_def*)
next
case (*EqR x y*)
then show *?case*
by (*auto simp*: *eqs_def*)
next
case (*ap Q x*)
then show *?case*
by (*auto simp*: *qps_def qp.intros elim*!: *ap.cases*)
next
case (*Neg x Q G*)
then show *?case*
by *auto*
next
case (*Disj x Q1 G1 Q2 G2*)
then show *?case*
by *auto*
next
case (*DisjL x Q1 G Q2*)
then have *sat* (*Q1* $\perp$ *x*) *I* σ
by (*subst sat_cp*[*symmetric*]) *auto*
with *DisjL* **show** *?case*
by *auto*
next
case (*DisjR x Q2 G Q1*)
then have *sat* (*Q2* $\perp$ *x*) *I* σ
by (*subst sat_cp*[*symmetric*]) *auto*
with *DisjR* **show** *?case*
by *auto*
next
case (*Conj x Q1 G1 Q2 G2*)
then show *?case*
by *auto*

```
next
  case (ConjL x Q1 G Q2)
  then have ¬ sat (Q1 ⊥ x) I σ
    by (subst sat_cp[symmetric]) auto
  with ConjL show ?case
    by auto
next
  case (ConjR x Q2 G Q1)
  then have ¬ sat (Q2 ⊥ x) I σ
    by (subst sat_cp[symmetric]) auto
  with ConjR show ?case
    by auto
next
  case (Exists x y Q G)
  then show ?case
    by fastforce
next
  case (Exists_gen x y Q G Gy)
  show ?case
    unfolding sat.simps erase.simps Exists_gen(1)[THEN eq_False[THEN iffD2]] if_False
  proof (intro ex_cong1)
    fix z
    show sat Q I (σ(y := z)) = sat (Q ⊥ x) I (σ(y := z))
    proof (cases z = σ x)
      case True
      with Exists_gen(2,4,5) show ?thesis
        by (auto dest: gen_sat gen_sat_erase simp: ball_Un)
    next
      case False
      with Exists_gen(1,2,4,5) show ?thesis
        by (intro Exists_gen(3)) (auto simp: ball_Un fun_upd_def)
    qed
  qed
qed

lemma cov_fv_aux: cov x Q G ⟹ Qqp ∈ G ⟹ x ∈ fv Qqp ∧ fv Qqp − {x} ⊆ fv Q
  by (induct x Q G arbitrary: Qqp rule: cov_induct)
    (auto simp: fv_subst subset_eq gen_fv[THEN conjunct1]
      gen_fv[THEN conjunct2, THEN set_mp] dest: gen_fv split: if_splits)

lemma cov_fv: cov x Q G ⟹ x ∈ fv Q ⟹ Qqp ∈ G ⟹ x ∈ fv Qqp ∧ fv Qqp ⊆ fv Q
  using cov_fv_aux[of x Q G Qqp] by auto

lemma Gen_Conj:
  Gen x Q1 ⟹ Gen x (Conj Q1 Q2)
  Gen x Q2 ⟹ Gen x (Conj Q1 Q2)
  by (auto intro: gen.intros)

lemma cov_Gen_qps: cov x Q G ⟹ x ∈ fv Q ⟹ Gen x (Conj Q (DISJ (qps G)))
  by (intro Gen_Conj(2) Gen_DISJ) (auto simp: qps_def dest: cov_fv)

lemma cov_equiv:
  assumes cov x Q G ⋀Q I σ. sat (simp Q) I σ = sat Q I σ
  shows Q ≜ Disj (simp (Conj Q (DISJ (qps G))))
    (Disj (DISJ ((λy. Conj (cp (Q[x → y])) (x ≈ y)) ‘ eqs x G))
    (Conj (Q ⊥ x) (Neg (Disj (DISJ (qps G)) (DISJ ((λy. x ≈ y) ‘ eqs x G))))))
  (is _ ≜ ?rhs)
unfolding equiv_def proof (intro allI impI)
```

```
  fix I σ
  show sat Q I σ = sat ?rhs I σ
    using cov_sat_erase[OF assms(1), of I σ] assms
    by (fastforce dest: sym simp del: cp.simps)
qed

fun csts_term where
  csts_term (Var x) = {}
| csts_term (Const c) = {c}

fun csts where
  csts (Bool b) = {}
| csts (Pred p ts) = (⋃ t ∈ set ts. csts_term t)
| csts (Eq x t) = csts_term t
| csts (Neg Q) = csts Q
| csts (Conj Q1 Q2) = csts Q1 ∪ csts Q2
| csts (Disj Q1 Q2) = csts Q1 ∪ csts Q2
| csts (Exists x Q) = csts Q

lemma finite_csts_term[simp]: finite (csts_term t)
  by (induct t) auto

lemma finite_csts[simp]: finite (csts t)
  by (induct t) auto

lemma ap_fresh_val: ap Q ⟹ σ x ∉ adom I ⟹ σ x ∉ csts Q ⟹ sat Q I σ ⟹ x ∉ fv Q
proof (induct Q pred: ap)
  case (Pred p ts)
  show ?case unfolding fv.simps fv_terms_set_def set_map UN_iff bex_simps
  proof safe
    fix t
    assume t ∈ set ts x ∈ fv_term_set t
    with Pred show False
      by (cases t) (force simp: adom_def eval_terms_def)+
  qed
qed auto

lemma qp_fresh_val: qp Q ⟹ σ x ∉ adom I ⟹ σ x ∉ csts Q ⟹ sat Q I σ ⟹ x ∉ fv Q
proof (induct Q arbitrary: σ rule: qp.induct)
  case (ap Q)
  then show ?case by (rule ap_fresh_val)
next
  case (exists Q z)
  from exists(2)[of σ] exists(2)[of σ(z := _)] exists(1,3−) show ?case
    by (cases x = z) (auto simp: exists_def fun_upd_def split: if_splits)
qed

lemma ex_fresh_val:
  fixes Q :: ('a :: infinite, 'b) fmla
  assumes finite (adom I) finite A
  shows ∃ x. x ∉ adom I ∧ x ∉ csts Q ∧ x ∉ A
  by (metis UnCI assms ex_new_if_finite finite_Un finite_csts infinite_UNIV)

definition fresh_val :: ('a :: infinite, 'b) fmla ⇒ ('a, 'b) intp ⇒ 'a set ⇒ 'a where
  fresh_val Q I A = (SOME x. x ∉ adom I ∧ x ∉ csts Q ∧ x ∉ A)

lemma fresh_val:
  finite (adom I) ⟹ finite A ⟹ fresh_val Q I A ∉ adom I
```

finite (*adom I*) ⟹ *finite A* ⟹ *fresh_val Q I A* ∉ *csts Q*
finite (*adom I*) ⟹ *finite A* ⟹ *fresh_val Q I A* ∉ *A*
using *someI_ex*[*OF ex_fresh_val*, *of I A Q*]
by (*auto simp*: *fresh_val_def*)

lemma *csts_exists*[*simp*]: *csts* (*exists x Q*) = *csts Q*
by (*auto simp*: *exists_def*)

lemma *csts_term_subst_term*[*simp*]: *csts_term* (*t*[*x* →*t y*]) = *csts_term t*
by (*cases t*) *auto*

lemma *csts_subst*[*simp*]: *csts* (*Q*[*x* → *y*]) = *csts Q*
by (*induct Q x y rule*: *subst.induct*) (*auto simp*: *Let_def*)

lemma *gen_csts*: *gen x Q G* ⟹ *Qqp* ∈ *G* ⟹ *csts Qqp* ⊆ *csts Q*
by (*induct x Q G arbitrary*: *Qqp rule*: *gen_induct*) (*auto simp*: *subset_eq*)

lemma *cov_csts*: *cov x Q G* ⟹ *Qqp* ∈ *G* ⟹ *csts Qqp* ⊆ *csts Q*
by (*induct x Q G arbitrary*: *Qqp rule*: *cov_induct*)
(*auto simp*: *subset_eq gen_csts*[*THEN set_mp*])

lemma *not_self_eqs*[*simp*]: *x* ∉ *eqs x G*
by (*auto simp*: *eqs_def*)

lemma (**in** *simplification*) *cov_Exists_equiv*:
fixes *Q* :: ('*a* :: {*infinite*, *linorder*}, '*b* :: *linorder*) *fmla*
assumes *cov x Q G x* ∈ *fv Q*
shows *Exists x Q* ≜ *Disj* (*Exists x* (*simp* (*Conj Q* (*DISJ* (*qps G*)))))
(*Disj* (*DISJ* ((λ*y*. *cp* (*Q*[*x* → *y*])) ‘ *eqs x G*)) (*cp* (*Q* ⊥ *x*)))
proof −
have *Exists x Q* ≜ *Exists x* (*Disj* (*simp* (*Conj Q* (*DISJ* (*qps G*))))
(*Disj* (*DISJ* ((λ*y*. *Conj* (*cp* (*Q*[*x* → *y*])) (*x* ≈ *y*)) ‘ *eqs x G*))
(*Conj* (*Q* ⊥ *x*) (*Neg* (*Disj* (*DISJ* (*qps G*)) (*DISJ* ((≈) *x* ‘ *eqs x G*)))))))
by (*rule equiv_Exists_cong*[*OF cov_equiv*[*OF assms*(*1*) *sat_simp*]])
also have ... ≜ *Disj* (*Exists x* (*simp* (*Conj Q* (*DISJ* (*qps G*))))) (*Disj*
(*Exists x* (*DISJ* ((λ*y*. *Conj* (*cp* (*Q*[*x* → *y*])) (*x* ≈ *y*)) ‘ *eqs x G*)))
(*Exists x* (*Conj* (*Q* ⊥ *x*) (*Neg* (*Disj* (*DISJ* (*qps G*)) (*DISJ* ((≈) *x* ‘ *eqs x G*)))))))
by (*auto intro*!: *equiv_trans*[*OF equiv_Exists_Disj*] *equiv_Disj_cong*[*OF equiv_refl*])
also have ... ≜ *Disj* (*Exists x* (*simp* (*Conj Q* (*DISJ* (*qps G*))))) (*Disj*
(*Exists x* (*DISJ* ((λ*y*. *cp* (*Q*[*x* → *y*])) ‘ *eqs x G*)) (*cp* (*Q* ⊥ *x*)))
proof (*rule equiv_Disj_cong*[*OF equiv_refl equiv_Disj_cong*])
show *Exists x* (*DISJ* ((λ*y*. *Conj* (*cp* (*Q*[*x* → *y*])) (*x* ≈ *y*)) ‘ *eqs x G*)) ≜ *DISJ* ((λ*y*. *cp* (*Q*[*x* → *y*]))
‘ *eqs x G*)
using *assms*(*1*) **unfolding** *equiv_def*
by *simp* (*auto simp*: *eqs_def*)
next
show *Exists x* (*Conj* (*Q* ⊥ *x*) (*Neg* (*Disj* (*DISJ* (*qps G*)) (*DISJ* ((≈) *x* ‘ *eqs x G*))))) ≜ *cp* (*Q* ⊥ *x*)
unfolding *equiv_def sat.simps sat_erase sat_cp*
sat_DISJ[*OF finite_qps*[*OF cov_finite*[*OF assms*(*1*)]]]
sat_DISJ[*OF finite_imageI*[*OF finite_eqs*[*OF cov_finite*[*OF assms*(*1*)]]]]
proof (*intro allI impI*)
fix *I* :: ('*a*, '*b*) *intp* **and** *σ*
assume *finite* (*adom I*)
then show (∃ *z*. *sat* (*Q* ⊥ *x*) *I σ* ∧ ¬ ((∃ *Q*∈*qps G*. *sat Q I* (*σ*(*x* := *z*))) ∨ (∃ *Q*∈(≈) *x* ‘ *eqs x G*.
sat Q I (*σ*(*x* := *z*))))) =
sat (*Q* ⊥ *x*) *I σ*
using *fresh_val*[*OF _ finite_imageI*[*OF finite_fv*], *of I Q σ Q*] *assms*
by (*auto 0 3 simp*: *qps_def eqs_def intro*!: *exI*[*of _ fresh_val Q I* (*σ* ‘ *fv Q*)]

```
      dest: cov_fv cov_csts[THEN set_mp]
      qp_fresh_val[where σ=σ(x := fresh_val Q I (σ ‘ fv Q)) and x=x and I=I])
  qed
 qed
 finally show ?thesis .
qed

definition eval_on V Q I =
 (let xs = sorted_list_of_set V
 in {ds. length xs = length ds ∧ (∃σ. sat Q I (σ[xs :=* ds]))})

definition eval Q I = eval_on (fv Q) Q I

lemmas eval_deep_def = eval_def[unfolded eval_on_def]

lemma (in simplification) cov_eval_fin:
 fixes Q :: ('a :: {infinite, linorder}, 'b :: linorder) fmla
 assumes cov x Q G x ∈ fv Q finite (adom I) ⋀σ. ¬ sat (Q ⊥ x) I σ
 shows eval Q I = eval_on (fv Q) (Disj (simp (Conj Q (DISJ (qps G)))))
  (DISJ ((λy. Conj (cp (Q[x → y])) (x ≈ y)) ‘ eqs x G))) I
  (is eval Q I = eval_on (fv Q) ?Q I)
proof −
 from assms(1) have fv: fv ?Q ⊆ fv Q
  by (auto dest!: fv_cp[THEN set_mp] fv_simp[THEN set_mp] fv_DISJ[THEN set_mp, rotated −1]
    eqs_in qps_in cov_fv[OF assms(1,2)] simp: fv_subst simp del: cp.simps split: if_splits)
 show ?thesis
  unfolding eval_deep_def eval_on_def Let_def fv
 proof (intro Collect_eqI arg_cong2[of _ _ _ _ (∧)] ex_cong1)
  fix ds σ
  show sat Q I (σ[sorted_list_of_set (fv Q) :=* ds]) ⟷
      sat ?Q I (σ[sorted_list_of_set (fv Q) :=* ds])
   by (subst cov_equiv[OF assms(1) sat_simp, unfolded equiv_def, rule_format, OF assms(3)])
    (auto simp: assms(4))
 qed simp
qed

lemma (in simplification) cov_sat_fin:
 fixes Q :: ('a :: {infinite, linorder}, 'b :: linorder) fmla
 assumes cov x Q G x ∈ fv Q finite (adom I) ⋀σ. ¬ sat (Q ⊥ x) I σ
 shows sat Q I σ = sat (Disj (simp (Conj Q (DISJ (qps G)))))
  (DISJ ((λy. Conj (cp (Q[x → y])) (x ≈ y)) ‘ eqs x G))) I σ
  (is sat Q I σ = sat ?Q I σ)
proof −
 from assms(1) have fv: fv ?Q ⊆ fv Q
  by (auto dest!: fv_cp[THEN set_mp] fv_simp[THEN set_mp] fv_DISJ[THEN set_mp, rotated −1]
    eqs_in qps_in cov_fv[OF assms(1,2)] simp: fv_subst simp del: cp.simps split: if_splits)
 show ?thesis
  by (subst cov_equiv[OF assms(1) sat_simp, unfolded equiv_def, rule_format, OF assms(3)])
   (auto simp: assms(4))
qed

lemma equiv_eval_eqI: finite (adom I) ⟹ fv Q = fv Q' ⟹ Q ≜ Q' ⟹ eval Q I = eval Q' I
 by (auto simp: eval_deep_def equiv_def)

lemma equiv_eval_on_eqI: finite (adom I) ⟹ Q ≜ Q' ⟹ eval_on X Q I = eval_on X Q' I
 by (auto simp: eval_on_def equiv_def)

lemma equiv_eval_on_eval_eqI: finite (adom I) ⟹ fv Q ⊆ fv Q' ⟹ Q ≜ Q' ⟹ eval_on (fv Q') Q
```

```
I = eval Q' I
  by (auto simp: eval_deep_def eval_on_def equiv_def)

lemma finite_eval_on_Disj2D:
  assumes finite X
  shows finite (eval_on X (Disj Q1 Q2) I) ⟹ finite (eval_on X Q2 I)
  unfolding eval_on_def Let_def
  by (auto elim!: finite_subset[rotated])

lemma finite_eval_Disj2D: finite (eval (Disj Q1 Q2) I) ⟹ finite (eval Q2 I)
  unfolding eval_deep_def Let_def
proof (safe elim!: finite_surj)
  fix ds σ
  assume length (sorted_list_of_set (fv Q2)) = length ds sat Q2 I (σ[sorted_list_of_set (fv Q2) :=*
ds])
  moreover obtain zs where zs ∈ extend (fv Q2) (sorted_list_of_set (fv Q1 ∪ fv Q2)) ds
    using extend_nonempty by blast
  ultimately show ds ∈ restrict (fv Q2) (sorted_list_of_set (fv (Disj Q1 Q2))) ‘
    {ds. length (sorted_list_of_set (fv (Disj Q1 Q2))) = length ds ∧
        (∃σ. sat (Disj Q1 Q2) I (σ[sorted_list_of_set (fv (Disj Q1 Q2)) :=* ds]))}
    by (auto simp: Let_def image_iff restrict_extend fun_upds_extend length_extend
      elim!: sat_fv_cong[THEN iffD2, rotated −1]
      intro!: exI[of _ zs] exI[of _ σ] disjI2)
qed

lemma infinite_eval_Disj2:
  fixes Q1 Q2 :: ('a :: {infinite, linorder}, 'b :: linorder) fmla
  assumes fv Q2 ⊂ fv (Disj Q1 Q2) sat Q2 I σ
  shows infinite (eval (Disj Q1 Q2) I)
proof −
  from assms(1) obtain z where z ∈ fv Q1 z ∉ fv Q2
    by auto
  then have d ∈ (λds. lookup (sorted_list_of_set (fv Q1 ∪ fv Q2)) ds z) ‘ eval (Disj Q1 Q2) I for d
    using assms(2)
    by (auto simp: fun_upds_map_self eval_deep_def Let_def length_extend intro!: exI[of _ σ] disjI2
imageI
      dest!: ex_lookup_extend[of _ _ (sorted_list_of_set (fv Q1 ∪ fv Q2)) map σ (sorted_list_of_set
(fv Q2)) d]
      elim!: sat_fv_cong[THEN iffD2, rotated −1] fun_upds_extend[THEN trans])
  then show ?thesis
    by (rule infinite_surj[OF infinite_UNIV, OF subsetI])
qed

lemma infinite_eval_on_Disj2:
  fixes Q1 Q2 :: ('a :: {infinite, linorder}, 'b :: linorder) fmla
  assumes fv Q2 ⊂ X fv Q1 ⊆ Xfinite X sat Q2 I σ
  shows infinite (eval_on X (Disj Q1 Q2) I)
proof −
  from assms(1) obtain z where z ∈ X z ∉ fv Q2
    by auto
  then have d ∈ (λds. lookup (sorted_list_of_set X) ds z) ‘ eval_on X (Disj Q1 Q2) I for d
    using assms ex_lookup_extend[of z fv Q2 (sorted_list_of_set X) map σ (sorted_list_of_set (fv Q2))
d]
    by (auto simp: fun_upds_map_self eval_on_def Let_def subset_eq length_extend intro!: exI[of _ σ]
disjI2 imageI
      elim!: sat_fv_cong[THEN iffD2, rotated −1] fun_upds_extend[rotated −1, THEN trans])
  then show ?thesis
    by (rule infinite_surj[OF infinite_UNIV, OF subsetI])
```

```
qed

lemma cov_eval_inf:
  fixes Q :: ('a :: {infinite, linorder}, 'b :: linorder) fmla
  assumes cov x Q G x ∈ fv Q finite (adom I) sat (Q ⊥ x) I σ
  shows infinite (eval Q I)
proof −
  let ?Q1 = Conj Q (DISJ (qps G))
  let ?Q2 = DISJ ((λy. Conj (cp (Q[x → y])) (x ≈ y)) ‘ eqs x G)
  define Q3 where Q3 = Conj (Q ⊥ x) (Neg (Disj (DISJ (qps G)) (DISJ ((λy. x ≈ y) ‘ eqs x G))))
  let ?Q = Disj ?Q1 (Disj ?Q2 Q3)
  from assms(1) have fv123: fv ?Q1 ⊆ fv Q fv ?Q2 ⊆ fv Q fv Q3 ⊆ fv Q and fin_fv[simp]: finite (fv
Q3) unfolding Q3_def
    by (auto dest!: fv_cp[THEN set_mp] fv_DISJ[THEN set_mp, rotated 1] fv_erase[THEN set_mp]
      eqs_in qps_in cov_fv[OF assms(1,2)] simp: fv_subst simp del: cp.simps)
  then have fv: fv ?Q ⊆ fv Q
    by auto
  from assms(1,2,4) have sat: sat Q3 I (σ(x := d)) if d ∉ adom I ∪ csts Q ∪ σ ‘ fv Q for d
    using that cov_fv[OF assms(1,2) qps_in] cov_fv[OF assms(1,2) eqs_in, of _ x]
      qp_fresh_val[OF qps_qp, of _ G σ(x := d) x I] cov_csts[OF assms(1) qps_in]
    by (auto 5 2 simp: image_iff Q3_def elim!: sat_fv_cong[THEN iffD2, rotated −1]
      dest: fv_erase[THEN set_mp] dest: eqs_in)
  from assms(3) have inf: infinite {d. d ∉ adom I ∪ csts Q ∪ σ ‘ fv Q}
    unfolding Compl_eq[symmetric] Compl_eq_Diff_UNIV
    by (intro Diff_infinite_finite) (auto simp: infinite_UNIV)
  { assume x ∈ fv Q3
    let ?f = λds. lookup (sorted_list_of_set (fv Q)) ds x
    from inf have infinite (eval_on (fv Q) Q3 I)
    proof (rule infinite_surj[where f=?f], intro subsetI, elim CollectE)
      fix z
      assume z ∉ adom I ∪ csts Q ∪ σ ‘ fv Q
      with ‹x ∈ fv Q3› fv123 sat show z ∈ ?f ‘ eval_on (fv Q) Q3 I
        by (auto simp: eval_on_def image_iff Let_def fun_upds_single subset_eq simp del: cp.simps
          intro!: exI[of _ σ] exI[of _ map (σ(x := z)) (sorted_list_of_set (fv Q))])
    qed
    then have infinite (eval_on (fv Q) ?Q I)
      by (rule contrapos_nn) (auto dest!: finite_eval_on_Disj2D[rotated])
  }
  moreover
  { assume x: x ∉ fv Q3
    from inf obtain d where d ∉ adom I ∪ csts Q ∪ σ ‘ fv Q
      by (meson not_finite_existsD)
    with fv123 sat[of d] assms(2) x have infinite (eval_on (fv Q) (Disj (Disj ?Q1 ?Q2) Q3) I)
      by (intro infinite_eval_on_Disj2[of _fv Q _ _ (σ(x := d))]) (auto simp del: cp.simps)
    moreover have eval_on (fv Q) (Disj (Disj ?Q1 ?Q2) Q3) I = eval_on (fv Q) ?Q I
      by (rule equiv_eval_on_eqI[OF assms(3) equiv_Disj_Assoc])
    ultimately have infinite (eval_on (fv Q) ?Q I)
      by simp
  }
  moreover have eval Q I = eval_on (fv Q) ?Q I
    unfolding Q3_def
   by (rule equiv_eval_on_eval_eqI[symmetric, OF assms(3) fv[unfolded Q3_def] cov_equiv[OF assms(1)
refl, THEN equiv_sym]])
  ultimately show ?thesis
    by auto
qed
```

2.12 More on Evaluation

lemma *eval_Bool_False*[*simp*]: *eval* (*Bool False*) *I* = {}
by (*auto simp*: *eval_deep_def*)

lemma *eval_on_False*[*simp*]: *eval_on X* (*Bool False*) *I* = {}
by (*auto simp*: *eval_on_def*)

lemma *eval_DISJ_prune_unsat*: *finite B* ⟹ *A* ⊆ *B* ⟹ ∀ *Q* ∈ *B* − *A*. ∀ σ. ¬ *sat Q I* σ ⟹ *eval_on X* (*DISJ A*) *I* = *eval_on X* (*DISJ B*) *I*
by (*auto simp*: *eval_on_def finite_subset*)

lemma *eval_DISJ*: *finite* 𝒬 ⟹ ∀ *Q* ∈ 𝒬. *fv Q* = *A* ⟹ *eval_on A* (*DISJ* 𝒬) *I* = (⋃ *Q* ∈ 𝒬. *eval Q I*)
by (*auto simp*: *eval_deep_def eval_on_def*)

lemma *eval_cp_DISJ_closed*: *finite* 𝒬 ⟹ ∀ *Q* ∈ 𝒬. *fv Q* = {} ⟹ *eval* (*cp* (*DISJ* 𝒬)) *I* = (⋃ *Q* ∈ 𝒬. *eval Q I*)
using *fv_DISJ*[*of* 𝒬] *fv_cp*[*of DISJ* 𝒬] **by** (*auto simp*: *eval_deep_def*)

lemma (**in** *simplification*) *eval_simp_DISJ_closed*: *finite* 𝒬 ⟹ ∀ *Q* ∈ 𝒬. *fv Q* = {} ⟹ *eval* (*simp* (*DISJ* 𝒬)) *I* = (⋃ *Q* ∈ 𝒬. *eval Q I*)
using *fv_DISJ*[*of* 𝒬] *fv_simp*[*of DISJ* 𝒬] **by** (*auto simp*: *eval_deep_def sat_simp*)

lemma *eval_cong*: *fv Q* = *fv Q'* ⟹ (⋀σ. *sat Q I* σ = *sat Q' I* σ) ⟹ *eval Q I* = *eval Q' I*
by (*auto simp*: *eval_deep_def*)

lemma *eval_on_cong*: (⋀σ. *sat Q I* σ = *sat Q' I* σ) ⟹ *eval_on X Q I* = *eval_on X Q' I*
by (*auto simp*: *eval_on_def*)

lemma *eval_empty_alt*: *eval Q I* = {} ⟷ (∀ σ. ¬ *sat Q I* σ)
proof (*intro iffI allI*)
fix σ
assume *eval Q I* = {}
then show ¬ *sat Q I* σ
by (*auto simp*: *eval_deep_def fun_upds_map_self*
dest!: *spec*[*of* _ *map* σ (*sorted_list_of_set* (*fv Q*))] *spec*[*of* _ σ])
qed (*auto simp*: *eval_deep_def*)

lemma *sat_EXISTS*: *distinct xs* ⟹ *sat* (*EXISTS xs Q*) *I* σ = (∃ *ds*. *length ds* = *length xs* ∧ *sat Q I* (σ[*xs* :=* *ds*]))
proof (*induct xs arbitrary*: *Q* σ)
case (*Cons x xs*)
then show *?case*
by (*auto 0 3 simp*: *EXISTS_def length_Suc_conv fun_upds_twist fun_upd_def*[*symmetric*])
qed (*simp add*: *EXISTS_def*)

lemma *eval_empty_close*: *eval* (*close Q*) *I* = {} ⟷ (∀ σ. ¬ *sat Q I* σ)
by (*subst eval_empty_alt*)
(*auto simp*: *sat_EXISTS fun_upds_map_self dest*: *spec2*[*of* _ σ *map* σ(*sorted_list_of_set* (*fv Q*))
for σ])

lemma *infinite_eval_on_extra_variables*:
assumes *finite X fv* (*Q* :: ('*a* :: *infinite*, '*b*) *fmla*) ⊂ *X* ∃ σ. *sat Q I* σ
shows *infinite* (*eval_on X Q I*)
proof −
from *assms* **obtain** *x* σ **where** *x* ∈ *X* − *fv Q fv Q* ⊆ *X sat Q I* σ
by *auto*
with *assms*(*1*) **show** *?thesis*
by (*intro infinite_surj*[*OF infinite_UNIV*, *of* λ*ds*. *ds* ! *index* (*sorted_list_of_set X*) *x*])

```
    (force simp: eval_on_def image_iff fun_upds_in
      elim!: sat_fv_cong[THEN iffD1, rotated]
      intro!: exI[of _ map (λy. if x = y then _ else σ y) (sorted_list_of_set X)] exI[of _ σ])
qed

lemma eval_on_cp: eval_on X (cp Q) = eval_on X Q
  by (auto simp: eval_on_def)

lemma (in simplification) eval_on_simp: eval_on X (simp Q) = eval_on X Q
  by (auto simp: eval_on_def sat_simp)

lemma (in simplification) eval_simp_False: eval (simp (Bool False)) I = {}
  using fv_simp[of Bool False] by (auto simp: eval_deep_def sat_simp)

abbreviation idx_of_var x Q ≡ index (sorted_list_of_set (fv Q)) x

lemma evalE: ds ∈ eval Q I ⟹ (⋀σ. length ds = card (fv Q) ⟹ sat Q I (σ[sorted_list_of_set (fv Q)
:=* ds]) ⟹ R) ⟹ R
  unfolding eval_deep_def by auto

lemma infinite_eval_Conj:
  assumes x ∉ fv Q infinite (eval Q I)
  shows infinite (eval (Conj Q (x ≈ y)) I)
    (is infinite (eval ?Qxy I))
proof (cases x = y)
  case True
  let ?f = remove_nth (idx_of_var x ?Qxy)
  let ?g = insert_nth (idx_of_var x ?Qxy) undefined
  show ?thesis
    using assms(2)
  proof (elim infinite_surj[of _ ?f], intro subsetI, elim evalE)
    fix ds σ
    assume ds: length ds = card (fv Q) sat Q I (σ[sorted_list_of_set (fv Q) :=* ds])
    show ds ∈ ?f ` eval ?Qxy I
    proof (intro image_eqI[of _ _ ?g ds])
      from ds assms(1) True show ds = ?f (?g ds)
        by (intro remove_nth_insert_nth[symmetric])
          (auto simp: less_Suc_eq_le[symmetric] set_insort_key)
    next
      from ds assms(1) True show ?g ds ∈ eval ?Qxy I
      by (auto simp: eval_deep_def Let_def length_insert_nth distinct_insort set_insort_key fun_upds_in
          simp del: insert_nth_take_drop elim!: sat_fv_cong[THEN iffD1, rotated]
          intro!: exI[of _ σ] trans[OF _ insert_nth_nth_index[symmetric]])
    qed
  qed
next
  case xy: False
  show ?thesis
  proof (cases y ∈ fv Q)
    case True
    let ?f = remove_nth (idx_of_var x ?Qxy)
    let ?g = λds. insert_nth (idx_of_var x ?Qxy) (ds ! idx_of_var y Q) ds
    from assms(2) show ?thesis
    proof (elim infinite_surj[of _ ?f], intro subsetI, elim evalE)
      fix ds σ
      assume ds: length ds = card (fv Q) sat Q I (σ[sorted_list_of_set (fv Q) :=* ds])
      show ds ∈ ?f ` eval ?Qxy I
      proof (intro image_eqI[of _ _ ?g ds])
```

```
      from ds assms(1) True show ds = ?f (?g ds)
        by (intro remove_nth_insert_nth[symmetric])
          (auto simp: less_Suc_eq_le[symmetric] set_insort_key)
    next
      from assms(1) True have remove1 x (insort y (sorted_list_of_set (insert x (fv Q) − {y}))) =
sorted_list_of_set (fv Q)
        by (metis Diff_insert_absorb finite_fv finite_insert insert_iff
         sorted_list_of_set.fold_insort_key.remove sorted_list_of_set.sorted_key_list_of_set_remove)
      moreover have index (insort y (sorted_list_of_set (insert x (fv Q) − {y}))) x ≤ length ds
        using ds(1) assms(1) True
        by (subst less_Suc_eq_le[symmetric]) (auto simp: set_insort_key intro: index_less_size)
      ultimately show ?g ds ∈ eval ?Qxy I
        using ds assms(1) True
          by (auto simp: eval_deep_def Let_def length_insert_nth distinct_insort set_insort_key
fun_upds_in nth_insert_nth
          simp del: insert_nth_take_drop elim!: sat_fv_cong[THEN iffD1, rotated]
          intro!: exI[of _ σ] trans[OF _ insert_nth_nth_index[symmetric]])
    qed
  qed
 next
  case False
  let ?Qxx = Conj Q (x ≈ x)
  let ?f = remove_nth (idx_of_var x ?Qxx) o remove_nth (idx_of_var y ?Qxy)
  let ?g1 = insert_nth (idx_of_var y ?Qxy) undefined
  let ?g2 = insert_nth (idx_of_var x ?Qxx) undefined
  let ?g = ?g1 o ?g2
  from assms(2) show ?thesis
  proof (elim infinite_surj[of _ ?f], intro subsetI, elim evalE)
    fix ds σ
    assume ds: length ds = card (fv Q) sat Q I (σ[sorted_list_of_set (fv Q) :=* ds])
    then show ds ∈ ?f ‘ eval ?Qxy I
    proof (intro image_eqI[of _ _ ?g ds])
      from ds assms(1) xy False show ds = ?f (?g ds)
        by (auto simp: less_Suc_eq_le[symmetric] set_insort_key index_less_size
          length_insert_nth remove_nth_insert_nth simp del: insert_nth_take_drop)
    next
      from ds(1) have index (insort x (sorted_list_of_set (fv Q))) x ≤ length ds
        by (auto simp: less_Suc_eq_le[symmetric] set_insort_key)
     moreover from ds(1) have index (insort y (insort x (sorted_list_of_set (fv Q)))) y ≤ Suc (length
ds)
        by (auto simp: less_Suc_eq_le[symmetric] set_insort_key)
      ultimately show ?g ds ∈ eval ?Qxy I
        using ds assms(1) xy False unfolding eval_deep_def Let_def
        by (auto simp: fun_upds_in distinct_insort set_insort_key length_insert_nth
          insert_nth_nth_index nth_insert_nth elim!: sat_fv_cong[THEN iffD1, rotated]
       intro!: exI[of _ σ] trans[OF _ insert_nth_nth_index[symmetric]] simp del: insert_nth_take_drop)
[]
    qed
  qed
 qed
  qed

lemma infinite_Implies_mono_on: infinite (eval_on X Q I) ⟹ finite X ⟹ (⋀σ. sat (Impl Q Q′) I
σ) ⟹ infinite (eval_on X Q′ I)
 by (erule contrapos_nn, rule finite_subset[rotated]) (auto simp: eval_on_def image_iff)
```

3 Restricting Bound Variables

fun *flat_Disj* **where**
  flat_Disj (*Disj Q1 Q2*) = *flat_Disj Q1* ∪ *flat_Disj Q2*
| *flat_Disj Q* = {*Q*}

lemma *finite_flat_Disj*[*simp*]: *finite* (*flat_Disj Q*)
  by (*induct Q rule*: *flat_Disj.induct*) *auto*

lemma *DISJ_flat_Disj*: *DISJ* (*flat_Disj Q*) ≜ *Q*
  by (*induct Q rule*: *flat_Disj.induct*) (*auto simp*: *DISJ_union*[*THEN equiv_trans*] *simp del*: *cp.simps*)

lemma *fv_flat_Disj*: (⋃ *Q'* ∈ *flat_Disj Q*. *fv Q'*) = *fv Q*
  by (*induct Q rule*: *flat_Disj.induct*) *auto*

lemma *fv_flat_DisjD*: *Q'* ∈ *flat_Disj Q* ⟹ *x* ∈ *fv Q'* ⟹ *x* ∈ *fv Q*
  by (*auto simp*: *fv_flat_Disj*[*of Q*, *symmetric*])

lemma *cpropagated_flat_DisjD*: *Q'* ∈ *flat_Disj Q* ⟹ *cpropagated Q* ⟹ *cpropagated Q'*
  by (*induct Q rule*: *flat_Disj.induct*) *auto*

lemma *flat_Disj_sub*: *flat_Disj Q* ⊆ *sub Q*
  by (*induct Q*) *auto*

lemma (**in** *simplification*) *simplified_flat_DisjD*: *Q'* ∈ *flat_Disj Q* ⟹ *simplified Q* ⟹ *simplified Q'*
  by (*elim simplified_sub set_mp*[*OF flat_Disj_sub*])

definition *fixbound* **where**
  fixbound 𝒬 *x* = {*Q* ∈ 𝒬. *x* ∈ *nongens Q*}

definition (**in** *simplification*) *rb_spec* **where**
  rb_spec Q = *SPEC* (λ*Q'*. *rrb Q'* ∧ *simplified Q'* ∧ *Q* ≜ *Q'* ∧ *fv Q'* ⊆ *fv Q*)

definition (**in** *simplification*) *rb_INV* **where**
  rb_INV x Q 𝒬 = (*finite* 𝒬 ∧
    Exists x Q ≜ *DISJ* (*exists x* ‘ 𝒬) ∧
    (∀ *Q'* ∈ 𝒬. *rrb Q'* ∧ *fv Q'* ⊆ *fv Q* ∧ *simplified Q'*))

lemma (**in** *simplification*) *rb_INV_I*:
  finite 𝒬 ⟹ *Exists x Q* ≜ *DISJ* (*exists x* ‘ 𝒬) ⟹ (⋀*Q'*. *Q'* ∈ 𝒬 ⟹ *rrb Q'*) ⟹
  (⋀*Q'*. *Q'* ∈ 𝒬 ⟹ *fv Q'* ⊆ *fv Q*) ⟹ (⋀*Q'*. *Q'* ∈ 𝒬 ⟹ *simplified Q'*) ⟹ *rb_INV x Q* 𝒬
  unfolding *rb_INV_def* **by** *auto*

fun (**in** *simplification*) *rb* :: ('*a* :: {*infinite*, *linorder*}, '*b* :: *linorder*) *fmla* ⇒ ('*a*, '*b*) *fmla nres* **where**
  rb (*Neg Q*) = *do* { *Q'* ← *rb Q*; *RETURN* (*simp* (*Neg Q'*))}
| *rb* (*Disj Q1 Q2*) = *do* { *Q1'* ← *rb Q1*; *Q2'* ← *rb Q2*; *RETURN* (*simp* (*Disj Q1' Q2'*))}
| *rb* (*Conj Q1 Q2*) = *do* { *Q1'* ← *rb Q1*; *Q2'* ← *rb Q2*; *RETURN* (*simp* (*Conj Q1' Q2'*))}
| *rb* (*Exists x Q*) = *do* {
    Q' ← *rb Q*;
    𝒬 ← *WHILE*$_T$ *rb_INV x Q'*
      (λ𝒬. *fixbound* 𝒬 *x* ≠ {}) (λ𝒬. *do* {
        Qfix ← *RES* (*fixbound* 𝒬 *x*);
        G ← *SPEC* (*cov x Qfix*);
        RETURN (𝒬 − {*Qfix*} ∪
          {*simp* (*Conj Qfix* (*DISJ* (*qps G*)))} ∪
          (⋃ *y* ∈ *eqs x G*. {*cp* (*Qfix*[*x* → *y*])}) ∪
          {*cp* (*Qfix* ⊥ *x*)})})
      (*flat_Disj Q'*);

```
    RETURN (simp (DISJ (exists x ‘ 𝒬)))}
| rb 𝒬 = do { RETURN (simp 𝒬) }

lemma (in simplification) cov_fixbound: cov x Q G ⟹ x ∈ fv Q ⟹
  fixbound (insert (cp (Q ⊥ x)) (insert (simp (Conj Q (DISJ (qps G))))
  (𝒬 − {Q} ∪ ((λy. cp (Q[x → y])) ‘ eqs x G)))) x = fixbound 𝒬 x − {Q}
  using Gen_simp[OF cov_Gen_qps[of x Q G]]
  by (auto 4 4 simp: fixbound_def nongens_def fv_subst split: if_splits
      dest!: fv_cp[THEN set_mp] fv_simp[THEN set_mp] fv_erase[THEN set_mp] dest: arg_cong[of _
_ fv] simp del: cp.simps)

lemma finite_fixbound[simp]: finite 𝒬 ⟹ finite (fixbound 𝒬 x)
  unfolding fixbound_def by auto

lemma fixboundE[elim_format]: Q ∈ fixbound 𝒬 x ⟹ x ∈ fv Q ∧ Q ∈ 𝒬 ∧ ¬ Gen x Q
  unfolding fixbound_def nongens_def by auto

lemma fixbound_fv: Q ∈ fixbound 𝒬 x ⟹ x ∈ fv Q
  unfolding fixbound_def nongens_def by auto

lemma fixbound_in: Q ∈ fixbound 𝒬 x ⟹ Q ∈ 𝒬
  unfolding fixbound_def nongens_def by auto

lemma fixbound_empty_Gen: fixbound 𝒬 x = {} ⟹ x ∈ fv Q ⟹ Q ∈ 𝒬 ⟹ Gen x Q
  unfolding fixbound_def nongens_def by auto

lemma fixbound_insert:
  fixbound (insert Q 𝒬) x = (if Gen x Q ∨ x ∉ fv Q then fixbound 𝒬 x else insert Q (fixbound 𝒬 x))
  by (auto simp: fixbound_def nongens_def)

lemma fixbound_empty[simp]:
  fixbound {} x = {}
  by (auto simp: fixbound_def)

lemma flat_Disj_Exists_sub: Q' ∈ flat_Disj Q ⟹ Exists y Qy ∈ sub Q' ⟹ Exists y Qy ∈ sub Q
  by (induct Q arbitrary: Q' rule: flat_Disj.induct) auto

lemma rrb_flat_Disj[simp]: Q ∈ flat_Disj Q' ⟹ rrb Q' ⟹ rrb Q
  by (induct Q' rule: flat_Disj.induct) auto

lemma (in simplification) rb_INV_finite[simp]: rb_INV x Q 𝒬 ⟹ finite 𝒬
  by (auto simp: rb_INV_def)

lemma (in simplification) rb_INV_fv: rb_INV x Q 𝒬 ⟹ Q' ∈ 𝒬 ⟹ z ∈ fv Q' ⟹ z ∈ fv Q
  by (auto simp: rb_INV_def)

lemma (in simplification) rb_INV_rrb: rb_INV x Q 𝒬 ⟹ Q' ∈ 𝒬 ⟹ rrb Q'
  by (auto simp: rb_INV_def)

lemma (in simplification) rb_INV_cpropagated: rb_INV x Q 𝒬 ⟹ Q' ∈ 𝒬 ⟹ simplified Q'
  by (auto simp: rb_INV_def)

lemma (in simplification) rb_INV_equiv: rb_INV x Q 𝒬 ⟹ Exists x Q ≙ DISJ (exists x ‘ 𝒬)
  by (auto simp: rb_INV_def)

lemma (in simplification) rb_INV_init[simp]: simplified Q ⟹ rrb Q ⟹ rb_INV x Q (flat_Disj Q)
  by (auto simp: rb_INV_def fv_flat_DisjD simplified_flat_DisjD
    equiv_trans[OF equiv_Exists_cong[OF DISJ_flat_Disj[THEN equiv_sym]] Exists_DISJ, simplified])
```

lemma (**in** *simplification*) *rb_INV_step*[*simp*]:
fixes Q :: ($'a$ :: {*infinite*, *linorder*}, $'b$ :: *linorder*) *fmla*
assumes *rb_INV x Q 𝒬 Q′* ∈ *fixbound 𝒬 x cov x Q′ G*
shows *rb_INV x Q* (*insert* (*cp* (*Q′* ⊥ *x*)) (*insert* (*simp* (*Conj Q′* (*DISJ* (*qps G*)))) (*𝒬* − {*Q′*} ∪ (λ*y*. *cp* (*Q′*[*x* → *y*])) ‘ *eqs x G*)))
proof (*rule rb_INV_I*, *goal_cases finite equiv rrb fv simplified*)
case *finite*
from *assms*(*1*,*3*) **show** *?case* **by** *simp*
next
case *equiv*
from *assms* **show** *?case*
unfolding *rb_INV_def*
by (*auto 0 5 simp*: *fixbound_fv exists_cp_erase exists_cp_subst eqs_noteq exists_Exists image_image image_Un insert_commute ac_simps dest*: *fixbound_in elim*!: *equiv_trans intro*: *equiv_trans*[*OF DISJ_push_in*] *equiv_trans*[*OF DISJ_insert_reorder′*] *equiv_trans*[*OF DISJ_insert_reorder*] *intro*!: *equiv_trans*[*OF DISJ_exists_pull_out*] *equiv_trans*[*OF equiv_Disj_cong*[*OF cov_Exists_equiv equiv_refl*]] *equiv_trans*[*OF equiv_Disj_cong*[*OF equiv_Disj_cong*[*OF equiv_Exists_exists_cong*[*OF equiv_refl*] *equiv_refl*] *equiv_refl*] *equiv_refl*]] *simp del*: *cp.simps*)
next
case (*rrb Q*)
with *assms* **show** *?case*
unfolding *rb_INV_def*
by (*auto intro*!: *rrb_cp_subst rrb_cp*[*OF rrb_erase*] *rrb_simp*[*of Conj _ _*] *dest*: *fixbound_in simp del*: *cp.simps*)
next
case (*fv Q′*)
with *assms* **show** *?case*
unfolding *rb_INV_def*
by (*auto 0 4 dest*!: *fv_cp*[*THEN set_mp*] *fv_simp*[*THEN set_mp*] *fv_DISJ*[*THEN set_mp*, *rotated 1*] *fv_erase*[*THEN set_mp*] *cov_fv*[*OF assms*(*3*) *_ qps_in*, *rotated*] *cov_fv*[*OF assms*(*3*) *_ eqs_in*, *rotated*] *dest*: *fixbound_in simp*: *fv_subst fixbound_fv split*: *if_splits simp del*: *cp.simps*)
next
case (*simplified Q′*)
with *assms* **show** *?case*
unfolding *rb_INV_def* **by** (*auto simp*: *simplified_simp simplified_cp simp del*: *cp.simps*)
qed

lemma (**in** *simplification*) *rb_correct*:
fixes Q :: ($'a$ :: {*linorder*, *infinite*}, $'b$ :: *linorder*) *fmla*
shows *rb Q* ≤ *rb_spec Q*
proof (*induct Q rule*: *rb.induct*[*case_names Neg Disj Conj Exists Pred Bool Eq*])
case (*Exists x Q*)
then show *?case*
unfolding *rb.simps rb_spec_def bind_rule_complete*
by (*rule order_trans*, *refine_vcg WHILEIT_rule*[**where** *R*=*measure* (λ*𝒬*. *card* (*fixbound 𝒬 x*))])
(*auto simp*: *rb_INV_rrb rrb_simp simplified_simp fixbound_fv equiv_trans*[*OF equiv_Exists_cong rb_INV_equiv*] *cov_fixbound fixbound_empty_Gen card_gt_0_iff UNION_singleton_eq_range subset_eq intro*!: *equiv_simp*[*THEN equiv_trans*, *THEN equiv_sym*, *OF equiv_sym*]

dest!: *fv_DISJ*[*THEN set_mp*, *rotated 1*] *fv_simp*[*THEN set_mp*] *elim*!: *bspec elim*: *rb_INV_fv simp del*: *cp.simps*)
qed (*auto simp*: *rb_spec_def bind_rule_complete rrb_simp simplified_simp subset_eq dest*!: *fv_simp*[*THEN set_mp*]
elim!: *order_trans intro*!: *equiv_simp*[*THEN equiv_trans*, *THEN equiv_sym*, *OF equiv_sym*] *simp del*: *cp.simps*)

4 Refining the Non-Deterministic *simplification.rb* Function

fun *gen_size* **where**
gen_size (*Bool b*) = *1*
| *gen_size* (*Eq x t*) = *1*
| *gen_size* (*Pred p ts*) = *1*
| *gen_size* (*Neg* (*Neg Q*)) = *Suc* (*gen_size Q*)
| *gen_size* (*Neg* (*Conj Q1 Q2*)) = *Suc* (*Suc* (*gen_size* (*Neg Q1*) + *gen_size* (*Neg Q2*)))
| *gen_size* (*Neg* (*Disj Q1 Q2*)) = *Suc* (*Suc* (*gen_size* (*Neg Q1*) + *gen_size* (*Neg Q2*)))
| *gen_size* (*Neg Q*) = *Suc* (*gen_size Q*)
| *gen_size* (*Conj Q1 Q2*) = *Suc* (*gen_size Q1* + *gen_size Q2*)
| *gen_size* (*Disj Q1 Q2*) = *Suc* (*gen_size Q1* + *gen_size Q2*)
| *gen_size* (*Exists x Q*) = *Suc* (*gen_size Q*)

function (*sequential*) *gen_impl* **where**
gen_impl x (*Bool False*) = [{}]
| *gen_impl x* (*Bool True*) = []
| *gen_impl x* (*Eq y* (*Const c*)) = (*if x* = *y then* [{*Eq y* (*Const c*)}] *else* [])
| *gen_impl x* (*Eq y* (*Var z*)) = []
| *gen_impl x* (*Pred p ts*) = (*if x* ∈ *fv_terms_set ts then* [{*Pred p ts*}] *else* [])
| *gen_impl x* (*Neg* (*Neg Q*)) = *gen_impl x Q*
| *gen_impl x* (*Neg* (*Conj Q1 Q2*)) = *gen_impl x* (*Disj* (*Neg Q1*) (*Neg Q2*))
| *gen_impl x* (*Neg* (*Disj Q1 Q2*)) = *gen_impl x* (*Conj* (*Neg Q1*) (*Neg Q2*))
| *gen_impl x* (*Neg* _) = []
| *gen_impl x* (*Disj Q1 Q2*) = [*G1* ∪ *G2*. *G1* ← *gen_impl x Q1*, *G2* ← *gen_impl x Q2*]
| *gen_impl x* (*Conj Q1* (*y* ≈ *z*)) = (*if x* = *y then List.union* (*gen_impl x Q1*) (*map* (*image* (λ*Q*. *cp* (*Q*[*z* → *x*]))) (*gen_impl z Q1*))
else if x = *z then List.union* (*gen_impl x Q1*) (*map* (*image* (λ*Q*. *cp* (*Q*[*y* → *x*]))) (*gen_impl y Q1*))
else gen_impl x Q1)|
gen_impl x (*Conj Q1 Q2*) = *List.union* (*gen_impl x Q1*) (*gen_impl x Q2*)
| *gen_impl x* (*Exists y Q*) = (*if x* = *y then* [] *else map* (*image* (*exists y*)) (*gen_impl x Q*))
by *pat_completeness auto*
termination by (*relation measure* (λ(*x*, *Q*). *gen_size Q*)) *simp_all*

lemma *gen_impl_gen*: *G* ∈ *set* (*gen_impl x Q*) ⟹ *gen x Q G*
by (*induct x Q arbitrary*: *G rule*: *gen_impl.induct*)
(*auto 5 2 simp*: *fv_terms_set_def intro*: *gen.intros simp*: *image_iff split*: *if_splits*)

lemma *gen_gen_impl*: *gen x Q G* ⟹ *G* ∈ *set* (*gen_impl x Q*)
proof (*induct x Q G rule*: *gen.induct*)
case (*7 x Q1 G Q2*)
then show *?case*
proof (*cases Q2*)
case (*Eq x t*)
with *7* **show** *?thesis*
by (*cases t*) *auto*
qed *auto*
qed (*auto elim*!: *ap.cases simp*: *image_iff*)

lemma *set_gen_impl*: *set* (*gen_impl x Q*) = {*G*. *gen x Q G*}
by (*auto simp*: *gen_impl_gen gen_gen_impl*)

definition *flat xss* = *fold List.union xss* []

primrec *cov_impl* **where**
cov_impl x (*Bool b*) = [{}]
| *cov_impl x* (*Eq y t*) = (*case t of*
Const c ⇒ [*if x* = *y then* {*Eq y* (*Const c*)} *else* {}]
| *Var z* ⇒ [*if x* = *y* ∧ *x* ≠ *z then* {*x* ≈ *z*}
else if x = *z* ∧ *x* ≠ *y then* {*x* ≈ *y*}
else {}])
| *cov_impl x* (*Pred p ts*) = [*if x* ∈ *fv_terms_set ts then* {*Pred p ts*} *else* {}]
| *cov_impl x* (*Neg Q*) = *cov_impl x Q*
| *cov_impl x* (*Disj Q1 Q2*) = (*case* (*cp* (*Q1* ⊥ *x*), *cp* (*Q2* ⊥ *x*)) *of*
(*Bool True*, *Bool True*) ⇒ *List.union* (*cov_impl x Q1*) (*cov_impl x Q2*)
| (*Bool True*, _) ⇒ *cov_impl x Q1*
| (_, *Bool True*) ⇒ *cov_impl x Q2*
| (_, _) ⇒ [*G1* ∪ *G2*. *G1* ← *cov_impl x Q1*, *G2* ← *cov_impl x Q2*])
| *cov_impl x* (*Conj Q1 Q2*) = (*case* (*cp* (*Q1* ⊥ *x*), *cp* (*Q2* ⊥ *x*)) *of*
(*Bool False*, *Bool False*) ⇒ *List.union* (*cov_impl x Q1*) (*cov_impl x Q2*)
| (*Bool False*, _) ⇒ *cov_impl x Q1*
| (_, *Bool False*) ⇒ *cov_impl x Q2*
| (_, _) ⇒ [*G1* ∪ *G2*. *G1* ← *cov_impl x Q1*, *G2* ← *cov_impl x Q2*])
| *cov_impl x* (*Exists y Q*) = (*if x* = *y then* [{}] *else flat* (*map* (λ*G*.
(*if x* ≈ *y* ∈ *G then* [*exists y* ‘ (*G* − {*x* ≈ *y*}) ∪ (λ*Q*. *cp* (*Q*[*y* → *x*])) ‘ *G*′. *G*′ ← *gen_impl y Q*]
else [*exists y* ‘ *G*])) (*cov_impl x Q*)))

lemma *union_empty_iff*: *List.union xs ys* = [] ⟷ *xs* = [] ∧ *ys* = []
by (*induct xs arbitrary*: *ys*) (*force simp*: *List.union_def List.insert_def*)+

lemma *fold_union_empty_iff*: *fold List.union xss ys* = [] ⟷ (∀ *xs* ∈ *set xss*. *xs* = []) ∧ *ys* = []
by (*induct xss arbitrary*: *ys*) (*auto simp*: *union_empty_iff*)

lemma *flat_empty_iff*: *flat xss* = [] ⟷ (∀ *xs* ∈ *set xss*. *xs* = [])
by (*auto simp*: *flat_def fold_union_empty_iff*)

lemma *set_fold_union*: *set* (*fold List.union xss ys*) = (⋃ (*set* ‘ *set xss*)) ∪ *set ys*
by (*induct xss arbitrary*: *ys*) *auto*

lemma *set_flat*: *set* (*flat xss*) = ⋃ (*set* ‘ *set xss*)
unfolding *flat_def* **by** (*auto simp*: *set_fold_union*)

lemma *rrb_cov_impl*: *rrb Q* ⟹ *cov_impl x Q* ≠ []
proof (*induct Q arbitrary*: *x*)
case (*Exists y Q*)
then show *?case*
by (*cases* ∃ *G* ∈ *set* (*cov_impl x Q*). *x* ≈ *y* ∈ *G*)
(*auto simp*: *flat_empty_iff image_iff dest*: *gen_gen_impl intro*!: *UnI1 bexI*[*rotated*])
qed (*auto split*: *term.splits fmla.splits bool.splits simp*: *union_empty_iff*)

lemma *cov_Eq_self*: *cov x* (*y* ≈ *y*) {}
by (*metis Un_absorb cov.Eq_self cov.nonfree fv.simps*(3) *fv_term_set.simps*(1) *singletonD*)

lemma *cov_impl_cov*: *G* ∈ *set* (*cov_impl x Q*) ⟹ *cov x Q G*
proof (*induct Q arbitrary*: *x G*)
case (*Eq y t*)

```
  then show ?case
    by (auto simp: cov_Eq_self intro: cov.intros ap.intros split: term.splits)
qed (auto simp: set_flat set_gen_impl intro: cov.intros ap.intros
  split: term.splits fmla.splits bool.splits if_splits)

definition fixbound_impl 𝒬 x = filter (λQ. x ∈ fv Q ∧ gen_impl x Q = []) (sorted_list_of_set 𝒬)

lemma set_fixbound_impl: finite 𝒬 ⟹ set (fixbound_impl 𝒬 x) = fixbound 𝒬 x
  by (auto simp: fixbound_def nongens_def fixbound_impl_def set_gen_impl
    dest: arg_cong[of _ _ set] simp flip: List.set_empty)

lemma fixbound_empty_iff: finite 𝒬 ⟹ fixbound 𝒬 x ≠ {} ⟷ fixbound_impl 𝒬 x ≠ []
  by (auto simp: set_fixbound_impl dest: arg_cong[of _ _ set] simp flip: List.set_empty)

lemma fixbound_impl_hd_in: finite 𝒬 ⟹ fixbound_impl 𝒬 x = y # ys ⟹ y ∈ 𝒬
  by (auto simp: fixbound_impl_def dest!: arg_cong[of _ _ set])

fun (in simplification) rb_impl :: ('a :: {infinite, linorder}, 'b :: linorder) fmla ⇒ ('a, 'b) fmla nres where
  rb_impl (Neg Q) = do { Q' ← rb_impl Q; RETURN (simp (Neg Q'))}
| rb_impl (Disj Q1 Q2) = do { Q1' ← rb_impl Q1; Q2' ← rb_impl Q2; RETURN (simp (Disj Q1' Q2'))}
| rb_impl (Conj Q1 Q2) = do { Q1' ← rb_impl Q1; Q2' ← rb_impl Q2; RETURN (simp (Conj Q1' Q2'))}
| rb_impl (Exists x Q) = do {
    Q' ← rb_impl Q;
    𝒬 ← WHILE
      (λ𝒬. fixbound_impl 𝒬 x ≠ []) (λ𝒬. do {
        Qfix ← RETURN (hd (fixbound_impl 𝒬 x));
        G ← RETURN (hd (cov_impl x Qfix));
        RETURN (𝒬 − {Qfix} ∪
          {simp (Conj Qfix (DISJ (qps G)))} ∪
          (⋃y ∈ eqs x G. {cp (Qfix[x → y])}) ∪
          {cp (Qfix ⊥ x)})})
      (flat_Disj Q');
    RETURN (simp (DISJ (exists x ` 𝒬)))}
| rb_impl Q = do { RETURN (simp Q) }

lemma (in simplification) rb_impl_refines_rb: rb_impl Q ≤ rb Q
  apply (induct Q)
  apply (unfold rb.simps rb_impl.simps)
  apply refine_mono
  apply refine_mono
  apply refine_mono
  apply refine_mono
  apply refine_mono
  apply refine_mono
  apply refine_mono
  subgoal for x Q' Q
    apply (rule order_trans[OF WHILE_le_WHILEI[where I=rb_INV x Q]])
    apply (rule order_trans[OF WHILEI_le_WHILEIT])
    apply (rule WHILEIT_refine[OF _ _ _ refine_IdI, THEN refine_IdD])
      apply (simp_all add: fixbound_empty_iff) [3]
    apply refine_mono
     apply (auto simp flip: set_fixbound_impl simp: neq_Nil_conv fixbound_impl_hd_in
       intro!: cov_impl_cov rrb_cov_impl hd_in_set rb_INV_rrb)
    done
  done
```

fun (**in** *simplification*) *rb_impl_det* :: (*'a* :: {*infinite*, *linorder*}, *'b* :: *linorder*) *fmla* ⇒ (*'a*, *'b*) *fmla dres*
where
rb_impl_det (*Neg Q*) = *do* { *Q'* ← *rb_impl_det Q*; *dRETURN* (*simp* (*Neg Q'*))}
| *rb_impl_det* (*Disj Q1 Q2*) = *do* { *Q1'* ← *rb_impl_det Q1*; *Q2'* ← *rb_impl_det Q2*; *dRETURN* (*simp* (*Disj Q1' Q2'*))}
| *rb_impl_det* (*Conj Q1 Q2*) = *do* { *Q1'* ← *rb_impl_det Q1*; *Q2'* ← *rb_impl_det Q2*; *dRETURN* (*simp* (*Conj Q1' Q2'*))}
| *rb_impl_det* (*Exists x Q*) = *do* {
  Q' ← *rb_impl_det Q*;
  𝒬 ← *dWHILE*
   (λ*𝒬*. *fixbound_impl 𝒬 x* ≠ []) (λ*𝒬*. *do* {
    Qfix ← *dRETURN* (*hd* (*fixbound_impl 𝒬 x*));
    G ← *dRETURN* (*hd* (*cov_impl x Qfix*));
    dRETURN (*𝒬* − {*Qfix*} ∪
     {*simp* (*Conj Qfix* (*DISJ* (*qps G*)))} ∪
     (⋃ *y* ∈ *eqs x G*. {*cp* (*Qfix*[*x* → *y*])}) ∪
     {*cp* (*Qfix* ⊥ *x*)})})
   (*flat_Disj Q'*);
  dRETURN (*simp* (*DISJ* (*exists x ' 𝒬*)))}
| *rb_impl_det Q* = *do* { *dRETURN* (*simp Q*) }

lemma (**in** *simplification*) *rb_impl_det_refines_rb_impl*: *nres_of* (*rb_impl_det Q*) ≤ *rb_impl Q*
by (*induct Q*; *unfold rb_impl.simps rb_impl_det.simps*) *refine_transfer+*

lemmas (**in** *simplification*) *RB_correct* =
rb_impl_det_refines_rb_impl[*THEN order_trans*, *OF*
rb_impl_refines_rb[*THEN order_trans*, *OF*
rb_correct]]

5 Restricting Free Variables

definition *fixfree* :: ((*'a*, *'b*) *fmla* × *nat rel*) *set* ⇒ ((*'a*, *'b*) *fmla* × *nat rel*) *set* **where**
fixfree 𝒬fin = {(*Qfix*, *Qeq*) ∈ *𝒬fin*. *nongens Qfix* ≠ {}}

definition *disjointvars Q Qeq* = (⋃ *V* ∈ *classes Qeq*. *if V* ∩ *fv Q* = {} *then V else* {})

fun *Conjs* **where**
Conjs Q [] = *Q*
| *Conjs Q* ((*x*, *y*) # *xys*) = *Conjs* (*Conj Q* (*x* ≈ *y*)) *xys*

function (*sequential*) *Conjs_disjoint* **where**
Conjs_disjoint Q xys = (*case find* (λ(*x*,*y*). {*x*, *y*} ∩ *fv Q* ≠ {}) *xys of*
  None ⇒ *Conjs Q xys*
 | *Some* (*x*, *y*) ⇒ *Conjs_disjoint* (*Conj Q* (*x* ≈ *y*)) (*remove1* (*x*, *y*) *xys*))
by *pat_completeness auto*
termination
by (*relation measure* (λ(*Q*, *xys*). *length xys*))
 (*auto split*: *if_splits simp*: *length_remove1 neq_Nil_conv dest*!: *find_SomeD dest*: *length_pos_if_in_set*)

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

definition *CONJ* **where**
CONJ = (λ(*Q*, *Qeq*). *Conjs Q* (*sorted_list_of_set Qeq*))

definition *CONJ_disjoint* **where**
CONJ_disjoint = (λ(*Q*, *Qeq*). *Conjs_disjoint Q* (*sorted_list_of_set Qeq*))

definition *inf* **where**
inf $\mathcal{Q}fin$ Q = {(Q', Qeq) ∈ $\mathcal{Q}fin$. *disjointvars* Q' Qeq ≠ {} ∨ *fv* Q' ∪ *Field* Qeq ≠ *fv* Q}

definition *FV* **where**
FV Q $Qfin$ $Qinf$ ≡ (*fv* $Qfin$ = *fv* Q ∨ $Qfin$ = *Bool False*) ∧ *fv* $Qinf$ = {}

definition *EVAL* **where**
EVAL Q $Qfin$ $Qinf$ ≡ (∀ I. *finite* (*adom* I) ⟶ (*if eval* $Qinf$ I = {} *then*
eval $Qfin$ I = *eval* Q I *else infinite* (*eval* Q I)))

definition *EVAL′* **where**
EVAL′ Q $Qfin$ $Qinf$ ≡ (∀ I. *finite* (*adom* I) ⟶ (*if eval* $Qinf$ I = {} *then*
eval_on (*fv* Q) $Qfin$ I = *eval* Q I *else infinite* (*eval* Q I)))

definition (**in** *simplification*) *split_spec* :: ('*a* :: {*infinite*, *linorder*}, '*b* :: *linorder*) *fmla* ⇒ (('*a*, '*b*) *fmla* × ('*a*, '*b*) *fmla*) *nres* **where**
split_spec Q = *SPEC* (λ($Qfin$, $Qinf$). *sr* $Qfin$ ∧ *sr* $Qinf$ ∧ *FV* Q $Qfin$ $Qinf$ ∧ *EVAL* Q $Qfin$ $Qinf$ ∧
simplified $Qfin$ ∧ *simplified* $Qinf$)

definition (**in** *simplification*) *assemble* = (λ($\mathcal{Q}fin$, $\mathcal{Q}inf$). (*simp* (*DISJ* (*CONJ_disjoint* ‘ $\mathcal{Q}fin$)), *simp* (*DISJ* (*close* ‘ $\mathcal{Q}inf$))))

fun *leftfresh* **where**
leftfresh Q [] = *True*
| *leftfresh* Q ((x, y) # xys) = (x ∉ *fv* Q ∧ *leftfresh* (*Conj* Q (x ≈ y)) xys)

definition (**in** *simplification*) *wf_state* Q P =
(λ($\mathcal{Q}fin$, $\mathcal{Q}inf$). *finite* $\mathcal{Q}fin$ ∧ *finite* $\mathcal{Q}inf$ ∧
(∀ ($Qfix$, Qeq) ∈ $\mathcal{Q}fin$. P $Qfix$ ∧ *simplified* $Qfix$ ∧ (∃ xs. *leftfresh* $Qfix$ xs ∧ *distinct* xs ∧ *set* xs = Qeq)
∧ *fv* $Qfix$ ∪ *Field* Qeq ⊆ *fv* Q ∧ *irrefl* Qeq))

definition (**in** *simplification*) *split_INV1* Q = (λ$\mathcal{Q}pair$. *wf_state* Q *rrb* $\mathcal{Q}pair$ ∧ (*let* ($Qfin$, $Qinf$) = *assemble* $\mathcal{Q}pair$ *in EVAL′* Q $Qfin$ $Qinf$))
definition (**in** *simplification*) *split_INV2* Q = (λ$\mathcal{Q}pair$. *wf_state* Q *sr* $\mathcal{Q}pair$ ∧ (*let* ($Qfin$, $Qinf$) = *assemble* $\mathcal{Q}pair$ *in EVAL′* Q $Qfin$ $Qinf$))

definition (**in** *simplification*) *split* :: ('*a* :: {*infinite*, *linorder*}, '*b* :: *linorder*) *fmla* ⇒ (('*a*, '*b*) *fmla* × ('*a*, '*b*) *fmla*) *nres* **where**
split Q = *do* {
Q' ← *rb* Q;
$\mathcal{Q}pair$ ← $WHILE_T{}^{split\_INV1\ Q}$
(λ($\mathcal{Q}fin$, _). *fixfree* $\mathcal{Q}fin$ ≠ {}) (λ($\mathcal{Q}fin$, $\mathcal{Q}inf$). *do* {
($Qfix$, Qeq) ← *RES* (*fixfree* $\mathcal{Q}fin$);
x ← *RES* (*nongens* $Qfix$);
G ← *SPEC* (*cov* x $Qfix$);
let $\mathcal{Q}fin$ = $\mathcal{Q}fin$ − {($Qfix$, Qeq)} ∪
{(*simp* (*Conj* $Qfix$ (*DISJ* (*qps* G))), Qeq)} ∪
(⋃ y ∈ *eqs* x G. {(*cp* ($Qfix$[x → y]), Qeq ∪ {(x,y)})});
let $\mathcal{Q}inf$ = $\mathcal{Q}inf$ ∪ {*cp* ($Qfix$ ⊥ x)};
RETURN ($\mathcal{Q}fin$, $\mathcal{Q}inf$)})
({(Q', {})}, {});
$\mathcal{Q}pair$ ← $WHILE_T{}^{split\_INV2\ Q}$
(λ($\mathcal{Q}fin$, _). *inf* $\mathcal{Q}fin$ Q ≠ {}) (λ($\mathcal{Q}fin$, $\mathcal{Q}inf$). *do* {
$Qpair$ ← *RES* (*inf* $\mathcal{Q}fin$ Q);
let $\mathcal{Q}fin$ = $\mathcal{Q}fin$ − {$Qpair$};
let $\mathcal{Q}inf$ = $\mathcal{Q}inf$ ∪ {*CONJ* $Qpair$};
RETURN ($\mathcal{Q}fin$, $\mathcal{Q}inf$)})
$\mathcal{Q}pair$;

```
    let (Qfin, Qinf) = assemble Qpair;
    Qinf ← rb Qinf;
    RETURN (Qfin, Qinf)}

lemma finite_fixfree[simp]: finite Q ⟹ finite (fixfree Q)
  unfolding fixfree_def by (auto elim!: finite_subset[rotated])

lemma (in simplification) split_step_in_mult:
  assumes (Qfin, Qeq) ∈ Qfin finite Qfin x ∈ nongens Qfin cov x Qfin G fv Qfin ⊆ F
  shows ((nongens ∘ fst) `# mset_set (insert (simp (Conj Qfin (DISJ (qps G))), Qeq) (Qfin − {(Qfin,
Qeq)} ∪ (λy. (cp (Qfin[x → y]), insert (x, y) Qeq)) ` eqs x G)),
        (nongens ∘ fst) `# mset_set Qfin) ∈ mult {(X, Y). X ⊂ Y ∧ Y ⊆ F}
    (is (?f (insert ?Q (?A ∪ ?B)), ?C) ∈ mult ?R)
proof (subst preorder.mult_DM[where less_eq = (in_rel ?R)^==])
  define X where X = {(Qfin, Qeq)}
  define Y where Y = insert ?Q ?B − (?A ∩ insert ?Q ?B)
  have ?f X ≠ {#}
    unfolding X_def by auto
  moreover from assms(1,2) have ?f X ⊆# ?C
    unfolding X_def by (auto intro!: image_eqI[where x = (Qfin, Qeq)])
  moreover from assms(1,2,4) have XY:
    insert ?Q (?A ∪ ?B) = Qfin − X ∪ Y X ⊆ Qfin (Qfin − X) ∩ Y = {} finite X finite Y
    unfolding X_def Y_def by auto
  with assms(2) have ?f (insert ?Q (?A ∪ ?B)) = ?C − ?f X + ?f Y
    by (force simp: mset_set_Union mset_set_Diff multiset.map_comp o_def
        dest: subset_imp_msubset_mset_set elim: subset_mset.trans
        intro!: subset_imp_msubset_mset_set image_mset_subseteq_mono subset_mset.diff_add_assoc2
        trans[OF image_mset_Diff])
  moreover
  { fix A
    assume A ∈ Y
    then have A ∈ insert ?Q ?B
      unfolding Y_def by blast
    with assms(3,4) have nongens (fst A) ⊆ nongens Qfin − {x}
      using Gen_cp_subst[of _ Qfin x] Gen_simp[OF cov_Gen_qps[OF assms(4)]]
        gen_Gen_simp[OF gen.intros(7)[OF disjI1], of _ Qfin _ DISJ (qps G)]
      by (fastforce simp: nongens_def fv_subst simp del: cp.simps
        intro!: gen.intros(7) dest!: fv_cp[THEN set_mp] fv_simp[THEN set_mp] fv_DISJ[THEN set_mp,
rotated 1]
          elim: cov_fv[OF assms(4) _ qps_in, THEN conjunct2, THEN set_mp]
          cov_fv[OF assms(4) _ eqs_in, THEN conjunct2, THEN set_mp])
    with assms(3) have nongens (fst A) ⊂ nongens Qfin
      by auto
    with assms(5) have ∃ B ∈ X. nongens (fst A) ⊂ nongens (fst B) ∧ nongens (fst B) ⊆ F
      by (auto simp: X_def nongens_def)
  }
  with XY have ⋀A. A ∈# ?f Y ⟹ ∃ B. B ∈# ?f X ∧ A ⊂ B ∧ B ⊆ F
    by auto
  ultimately
  show ∃ X Y. X ≠ {#} ∧ X ⊆# ?C ∧ ?f (insert ?Q (?A ∪ ?B)) = ?C − X + Y ∧ (∀ k. k ∈# Y ⟶
(∃ a. a ∈# X ∧ k ⊂ a ∧ a ⊆ F))
    by blast
qed (unfold_locales, auto)

lemma EVAL_cong:
  Qinf ≜ Qinf' ⟹ fv Qinf = fv Qinf' ⟹ EVAL Q Qfin Qinf = EVAL Q Qfin Qinf'
  using equiv_eval_eqI[of _ Qinf Qinf']
  by (auto simp: EVAL_def)
```

lemma *EVAL′_cong*:
Qinf ≜ *Qinf′* ⟹ *fv Qinf* = *fv Qinf′* ⟹ *EVAL′ Q Qfin Qinf* = *EVAL′ Q Qfin Qinf′*
using *equiv_eval_eqI*[*of* _ *Qinf Qinf′*]
by (*auto simp*: *EVAL′_def*)

lemma *fv_Conjs*[*simp*]: *fv* (*Conjs Q xys*) = *fv Q* ∪ *Field* (*set xys*)
by (*induct Q xys rule*: *Conjs.induct*) *auto*

lemma *fv_Conjs_disjoint*[*simp*]: *distinct xys* ⟹ *fv* (*Conjs_disjoint Q xys*) = *fv Q* ∪ *Field* (*set xys*)
proof (*induct Q xys rule*: *Conjs_disjoint.induct*)
case (*1 Q xys*)
then show *?case*
by (*subst Conjs_disjoint.simps*)
(*auto split*: *option.splits simp*: *Field_def subset_eq dest*: *find_SomeD*(*2*))
qed

lemma *fv_CONJ*[*simp*]: *finite Qeq* ⟹ *fv* (*CONJ* (*Q*, *Qeq*)) = *fv Q* ∪ *Field Qeq*
unfolding *CONJ_def* **by** (*auto dest*!: *fv_cp*[*THEN set_mp*])

lemma *fv_CONJ_disjoint*[*simp*]: *finite Qeq* ⟹ *fv* (*CONJ_disjoint* (*Q*, *Qeq*)) = *fv Q* ∪ *Field Qeq*
unfolding *CONJ_disjoint_def* **by** *auto*

lemma *rrb_Conjs*: *rrb Q* ⟹ *rrb* (*Conjs Q xys*)
by (*induct Q xys rule*: *Conjs.induct*) *auto*

lemma *CONJ_empty*[*simp*]: *CONJ* (*Q*, {}) = *Q*
by (*auto simp*: *CONJ_def*)

lemma *CONJ_disjoint_empty*[*simp*]: *CONJ_disjoint* (*Q*, {}) = *Q*
by (*auto simp*: *CONJ_disjoint_def Conjs_disjoint.simps*)

lemma *Conjs_eq_False_iff*[*simp*]: *irrefl* (*set xys*) ⟹ *Conjs Q xys* = *Bool False* ⟷ *Q* = *Bool False* ∧ *xys* = []
by (*induct Q xys rule*: *Conjs.induct*) (*auto simp*: *Let_def is_Bool_def irrefl_def*)

lemma *CONJ_eq_False_iff*[*simp*]: *finite Qeq* ⟹ *irrefl Qeq* ⟹ *CONJ* (*Q*, *Qeq*) = *Bool False* ⟷ *Q* = *Bool False* ∧ *Qeq* = {}
by (*auto simp*: *CONJ_def*)

lemma *Conjs_disjoint_eq_False_iff*[*simp*]: *irrefl* (*set xys*) ⟹ *Conjs_disjoint Q xys* = *Bool False* ⟷ *Q* = *Bool False* ∧ *xys* = []
proof (*induct Q xys rule*: *Conjs_disjoint.induct*)
case (*1 Q xys*)
then show *?case*
by (*subst Conjs_disjoint.simps*)
(*auto simp*: *Let_def is_Bool_def irrefl_def split*: *option.splits*)
qed

lemma *CONJ_disjoint_eq_False_iff*[*simp*]: *finite Qeq* ⟹ *irrefl Qeq* ⟹ *CONJ_disjoint* (*Q*, *Qeq*) = *Bool False* ⟷ *Q* = *Bool False* ∧ *Qeq* = {}
by (*auto simp*: *CONJ_disjoint_def*)

lemma *sr_Conjs_disjoint*:
distinct xys ⟹ (∀ *V*∈*classes* (*set xys*). *V* ∩ *fv Q* ≠ {}) ⟹ *sr Q* ⟹ *sr* (*Conjs_disjoint Q xys*)
proof (*induct Q xys rule*: *Conjs_disjoint.induct*)
case (*1 Q xys*)
show *?case*

```
  proof (cases find (λ(x, y). {x, y} ∩ fv Q ≠ {}) xys)
    case None
    with 1(2−) show ?thesis
      using classes_intersect_find_not_None[of xys fv Q]
      by (cases xys) (simp_all add: Conjs_disjoint.simps)
  next
    case (Some xy)
    then obtain x y where xy: xy = (x, y) and xy_in: (x, y) ∈ set xys
      by (cases xy) (auto dest!: find_SomeD)
    with Some 1(4) have sr (Conj Q (x ≈ y))
      by (auto dest: find_SomeD simp: sr_Conj_eq)
    moreover from 1(2,3) have ∀ V∈classes (set (remove1 (x, y) xys)). V ∩ fv (Conj Q (x ≈ y)) ≠ {}
      by (subst (asm) insert_remove_id[OF xy_in], unfold classes_insert)
        (auto simp: class_None_eq class_Some_eq split: option.splits if_splits)
    ultimately show ?thesis
      using 1(2−) Some xy 1(1)[OF Some xy[symmetric]]
      by (simp add: Conjs_disjoint.simps)
  qed
qed

lemma sr_CONJ_disjoint:
  inf Qfin Q = {} ⟹ (Qfin, Qeq) ∈ Qfin ⟹ finite Qeq ⟹ sr Qfin ⟹ sr (CONJ_disjoint (Qfin,
Qeq))
  unfolding inf_def disjointvars_def CONJ_disjoint_def prod.case
  by (drule arg_cong[of _ _ λA. (Qfin, Qeq) ∈ A], intro sr_cp sr_Conjs_disjoint)
    (auto simp only: mem_Collect_eq prod.case simp_thms distinct_sorted_list_of_set
      set_sorted_list_of_set SUP_bot_conv classes_nonempty split: if_splits)

lemma equiv_Conjs_cong: Q ≜ Q' ⟹ Conjs Q xys ≜ Conjs Q' xys
  by (induct Q xys arbitrary: Q' rule: Conjs.induct) auto

lemma Conjs_pull_out: Conjs Q (xys @ (x, y) # xys') ≜ Conjs (Conj Q (x ≈ y)) (xys @ xys')
  by (induct Q xys rule: Conjs.induct)
    (auto elim!: equiv_trans intro!: equiv_Conjs_cong intro: equiv_def[THEN iffD2])

lemma Conjs_reorder: distinct xys ⟹ distinct xys' ⟹ set xys = set xys' ⟹ Conjs Q xys ≜ Conjs Q
xys'
proof (induct Q xys arbitrary: xys' rule: Conjs.induct)
  case (2 Q x y xys)
  from 2(4) obtain i where i: i < length xys' xys' ! i = (x, y)
    by (auto simp: set_eq_iff in_set_conv_nth)
  with 2(2−4) have ∗: set xys = set (take i xys') ∪ set (drop (Suc i) xys')
    by (subst (asm) (1 2) id_take_nth_drop[of i xys'])
      (auto simp: set_eq_iff dest: in_set_takeD in_set_dropD)
  from i 2(2,3) show ?case
    by (subst id_take_nth_drop[OF i(1)], subst (asm) (3) id_take_nth_drop[OF i(1)])
      (auto simp: ∗ intro!: equiv_trans[OF _ Conjs_pull_out[THEN equiv_sym]] 2(1))
qed simp

lemma ex_Conjs_disjoint_eq_Conjs:
  distinct xys ⟹ ∃ xys'. distinct xys' ∧ set xys = set xys' ∧ Conjs_disjoint Q xys = Conjs Q xys'
proof (induct Q xys rule: Conjs_disjoint.induct)
  case (1 Q xys)
  show ?case
  proof (cases find (λ(x, y). {x, y} ∩ fv Q ≠ {}) xys)
    case None
    with 1(2) show ?thesis
      by (subst Conjs_disjoint.simps) (auto intro!: exI[of _ xys])
```

```
  next
    case (Some xy)
    with 1(1)[of xy fst xy snd xy] 1(2)
    obtain xys' where distinct xys'
      set xys − {xy} = set xys'
      Conjs_disjoint (Conj Q (fst xy ≈ snd xy)) (remove1 xy xys) =
       Conjs (Conj Q (fst xy ≈ snd xy)) xys'
      by auto
    with Some show ?thesis
      by (subst Conjs_disjoint.simps, intro exI[of _ xy # xys'])
        (auto simp: set_eq_iff dest: find_SomeD)
  qed
qed

lemma Conjs_disjoint_equiv_Conjs:
  assumes distinct xys
  shows Conjs_disjoint Q xys ≜ Conjs Q xys
proof −
  from assms obtain xys' where xys': distinct xys' set xys = set xys' and Conjs_disjoint Q xys = Conjs
Q xys'
    using ex_Conjs_disjoint_eq_Conjs by blast
  note this(3)
  also have ... ≜ Conjs Q xys
    by (intro Conjs_reorder xys' sym assms)
  finally show ?thesis
    by blast
qed

lemma infinite_eval_Conjs: infinite (eval Q I) ⟹ leftfresh Q xys ⟹ infinite (eval (Conjs Q xys) I)
proof (induct Q xys rule: Conjs.induct)
  case (2 Q x y xys)
  then show ?case
    unfolding Conjs.simps
    by (intro 2(1) infinite_eval_Conj) auto
qed simp

lemma leftfresh_fv_subset: leftfresh Q xys ⟹ fv Q' ⊆ fv Q ⟹ leftfresh Q' xys
  by (induct Q xys arbitrary: Q' rule: leftfresh.induct) (auto simp: subset_eq)

lemma fun_upds_map: (∀ x. x ∉ set ys ⟶ σ x = τ x) ⟹ σ[ys :=* map τ ys] = τ
  by (induct ys arbitrary: σ) auto

lemma map_fun_upds: length xs = length ys ⟹ distinct xs ⟹ map (σ[xs :=* ys]) xs = ys
  by (induct xs ys arbitrary: σ rule: list_induct2) auto

lemma zip_map: zip xs (map f xs) = map (λx. (x, f x)) xs
  by (induct xs) auto

lemma filter_sorted_list_of_set:
  finite B ⟹ A ⊆ B ⟹ filter (λx. x ∈ A) (sorted_list_of_set B) = sorted_list_of_set A
proof (induct B arbitrary: A rule: finite_induct)
  case (insert x B)
  then have finite A by (auto simp: finite_subset)
  moreover
  from insert(1,2) have filter (λy. y ∈ A − {x}) (sorted_list_of_set B) =
          filter (λx. x ∈ A) (sorted_list_of_set B)
    by (intro filter_cong) auto
  ultimately show ?case
```

using *insert(1,2,4) insert(3)*[*of* $A - \{x\}$] *sorted_list_of_set_insert_remove*[*of* A x]
by (*cases* $x \in A$) (*auto simp*: *filter_insort filter_insort_triv subset_insert_iff insert_absorb*)
qed *simp*

lemma *infinite_eval_eval_on*[*rotated 2*]:
assumes *fv* $Q \subseteq X$ *finite* X
shows *infinite* (*eval* Q I) $\Longrightarrow$ *infinite* (*eval_on* X Q I)
proof (*erule infinite_surj*[*of* _ λxs. *map snd* (*filter* (λ(*x*,_). $x \in$ *fv* Q) (*zip* (*sorted_list_of_set* X) *xs*))],
unfold eval_deep_def Let_def, *safe*)
fix *xs* σ
assume *len*: *length* (*sorted_list_of_set* (*fv* Q)) = *length xs* **and**
sat Q I (σ[*sorted_list_of_set* (*fv* Q) :=* *xs*]) (**is** *sat* Q I *?*τ)
moreover from *assms len* **have** σ[*sorted_list_of_set* X :=* *map ?*τ (*sorted_list_of_set* X)] = *?*τ
by (*intro fun_upds_map*) *force*
ultimately show *xs* $\in$ (λxs. *map snd* (*filter* (λ(*x*, _). $x \in$ *fv* Q) (*zip* (*sorted_list_of_set* X) *xs*))) ‘
eval_on X Q I **using** *assms*
by (*auto simp*: *eval_on_def image_iff zip_map filter_map o_def filter_sorted_list_of_set map_fun_upds*
intro!: *exI*[*of* _ *map* (σ[*sorted_list_of_set* (*fv* Q) :=* *xs*]) (*sorted_list_of_set* X)] *exI*[*of* _ σ])
qed

lemma *infinite_eval_CONJ_disjoint*:
assumes *infinite* (*eval* Q I) *finite* (*adom* I) *fv* $Q \subseteq X$ *Field* $Qeq \subseteq X$ *finite* X $\exists$*xys*. *distinct xys* $\wedge$ *leftfresh* Q *xys* $\wedge$ *set xys* = Qeq
shows *infinite* (*eval_on* X (*CONJ_disjoint* (Q, Qeq)) I)
proof −
from *assms*(*6*) **obtain** *xys* **where** *distinct xys leftfresh* Q *xys set xys* = Qeq
by *blast*
with *assms*(*1*−*5*) **show** *?thesis*
using *infinite_eval_eval_on*[*OF infinite_eval_Conjs*[*of* Q I *xys*], *of* X] *equiv_eval_on_eqI*[*of* I *Conjs_disjoint* Q (*sorted_list_of_set* Qeq) *Conjs* Q *xys* X]
equiv_trans[*OF Conjs_disjoint_equiv_Conjs*[*of sorted_list_of_set* Qeq Q] *Conjs_reorder*[*of* _ *xys*]]
fv_Conjs[*of* Q *xys*]
by (*force simp*: *CONJ_disjoint_def subset_eq equiv_eval_on_eqI*[*OF* _ *equiv_cp*])
qed

lemma *sat_Conjs*: *sat* (*Conjs* Q *xys*) I σ $\longleftrightarrow$ *sat* Q I σ $\wedge$ ($\forall$ (x, y) $\in$ *set xys*. *sat* ($x \approx y$) I σ)
by (*induct* Q *xys rule*: *Conjs.induct*) *auto*

lemma *sat_Conjs_disjoint*: *sat* (*Conjs_disjoint* Q *xys*) I σ $\longleftrightarrow$ *sat* Q I σ $\wedge$ ($\forall$ (x, y) $\in$ *set xys*. *sat* ($x \approx y$) I σ)
proof (*induct* Q *xys rule*: *Conjs_disjoint.induct*)
case (*1* Q *xys*)
then show *?case*
by (*subst Conjs_disjoint.simps*)
(*auto simp*: *sat_Conjs dest*: *find_SomeD*(*2*) *set_remove1_subset*[*THEN set_mp*] *in_set_remove_cases*[*rotated*] *split*: *option.splits*)
qed

lemma *sat_CONJ*: *finite* Qeq $\Longrightarrow$ *sat* (*CONJ* (Q, Qeq)) I σ $\longleftrightarrow$ *sat* Q I σ $\wedge$ ($\forall$ (x, y) $\in$ Qeq. *sat* ($x \approx y$) I σ)
unfolding *CONJ_def* **by** (*auto simp*: *sat_Conjs*)

lemma *sat_CONJ_disjoint*: *finite* Qeq $\Longrightarrow$ *sat* (*CONJ_disjoint* (Q, Qeq)) I σ $\longleftrightarrow$ *sat* Q I σ $\wedge$ ($\forall$ (x, y) $\in$ Qeq. *sat* ($x \approx y$) I σ)
unfolding *CONJ_disjoint_def* **by** (*auto simp*: *sat_Conjs_disjoint*)

lemma *Conjs_inject*: *Conjs* Q *xys* = *Conjs* Q' *xys* $\longleftrightarrow$ $Q = Q'$
by (*induct* Q *xys arbitrary*: Q' *rule*: *Conjs.induct*) *auto*

```
lemma nonempty_disjointvars_infinite:
  assumes disjointvars (Qfin :: ('a :: infinite, 'b) fmla) Qeq ≠ {}
    finite Qeq fv Qfin ∪ Field Qeq ⊆ X finite X sat Qfin I σ ∀(x, y)∈Qeq. σ x = σ y
  shows infinite (eval_on X (CONJ_disjoint (Qfin, Qeq)) I)
proof −
  from assms(1) obtain x V where xV: V ∈ classes Qeq x ∈ V V ∩ fv Qfin = {}
    by (auto simp: disjointvars_def)
  show ?thesis
  proof (rule infinite_surj[OF infinite_UNIV, of λds. ds ! index (sorted_list_of_set X) x], safe)
    fix z
    let ?ds = map (λv. if v ∈ V then z else σ v) (sorted_list_of_set X)
    from xV have x ∈ Field Qeq
      by (metis UnionI classes_cover)
    { fix a b
      assume ∗: (a, b) ∈ Qeq
      from this edge_same_class[OF xV(1) this] assms(3,6) have a ∈ X b ∈ X a ∈ V ⟷ b ∈ V σ a
= σ b
        by (auto dest: FieldI1 FieldI2)
      with xV(1) assms(3,4) have (σ[sorted_list_of_set X :=* ?ds]) a = (σ[sorted_list_of_set X :=*
?ds]) b
        by (subst (1 2) fun_upds_in) auto
    }
    with assms(2−) xV ‹x ∈ Field Qeq›
    show z ∈ (λds. ds ! index (sorted_list_of_set X) x) ‘ eval_on X (CONJ_disjoint (Qfin, Qeq)) I
     by (auto simp: eval_on_def CONJ_disjoint_def sat_Conjs_disjoint Let_def image_iff fun_upds_in
subset_eq
        intro!: exI[of _ map (λv. if v ∈ V then z else σ v) (sorted_list_of_set X)] exI[of _ σ]
        elim!: sat_fv_cong[THEN iffD1, rotated −1])
  qed
qed

lemma EVAL'_EVAL: EVAL' Q Qfin Qinf ⟹ FV Q Qfin Qinf ⟹ EVAL Q Qfin Qinf
  unfolding EVAL_def EVAL'_def FV_def
  by (subst (2) eval_def) auto

lemma cpropagated_Conjs_disjoint:
  distinct xys ⟹ irrefl (set xys) ⟹ ∀ V∈classes (set xys). V ∩ fv Q ≠ {} ⟹ cpropagated Q ⟹
cpropagated (Conjs_disjoint Q xys)
proof (induct Q xys rule: Conjs_disjoint.induct)
  case (1 Q xys)
  show ?case
  proof (cases find (λ(x, y). {x, y} ∩ fv Q ≠ {}) xys)
    case None
    with 1(2−) show ?thesis
      using classes_intersect_find_not_None[of xys fv Q]
      by (cases xys) (simp_all add: Conjs_disjoint.simps)
  next
    case (Some xy)
    then obtain x y where xy: xy = (x, y) and xy_in: (x, y) ∈ set xys
      by (cases xy) (auto dest!: find_SomeD)
    with Some 1(3,5) have cpropagated (Conj Q (x ≈ y))
      by (auto dest: find_SomeD simp: cpropagated_def irrefl_def is_Bool_def)
    moreover from 1(2,4) have ∀ V∈classes (set (remove1 (x, y) xys)). V ∩ fv (Conj Q (x ≈ y)) ≠ {}
      by (subst (asm) insert_remove_id[OF xy_in], unfold classes_insert)
        (auto simp: class_None_eq class_Some_eq split: option.splits if_splits)
    moreover from 1(3) have irrefl (set xys − {(x, y)})
      by (auto simp: irrefl_def)
```

ultimately show *?thesis*
using *1(2−) Some xy 1(1)*[*OF Some xy*[*symmetric*]]
by (*simp add*: *Conjs_disjoint.simps*)
qed
qed

lemma (**in** *simplification*) *simplified_Conjs_disjoint*:
distinct xys ⟹ *irrefl* (*set xys*) ⟹ ∀ *V*∈*classes* (*set xys*). *V* ∩ *fv Q* ≠ {} ⟹ *simplified Q* ⟹ *simplified* (*Conjs_disjoint Q xys*)
proof (*induct Q xys rule*: *Conjs_disjoint.induct*)
case (*1 Q xys*)
show *?case*
proof (*cases find* (λ(*x*, *y*). {*x*, *y*} ∩ *fv Q* ≠ {}) *xys*)
case *None*
with *1(2−)* **show** *?thesis*
using *classes_intersect_find_not_None*[*of xys fv Q*]
by (*cases xys*) (*simp_all add*: *Conjs_disjoint.simps*)
next
case (*Some xy*)
then obtain *x y* **where** *xy*: *xy* = (*x*, *y*) **and** *xy_in*: (*x*, *y*) ∈ *set xys*
by (*cases xy*) (*auto dest*!: *find_SomeD*)
with *Some 1(3,5)* **have** *simplified* (*Conj Q* (*x* ≈ *y*))
by (*auto dest*: *find_SomeD simp*: *irrefl_def intro*!: *simplified_Conj_eq*)
moreover from *1(2,4)* **have** ∀ *V*∈*classes* (*set* (*remove1* (*x*, *y*) *xys*)). *V* ∩ *fv* (*Conj Q* (*x* ≈ *y*)) ≠ {}
by (*subst* (*asm*) *insert_remove_id*[*OF xy_in*], *unfold classes_insert*)
(*auto simp*: *class_None_eq class_Some_eq split*: *option.splits if_splits*)
moreover from *1(3)* **have** *irrefl* (*set xys* − {(*x*, *y*)})
by (*auto simp*: *irrefl_def*)
ultimately show *?thesis*
using *1(2−) Some xy 1(1)*[*OF Some xy*[*symmetric*]]
by (*simp add*: *Conjs_disjoint.simps*)
qed
qed

lemma *disjointvars_empty_iff*: *disjointvars Q Qeq* = {} ⟷ (∀ *V*∈*classes Qeq*. *V* ∩ *fv Q* ≠ {})
unfolding *disjointvars_def UNION_empty_conv*
using *classes_nonempty* **by** *auto*

lemma *cpropagated_CONJ_disjoint*:
finite Qeq ⟹ *irrefl Qeq* ⟹ *disjointvars Q Qeq* = {} ⟹ *cpropagated Q* ⟹ *cpropagated* (*CONJ_disjoint* (*Q*, *Qeq*))
unfolding *CONJ_disjoint_def prod.case disjointvars_empty_iff*
by (*rule cpropagated_Conjs_disjoint*) *auto*

lemma (**in** *simplification*) *simplified_CONJ_disjoint*:
finite Qeq ⟹ *irrefl Qeq* ⟹ *disjointvars Q Qeq* = {} ⟹ *simplified Q* ⟹ *simplified* (*CONJ_disjoint* (*Q*, *Qeq*))
unfolding *CONJ_disjoint_def prod.case disjointvars_empty_iff*
by (*rule simplified_Conjs_disjoint*) *auto*

lemma (**in** *simplification*) *split_INV1_init*:
rrb Q′ ⟹ *simplified Q′* ⟹ *Q* ≜ *Q′* ⟹ *fv Q′* ⊆ *fv Q* ⟹ *split_INV1 Q* ({(*Q′*, {})}, {})
by (*auto simp add*: *split_INV1_def wf_state_def assemble_def FV_def EVAL′_def eval_def*[*symmetric*] *eval_simp_False irrefl_def*
sat_simp equiv_def intro!: *equiv_eval_on_eval_eqI del*: *equalityI dest*: *fv_simp*[*THEN set_mp*] *split*: *prod.splits*)

lemma (**in** *simplification*) *split_INV1_I*:

wf_state Q rrb (*Qfin*, *Qinf*) ⟹ *EVAL′ Q* (*simp* (*DISJ* (*CONJ_disjoint* ‘ *Qfin*))) (*simp* (*DISJ* (*close* ‘ *Qinf*))) ⟹
split_INV1 Q (*Qfin*, *Qinf*)
unfolding *split_INV1_def assemble_def* **by** *auto*

lemma *EVAL′_I*:
(⋀*I*. *finite* (*adom I*) ⟹ *eval Qinf I* = {} ⟹ *eval_on* (*fv Q*) *Qfin I* = *eval Q I*) ⟹
(⋀*I*. *finite* (*adom I*) ⟹ *eval Qinf I* ≠ {} ⟹ *infinite* (*eval Q I*)) ⟹ *EVAL′ Q Qfin Qinf*
unfolding *EVAL′_def* **by** *auto*

lemma (**in** *simplification*) *wf_state_Un*:
wf_state Q P (*Qfin*, *Qinf*) ⟹ *wf_state Q P* (*insert Qpair Qnew*, {*Q′*}) ⟹
wf_state Q P (*insert Qpair* (*Qfin* ∪ *Qnew*), *insert Q′ Qinf*)
by (*auto simp*: *wf_state_def*)

lemma (**in** *simplification*) *wf_state_Diff*:
wf_state Q P (*Qfin*, *Qinf*) ⟹ *wf_state Q P* (*Qfin* − *Qnew*, *Qinf*)
by (*auto simp*: *wf_state_def*)

lemma (**in** *simplification*) *split_INV1_step*:
assumes *split_INV1 Q* (*Qfin*, *Qinf*) (*Qfin*, *Qeq*) ∈ *fixfree Qfin x* ∈ *nongens Qfin cov x Qfin G*
shows *split_INV1 Q*
(*insert* (*simp* (*Conj Qfin* (*DISJ* (*qps G*))), *Qeq*)
(*Qfin* − {(*Qfin*, *Qeq*)} ∪ (λ*y*. (*cp* (*Qfin*[*x* → *y*]), *insert* (*x*, *y*) *Qeq*)) ‘ *eqs x G*),
insert (*cp* (*Qfin* ⊥ *x*)) *Qinf*)
(**is** *split_INV1 Q* (*?Qfin*, *?Qinf*))
proof (*intro split_INV1_I EVAL′_I*, *goal_cases wf fin inf*)
case *wf*
from *assms*(*1*) **have** *wf*: *wf_state Q rrb* (*Qfin*, *Qinf*)
by (*auto simp*: *split_INV1_def*)
with *assms*(*2*,*3*) **obtain** *xys* **where** ∗:
x ∈ *fv Qfin* (*Qfin*, *Qeq*) ∈ *Qfin finite Qfin finite Qeq finite Qinf fv Qfin* ⊆ *fv Q Field Qeq* ⊆ *fv Q*
distinct xys leftfresh Qfin xys set xys = *Qeq rrb Qfin irrefl Qeq*
by (*auto simp*: *fixfree_def nongens_def wf_state_def*)
moreover from ∗ **have** ∃ *xs*. *leftfresh* (*simp* (*Conj Qfin* (*DISJ* (*qps G*)))) *xs* ∧ *distinct xs* ∧ *set xs* = *set xys*
using *cov_fv*[*OF assms*(*4*) _ *qps_in*] *assms*(*4*)
by (*intro exI*[*of* _ *xys*])
(*force elim*!: *leftfresh_fv_subset dest*!: *fv_simp*[*THEN set_mp*] *fv_DISJ*[*THEN set_mp*, *rotated 1*])
moreover from ∗ **have** ∃ *xs*. *leftfresh* (*cp* (*Qfin*[*x* → *z*])) *xs* ∧ *distinct xs* ∧ *set xs* = *insert* (*x*, *z*) (*set xys*)
if *z* ∈ *eqs x G* **for** *z*
using *cov_fv*[*OF assms*(*4*) _ *eqs_in*, *of z x*] *assms*(*4*) *that*
by (*intro exI*[*of* _ *if* (*x*, *z*) ∈ *set xys then xys else* (*x*, *z*) # *xys*])
(*auto simp*: *fv_subst dest*!: *fv_cp*[*THEN set_mp*] *elim*!: *leftfresh_fv_subset*)
ultimately show *?case*
using *cov_fv*[*OF assms*(*4*) _ *qps_in*] *cov_fv*[*OF assms*(*4*) _ *eqs_in*] *assms*(*4*)
by (*intro wf_state_Un wf_state_Diff wf*)
(*auto simp*: *wf_state_def rrb_simp simplified_simp simplified_cp rrb_cp_subst fv_subst subset_eq irrefl_def*
dest!: *fv_cp*[*THEN set_mp*] *fv_simp*[*THEN set_mp*] *fv_DISJ*[*THEN set_mp*, *rotated 1*])
next
case (*fin I*)
note *eq* = *trans*[*OF sat_simp sat_DISJ*, *symmetric*]
from *assms* **have** ∗:
x ∈ *fv Qfin* (*Qfin*, *Qeq*) ∈ *Qfin fv Qfin* ⊆ *fv Q Field Qeq* ⊆ *fv Q* **and**
finite[*simp*]: *finite Qfin finite Qeq finite Qinf*
by (*auto simp*: *split_INV1_def fixfree_def nongens_def wf_state_def*)

```
  with fin have unsat: ∀σ. ¬ sat (Qfin ⊥ x) I σ and ∀x∈Qinf. ∀σ. ¬ sat x I σ
    by (auto simp: eval_empty_close eval_simp_DISJ_closed)
  with fin(1) assms(1) ∗ have eval_on (fv Q) (simp (DISJ (CONJ_disjoint ‘ Qfin))) I = eval Q I
    unfolding split_INV1_def Let_def assemble_def prod.case EVAL'_def
    by (auto simp: eval_empty_close eval_simp_DISJ_closed)
  with assms(4) show ?case
  proof (elim trans[rotated], intro eval_on_cong box_equals[OF _ eq eq])
    fix σ
    from ∗ have (∃ Q∈Qfin. sat (CONJ_disjoint Q) I σ) ⟷
        sat (CONJ_disjoint (Qfin, Qeq)) I σ ∨ (∃ Q∈Qfin − {(Qfin, Qeq)}. sat (CONJ_disjoint Q) I σ)
      using assms(4) by (auto simp: fixfree_def)
    also have sat (CONJ_disjoint (Qfin, Qeq)) I σ ⟷
        sat (CONJ_disjoint (simp (Conj Qfin (DISJ (qps G))), Qeq)) I σ ∨
        (∃ Q∈(λy. (cp (Qfin[x → y]), insert (x, y) Qeq)) ‘ eqs x G. sat (CONJ_disjoint Q) I σ)
      using cov_sat_fin[of x Qfin G I σ] assms(3,4) fin(1) unsat
      by (auto simp: eval_empty_close sat_CONJ_disjoint nongens_def)
    finally show (∃ Q∈CONJ_disjoint ‘ ?Qfin. sat Q I σ) ⟷ (∃ Q∈CONJ_disjoint ‘ Qfin. sat Q I σ)
      by auto
  qed simp_all
next
  case (inf I)
  from assms have ∗:
    x ∈ fv Qfin (Qfin, Qeq) ∈ Qfin finite Qfin finite Qeq finite Qinf fv Qfin ⊆ fv Q Field Qeq ⊆ fv Q
    ∃ xys. distinct xys ∧ leftfresh Qfin xys ∧ set xys = Qeq
    by (auto simp: split_INV1_def fixfree_def nongens_def wf_state_def)
  with inf obtain σ where sat (Qfin ⊥ x) I σ ∨ (∃ Q ∈ Qinf. sat Q I σ)
   by (subst (asm) eval_simp_DISJ_closed) (auto simp: eval_empty_close sat_CONJ simp del: fv_CONJ)
  then show ?case
  proof (elim disjE)
    assume sat (Qfin ⊥ x) I σ
    then have infinite (eval Qfin I)
      by (rule cov_eval_inf[OF assms(4) ∗(1) inf(1)])
    then have infinite (eval_on (fv Q) (CONJ_disjoint (Qfin, Qeq)) I)
      by (rule infinite_eval_CONJ_disjoint[OF _ inf(1) ∗(6,7) _ ∗(8)]) simp
    with ∗ have infinite (eval_on (fv Q) (simp (DISJ (CONJ_disjoint ‘ Qfin))) I)
      by (elim infinite_Implies_mono_on[rotated 3]) (auto simp: sat_simp)
    with inf assms(1) show ?case
      by (auto simp: split_INV1_def assemble_def EVAL'_def split: if_splits)
  next
    assume ∃ Q∈Qinf. sat Q I σ
    with inf(1) assms(1) ∗ show ?case
     by (auto simp: split_INV1_def assemble_def EVAL'_def eval_simp_DISJ_closed eval_empty_close
         split: if_splits)
  qed
qed

lemma (in simplification) split_INV1_decreases:
  assumes split_INV1 Q (Qfin, Qinf) (Qfin, Qeq) ∈ fixfree Qfin x ∈ nongens Qfin cov x Qfin G
  shows ((nongens ∘ fst) ‘# mset_set (insert (simp (Conj Qfin (DISJ (qps G))), Qeq) (Qfin − {(Qfin,
Qeq)} ∪ (λy. (cp (Qfin[x → y]), insert (x, y) Qeq)) ‘ eqs x G)),
        (nongens ∘ fst) ‘# mset_set Qfin) ∈ mult {(X, Y). X ⊂ Y ∧ Y ⊆ fv Q}
  using assms by (intro split_step_in_mult) (auto simp: fixfree_def split_INV1_def wf_state_def)

lemma (in simplification) split_INV2_init:
  split_INV1 Q (Qfin, Qinf) ⟹ fixfree Qfin = {} ⟹ split_INV2 Q (Qfin, Qinf)
  by (auto simp: split_INV1_def split_INV2_def wf_state_def sr_def fixfree_def)

lemma (in simplification) split_INV2_I:
```

wf_state Q sr (*𝒬fin*, *𝒬inf*) ⟹ *EVAL′ Q* (*simp* (*DISJ* (*CONJ_disjoint* ‘ *𝒬fin*))) (*simp* (*DISJ* (*close* ‘ *𝒬inf*))) ⟹
split_INV2 Q (*𝒬fin*, *𝒬inf*)
unfolding *split_INV2_def assemble_def* **by** *auto*

lemma (**in** *simplification*) *split_INV2_step*:
assumes *split_INV2 Q* (*𝒬fin*, *𝒬inf*) (*Qfin*, *Qeq*) ∈ *inf 𝒬fin Q*
shows *split_INV2 Q* (*𝒬fin* − {(*Qfin*, *Qeq*)}, *insert* (*CONJ* (*Qfin*, *Qeq*)) *𝒬inf*)
proof (*intro split_INV2_I EVAL′_I*, *goal_cases wf fin inf*)
case *wf*
with *assms*(*1*) **show** *?case*
by (*auto simp*: *split_INV2_def wf_state_def*)
next
case (*fin I*)
with *assms* **have** *finite*[*simp*]: *finite 𝒬fin finite Qeq* **and**
unsat: ⋀σ. ¬ *sat* (*CONJ* (*Qfin*, *Qeq*)) *I* σ **and**
eval: *eval_on* (*fv Q*) (*simp* (*DISJ* (*CONJ_disjoint* ‘ *𝒬fin*))) *I* = *eval Q I*
by (*auto simp*: *split_INV2_def inf_def wf_state_def assemble_def EVAL′_def eval_simp_DISJ_closed eval_empty_close*)
from *eval* **show** *?case*
proof (*elim trans*[*rotated*], *unfold eval_on_simp*, *intro eval_DISJ_prune_unsat ballI allI*; (*elim DiffE imageE*; *hypsubst_thin*)*?*)
fix *Qpair* σ
assume *Qpair* ∈ *𝒬fin CONJ_disjoint Qpair* ∉ *CONJ_disjoint* ‘ (*𝒬fin* − {(*Qfin*, *Qeq*)})
with *unsat*[*of* σ] **show** ¬ *sat* (*CONJ_disjoint Qpair*) *I* σ
by (*cases Qeq* = *snd Qpair*; *cases Qpair*) (*auto simp*: *sat_CONJ_disjoint sat_CONJ*)
qed *auto*
next
case (*inf I*)
from *assms* **have** ∗:
(*Qfin*, *Qeq*) ∈ *𝒬fin finite 𝒬fin finite Qeq finite 𝒬inf fv Qfin* ⊆ *fv Q Field Qeq* ⊆ *fv Q*
by (*auto simp*: *split_INV2_def inf_def wf_state_def*)
with *inf* **obtain** σ **where** *sat Qfin I* σ ∧ (∀ (*x*, *y*) ∈ *Qeq*. σ *x* = σ *y*) ∨ (∃ *Q* ∈ *𝒬inf*. *sat Q I* σ)
by (*subst* (*asm*) *eval_simp_DISJ_closed*) (*auto simp*: *eval_empty_close sat_CONJ simp del*: *fv_CONJ*)
then show *?case*
proof (*elim disjE conjE*)
assume *sat Qfin I* σ ∀ (*x*, *y*) ∈ *Qeq*. σ *x* = σ *y*
with *assms* ∗ **have** *infinite* (*eval_on* (*fv Q*) (*CONJ_disjoint* (*Qfin*, *Qeq*)) *I*)
using *nonempty_disjointvars_infinite*[*of Qfin Qeq fv Q I* σ]
infinite_eval_on_extra_variables[*of fv Q CONJ_disjoint* (*Qfin*, *Qeq*) *I*, *OF _ _ exI*, *of* σ]
by (*cases fv* (*CONJ_disjoint* (*Qfin*, *Qeq*)) ⊂ *fv Q*) (*auto simp*: *inf_def sat_CONJ sat_CONJ_disjoint*)
with ∗ **have** *infinite* (*eval_on* (*fv Q*) (*simp* (*DISJ* (*CONJ_disjoint* ‘ *𝒬fin*))) *I*)
by (*elim infinite_Implies_mono_on*[*rotated 3*]) (*auto simp*: *sat_simp*)
with *inf assms*(*1*) **show** *?case*
by (*auto simp*: *split_INV2_def assemble_def EVAL′_def split*: *if_splits*)
next
assume ∃ *Q* ∈ *𝒬inf*. *sat Q I* σ
with *inf*(*1*) *assms*(*1*) ∗ **show** *infinite* (*eval Q I*)
by (*auto simp*: *split_INV2_def assemble_def EVAL′_def eval_simp_DISJ_closed eval_empty_close split*: *if_splits*)
qed
qed

lemma (**in** *simplification*) *split_INV2_decreases*:
split_INV2 Q (*𝒬fin*, *𝒬inf*) ⟹ (*Qfin*, *Qeq*) ∈ *Restrict_Frees.inf 𝒬fin Q* ⟹ *card* (*𝒬fin* − {(*Qfin*, *Qeq*)}) < *card 𝒬fin*
by (*rule psubset_card_mono*) (*auto simp*: *inf_def split_INV2_def wf_state_def*)

lemma (**in** *simplification*) *split_INV2_stop_fin_sr*:
inf $\mathcal{Q}fin$ $Q = \{\} \Longrightarrow$ *split_INV2* Q ($\mathcal{Q}fin$, $\mathcal{Q}inf$) $\Longrightarrow$ *assemble* ($\mathcal{Q}fin$, $\mathcal{Q}inf$) = ($Qfin$, $Qinf$) $\Longrightarrow$ *sr* $Qfin$
by (*auto 0 4 simp*: *split_INV2_def assemble_def wf_state_def inf_def*
intro!: *sr_simp sr_DISJ*[*of* _ *fv* Q] *sr_CONJ_disjoint*[*of* $\mathcal{Q}fin$ Q])

lemma (**in** *simplification*) *split_INV2_stop_inf_sr*:
split_INV2 Q ($\mathcal{Q}fin$, $\mathcal{Q}inf$) $\Longrightarrow$ *assemble* ($\mathcal{Q}fin$, $\mathcal{Q}inf$) = ($Qfin$, $Qinf$) $\Longrightarrow$ *fv* $Q' \subseteq$ *fv* $Qinf \Longrightarrow$ *rrb* Q' $\Longrightarrow$ *sr* Q'
using *fv_DISJ_close*[*of* $\mathcal{Q}inf$] *fv_simp*[*of DISJ* (*close* ‘ $\mathcal{Q}inf$)]
by (*auto simp*: *split_INV2_def assemble_def wf_state_def sr_def nongens_def*)

lemma (**in** *simplification*) *split_INV2_stop_FV*:
assumes *fv* $Q' \subseteq$ *fv* $Qinf$ *inf* $\mathcal{Q}fin$ $Q = \{\}$ *split_INV2* Q ($\mathcal{Q}fin$, $\mathcal{Q}inf$) *assemble* ($\mathcal{Q}fin$, $\mathcal{Q}inf$) = ($Qfin$, $Qinf$)
shows *FV* Q $Qfin$ Q'
proof −
have *simplified* Q' *fv* Q' = *fv* Q **if** $Q' \in$ *CONJ_disjoint* ‘ $\mathcal{Q}fin$ **for** Q'
using *that assms*(*2*,*3*)
by (*auto simp*: *split_INV2_def wf_state_def inf_def simplified_CONJ_disjoint*)
with *assms*(*1*,*3*,*4*) **show** *?thesis*
using *fv_simp_DISJ_eq*[*of CONJ_disjoint* ‘ $\mathcal{Q}fin$ *fv* Q] *fv_DISJ_close*[*of* $\mathcal{Q}inf$] *fv_simp*[*of DISJ* (*close* ‘ $\mathcal{Q}inf$)]
by (*auto simp*: *split_INV2_def assemble_def wf_state_def FV_def*)
qed

lemma (**in** *simplification*) *split_INV2_stop_EVAL*:
assumes *fv* $Q' \subseteq$ *fv* $Qinf$ *inf* $\mathcal{Q}fin$ $Q = \{\}$ *split_INV2* Q ($\mathcal{Q}fin$, $\mathcal{Q}inf$) *assemble* ($\mathcal{Q}fin$, $\mathcal{Q}inf$) = ($Qfin$, $Qinf$) $Qinf \triangleq Q'$
shows *EVAL* Q $Qfin$ Q'
proof −
have *simplified* Q' *fv* Q' = *fv* Q **if** $Q' \in$ *CONJ_disjoint* ‘ $\mathcal{Q}fin$ **for** Q'
using *that assms*(*2*,*3*)
by (*auto simp*: *split_INV2_def wf_state_def inf_def simplified_CONJ_disjoint*)
with *assms*(*1*,*3*,*4*,*5*) **show** *?thesis*
using *fv_simp_DISJ_eq*[*of CONJ_disjoint* ‘ $\mathcal{Q}fin$ *fv* Q] *fv_DISJ_close*[*of* $\mathcal{Q}inf$] *fv_simp*[*of DISJ* (*close* ‘ $\mathcal{Q}inf$)]
by (*auto simp*: *split_INV2_def assemble_def wf_state_def sr_def EVAL'_cong FV_def elim*!: *EVAL'_EVAL*)
qed

lemma (**in** *simplification*) *simplified_assemble*:
assemble ($\mathcal{Q}fin$, $\mathcal{Q}inf$) = ($Qfin$, $Qinf$) $\Longrightarrow$ *simplified* $Qfin$
by (*auto simp*: *assemble_def simplified_simp*)

lemma (**in** *simplification*) *split_correct*:
notes *cp.simps*[*simp del*]
shows *split* $Q \leq$ *split_spec* Q
unfolding *split_def split_spec_def Let_def*
by (*refine_vcg rb_correct*[*THEN order_trans*, *unfolded rb_spec_def*]
WHILEIT_rule[**where** *I*=*split_INV1* Q **and** *R*=*inv_image* (*mult* {(X, Y). $X \subset Y \wedge Y \subseteq$ *fv* Q}) (*image_mset* (*nongens o fst*) *o mset_set o fst*)]
WHILEIT_rule[**where** *I*=*split_INV2* Q **and** *R*=*measure* (λ($\mathcal{Q}fin$, _). *card* $\mathcal{Q}fin$)])
(*auto simp*: *wf_mult finite_subset_wf split_step_in_mult*
conj_disj_distribR ex_disj_distrib card_gt_0_iff image_image image_Un
insert_commute ac_simps UNION_singleton_eq_range simplified_assemble
split_INV1_init split_INV1_step split_INV1_decreases
split_INV2_init split_INV2_step split_INV2_decreases
split_INV2_stop_fin_sr split_INV2_stop_inf_sr split_INV2_stop_FV split_INV2_stop_EVAL)

6 Refining the Non-Deterministic *simplification.split* Function

definition *fixfree_impl* $\mathcal{Q}$ = *map* (*apsnd set*) (*filter* (λ(Q, _ :: (*nat* × *nat*) *list*). $\exists\, x \in fv\ Q$. *gen_impl* $x\ Q$ = []) (*sorted_list_of_set* ((*apsnd sorted_list_of_set*) ' $\mathcal{Q}$)))

definition *nongens_impl* Q = *filter* (λx. *gen_impl* $x\ Q$ = []) (*sorted_list_of_set* (*fv* Q))

lemma *set_nongens_impl*: *set* (*nongens_impl* Q) = *nongens* Q
by (*auto simp*: *nongens_def nongens_impl_def set_gen_impl simp flip*: *List.set_empty*)

lemma *set_fixfree_impl*: *finite* $\mathcal{Q} \Longrightarrow \forall$(_, *Qeq*) $\in \mathcal{Q}$. *finite Qeq* $\Longrightarrow$ *set* (*fixfree_impl* $\mathcal{Q}$) = *fixfree* $\mathcal{Q}$
by (*fastforce simp*: *fixfree_def nongens_def fixfree_impl_def set_gen_impl image_iff apsnd_def map_prod_def simp flip*: *List.set_empty split*: *prod.splits intro*: *exI*[*of* _ *sorted_list_of_set* _])

lemma *fixfree_empty_iff*: *finite* $\mathcal{Q} \Longrightarrow \forall$(_, *Qeq*) $\in \mathcal{Q}$. *finite Qeq* $\Longrightarrow$ *fixfree* $\mathcal{Q} \neq \{\} \longleftrightarrow$ *fixfree_impl* $\mathcal{Q} \neq$ []
by (*auto simp*: *set_fixfree_impl dest*: *arg_cong*[*of* _ _ *set*] *simp flip*: *List.set_empty*)

definition *inf_impl* $\mathcal{Q}fin\ Q$ =
map (*apsnd set*) (*filter* (λ(*Qfix*, *xys*). *disjointvars Qfix* (*set xys*) $\neq \{\} \vee$ *fv Qfix* $\cup$ *Field* (*set xys*) $\neq$ *fv* Q) (*sorted_list_of_set* ((*apsnd sorted_list_of_set*) ' $\mathcal{Q}fin$)))

lemma *set_inf_impl*: *finite* $\mathcal{Q}fin \Longrightarrow \forall$(_, *Qeq*) $\in \mathcal{Q}fin$. *finite Qeq* $\Longrightarrow$ *set* (*inf_impl* $\mathcal{Q}fin\ Q$) = *inf* $\mathcal{Q}fin\ Q$
by (*fastforce simp*: *inf_def inf_impl_def image_iff*)

lemma *inf_empty_iff*: *finite* $\mathcal{Q}fin \Longrightarrow \forall$(_, *Qeq*) $\in \mathcal{Q}fin$. *finite Qeq* $\Longrightarrow$ *inf* $\mathcal{Q}fin\ Q \neq \{\} \longleftrightarrow$ *inf_impl* $\mathcal{Q}fin\ Q \neq$ []
by (*auto simp*: *set_inf_impl dest*: *arg_cong*[*of* _ _ *set*] *simp flip*: *List.set_empty*)

definition (**in** *simplification*) *split_impl* :: ('*a* :: {*infinite*, *linorder*}, '*b* :: *linorder*) *fmla* ⇒ (('*a*, '*b*) *fmla* × ('*a*, '*b*) *fmla*) *nres* **where**
split_impl Q = *do* {
  Q' ← *rb_impl* Q;
  $\mathcal{Q}pair$ ← *WHILE*
    (λ($\mathcal{Q}fin$, _). *fixfree_impl* $\mathcal{Q}fin \neq$ []) (λ($\mathcal{Q}fin$, $\mathcal{Q}inf$). *do* {
      (*Qfix*, *Qeq*) ← *RETURN* (*hd* (*fixfree_impl* $\mathcal{Q}fin$));
      x ← *RETURN* (*hd* (*nongens_impl Qfix*));
      G ← *RETURN* (*hd* (*cov_impl* x *Qfix*));
      let $\mathcal{Q}fin$ = $\mathcal{Q}fin$ − {(*Qfix*, *Qeq*)} ∪
        {(*simp* (*Conj Qfix* (*DISJ* (*qps* G))), *Qeq*)} ∪
        ($\bigcup y \in eqs\ x\ G$. {(*cp* (*Qfix*[$x \to y$]), *Qeq* ∪ {(x,y)})});
      let $\mathcal{Q}inf$ = $\mathcal{Q}inf$ ∪ {*cp* (*Qfix* ⊥ x)};
      RETURN ($\mathcal{Q}fin$, $\mathcal{Q}inf$)})
    ({(Q', {})}, {});
  $\mathcal{Q}pair$ ← *WHILE*
    (λ($\mathcal{Q}fin$, _). *inf_impl* $\mathcal{Q}fin\ Q \neq$ []) (λ($\mathcal{Q}fin$, $\mathcal{Q}inf$). *do* {
      $\mathcal{Q}pair$ ← *RETURN* (*hd* (*inf_impl* $\mathcal{Q}fin\ Q$));
      let $\mathcal{Q}fin$ = $\mathcal{Q}fin$ − {$\mathcal{Q}pair$};
      let $\mathcal{Q}inf$ = $\mathcal{Q}inf$ ∪ {*CONJ* $\mathcal{Q}pair$};
      RETURN ($\mathcal{Q}fin$, $\mathcal{Q}inf$)})
    $\mathcal{Q}pair$;

```
    let (Qfin, Qinf) = assemble Qpair;
    Qinf ← rb_impl Qinf;
    RETURN (Qfin, Qinf)}

lemma (in simplification) split_INV2_imp_split_INV1: split_INV2 Q Qpair ⟹ split_INV1 Q Qpair
  unfolding split_INV1_def split_INV2_def wf_state_def sr_def by auto

lemma hd_fixfree_impl_props:
  assumes finite Q ∀(_, Qeq) ∈ Q. finite Qeq fixfree_impl Q ≠ []
  shows hd (fixfree_impl Q) ∈ Q nongens (fst (hd (fixfree_impl Q))) ≠ {}
proof −
  from hd_in_set[of fixfree_impl Q] assms(3) have hd (fixfree_impl Q) ∈ set (fixfree_impl Q)
    by blast
  then have hd (fixfree_impl Q) ∈ fixfree Q
    by (auto simp: set_fixfree_impl assms(1,2))
  then show hd (fixfree_impl Q) ∈ Q nongens (fst (hd (fixfree_impl Q))) ≠ {}
    unfolding fixfree_def by auto
qed

lemma (in simplification) split_impl_refines_split: split_impl Q ≤ split Q
  apply (unfold split_def split_impl_def Let_def)
  supply rb_impl_refines_rb[refine_mono]
  apply refine_mono
   apply (rule order_trans[OF WHILE_le_WHILEI[where I=split_INV1 Q]])
   apply (rule order_trans[OF WHILEI_le_WHILEIT])
   apply (rule WHILEIT_refine[OF _ _ _ refine_IdI, THEN refine_IdD])
     apply (simp_all only: pair_in_Id_conv split: prod.splits) [4]
    apply (intro allI impI, hypsubst_thin)
    apply (subst fixfree_empty_iff; auto simp: split_INV1_def wf_state_def)
    apply (intro allI impI, simp only: prod.inject, elim conjE, hypsubst_thin)
   apply refine_mono
  apply (subst set_fixfree_impl[symmetric]; auto simp: split_INV1_def wf_state_def intro!: hd_in_set)
  apply clarsimp
  subgoal for Q' Qfin Qinf Qfix Qeq Qfix' Qeq'
    using hd_fixfree_impl_props(2)[of Qfin]
    by (force simp: split_INV1_def wf_state_def set_nongens_impl[symmetric] dest!: sym[of (Qfix', _)]
intro!: hd_in_set)
   apply clarsimp
  subgoal for Q' Qfin Qinf Qfix Qeq Qfix' Qeq'
    apply (intro RETURN_rule cov_impl_cov hd_in_set rrb_cov_impl)
    using hd_fixfree_impl_props(1)[of Qfin]
    by (force simp: split_INV1_def wf_state_def dest!: sym[of (Qfix', _)])
   apply (rule order_trans[OF WHILE_le_WHILEI[where I=split_INV1 Q]])
   apply (rule order_trans[OF WHILEI_le_WHILEIT])
  apply (rule WHILEIT_refine[OF _ _ _ refine_IdI, THEN refine_IdD])
     apply (simp_all only: pair_in_Id_conv split_INV2_imp_split_INV1 split: prod.splits) [4]
   apply (intro allI impI, simp only: prod.inject, elim conjE, hypsubst_thin)
   apply (subst inf_empty_iff; auto simp: split_INV2_def wf_state_def)
  apply (intro allI impI, simp only: prod.inject, elim conjE, hypsubst_thin)
  apply refine_mono
  apply (subst set_inf_impl[symmetric]; auto simp: split_INV2_def wf_state_def intro!: hd_in_set)
  done

definition (in simplification) split_impl_det :: ('a :: {infinite, linorder}, 'b :: linorder) fmla ⇒ (('a, 'b)
fmla × ('a, 'b) fmla) dres where
  split_impl_det Q = do {
    Q' ← rb_impl_det Q;
    Qpair ← dWHILE
```

```
    (λ(Qfin, _). fixfree_impl Qfin ≠ []) (λ(Qfin, Qinf). do {
      (Qfix, Qeq) ← dRETURN (hd (fixfree_impl Qfin));
      x ← dRETURN (hd (nongens_impl Qfix));
      G ← dRETURN (hd (cov_impl x Qfix));
      let Qfin = Qfin − {(Qfix, Qeq)} ∪
        {(simp (Conj Qfix (DISJ (qps G))), Qeq)} ∪
        (⋃ y ∈ eqs x G. {(cp (Qfix[x → y]), Qeq ∪ {(x,y)})});
      let Qinf = Qinf ∪ {cp (Qfix ⊥ x)};
      dRETURN (Qfin, Qinf)})
    ({(Q', {})}, {});
  Qpair ← dWHILE
    (λ(Qfin, _). inf_impl Qfin Q ≠ []) (λ(Qfin, Qinf). do {
      Qpair ← dRETURN (hd (inf_impl Qfin Q));
      let Qfin = Qfin − {Qpair};
      let Qinf = Qinf ∪ {CONJ Qpair};
      dRETURN (Qfin, Qinf)})
    Qpair;
  let (Qfin, Qinf) = assemble Qpair;
  Qinf ← rb_impl_det Qinf;
  dRETURN (Qfin, Qinf)}
```

lemma (**in** *simplification*) *split_impl_det_refines_split_impl*: *nres_of* (*split_impl_det Q*) ≤ *split_impl Q*
unfolding *split_impl_def split_impl_det_def Let_def*
by (*refine_transfer rb_impl_det_refines_rb_impl*)

lemmas (**in** *simplification*) *SPLIT_correct* =
split_impl_det_refines_split_impl[*THEN order_trans*, *OF*
split_impl_refines_split[*THEN order_trans*, *OF*
split_correct]]

7 Examples

global_interpretation *extra_cp*: *simplification cp cpropagated*
defines *RB* = *simplification.rb_impl_det cp*
and *assemble* = *simplification.assemble cp*
and *SPLIT* = *simplification.split_impl_det cp*
by *standard* (*auto simp only*: *sat_cp fv_cp rrb_cp gen_Gen_cp cpropagated_cp cpropagated_cp_triv cpropagated_sub Let_def is_Bool_def fv.simps cp.simps cpropagated_simps nocp.simps cpropagated_nocp split*: *if_splits*)

7.1 Restricting Bounds in the "Suspicious Users" Query

context
fixes *b s p u* :: *nat* **and** *B P S*
defines *b* ≡ *0*
and *s* ≡ *Suc 0*
and *p* ≡ *Suc* (*Suc 0*)
and *u* ≡ *Suc* (*Suc* (*Suc 0*))
and *B* ≡ λ*b*. *Pred* ''*B*'' [*Var b*] :: (*string*, *string*) *fmla*
and *P* ≡ λ*b p*. *Pred* ''*P*'' [*Var b*, *Var p*] :: (*string*, *string*) *fmla*
and *S* ≡ λ*p u s*. *Pred* ''*S*'' [*Var p*, *Var u*, *Var s*] :: (*string*, *string*) *fmla*
notes *cp.simps*[*simp del*]
begin

definition *Q_susp_user* **where**

```
  Q_susp_user = Conj (B b) (Exists s (Forall p (Impl (P b p) (S p u s))))
definition Q_susp_user_rb :: (string, string) fmla where
  Q_susp_user_rb = Conj (B b) (Disj (Exists s (Conj (Forall p (Impl (P b p) (S p u s))) (Exists p (S p
u s)))) (Forall p (Neg (P b p))))

lemma ex_rb_Q_susp_user: the_res (RB Q_susp_user) = Q_susp_user_rb
  by code_simp

end
```

7.2 Splitting a Disjunction of Predicates

```
context
  fixes x y :: nat and B P
  defines x ≡ 0
    and y ≡ 1
    and B ≡ λb. Pred ''B'' [Var b] :: (string, string) fmla
    and P ≡ λb p. Pred ''P'' [Var b, Var p] :: (string, string) fmla
  notes cp.simps[simp del]
begin

definition Q_disj where
  Q_disj = Disj (B x) (P x y)
definition Q_disj_split_fin :: (string, string) fmla where
  Q_disj_split_fin = Conj (Disj (B x) (P x y)) (P x y)
definition Q_disj_split_inf :: (string, string) fmla where
  Q_disj_split_inf = Exists x (B x)

lemma ex_split_Q_disj: the_res (SPLIT Q_disj) = (Q_disj_split_fin, Q_disj_split_inf)
  by code_simp

end
```

7.3 Splitting a Conjunction with an Equality

```
context
  fixes x u v :: nat and B
  defines x ≡ 0
    and u ≡ 1
    and v ≡ 2
    and B ≡ λb. Pred ''B'' [Var b] :: (string, string) fmla
  notes cp.simps[simp del]
begin

definition Q_eq where
  Q_eq = Conj (B x) (u ≈ v)
definition Q_eq_split_fin :: (string, string) fmla where
  Q_eq_split_fin = Bool False
definition Q_eq_split_inf :: (string, string) fmla where
  Q_eq_split_inf = Exists x (B x)

lemma ex_split_Q_eq: the_res (SPLIT Q_eq) = (Q_eq_split_fin, Q_eq_split_inf)
  by code_simp

end
```

7.4 Splitting the "Suspicious Users" Query

```
context
```

fixes $b\ s\ p\ u :: nat$ **and** $B\ P\ S$
defines $b \equiv 0$
and $s \equiv Suc\ 0$
and $p \equiv Suc\ (Suc\ 0)$
and $u \equiv Suc\ (Suc\ (Suc\ 0))$
and $B \equiv \lambda b.\ Pred\ ''B''\ [Var\ b] :: (string, string)\ fmla$
and $P \equiv \lambda b\ p.\ Pred\ ''P''\ [Var\ b,\ Var\ p] :: (string, string)\ fmla$
and $S \equiv \lambda p\ u\ s.\ Pred\ ''S''\ [Var\ p,\ Var\ u,\ Var\ s] :: (string, string)\ fmla$
notes *cp.simps*[*simp del*]
begin

definition $Q\_susp\_user\_split\_fin = Conj\ Q\_susp\_user\_rb\ (Exists\ s\ (Exists\ p\ (S\ p\ u\ s)))$
definition $Q\_susp\_user\_split\_inf = Exists\ b\ (Conj\ (B\ b)\ (Forall\ p\ (Neg\ (P\ b\ p))))$

lemma *ex_split_Q_susp_user*: $the\_res\ (SPLIT\ Q\_susp\_user) = (Q\_susp\_user\_split\_fin, Q\_susp\_user\_split\_inf)$
by *code_simp*

end

8 Collected Results from the ICDT'22 Paper

global_interpretation *icdt22*: *simplification* $\lambda x.\ x\ \lambda x.\ True$
by *standard auto*

lemma *cov_eval_fin*:
assumes $cov\ x\ (Q :: ('a :: \{infinite, linorder\}, 'b :: linorder)\ fmla)\ G\ x \in fv\ Q$
$finite\ (adom\ I)\ \bigwedge\sigma.\ \neg\ sat\ (Q \perp x)\ I\ \sigma$
shows $eval\ Q\ I = eval\ (Disj\ (Conj\ Q\ (DISJ\ (qps\ G)))\ (DISJ\ ((\lambda y.\ Conj\ (cp\ (Q[x \to y]))\ (x \approx y))\ `eqs\ x\ G)))\ I$
using *assms*
by (*intro trans*[*OF icdt22.cov_eval_fin*[*OF assms*]])
(*auto 0 3 simp*: *eval_def fv_subst intro*!: *arg_cong*[*of _ _* $\lambda X.\ eval\_on\ X\ \_\ \_$]
dest!: *fv_DISJ*[*THEN set_mp, rotated 1*] *fv_cp*[*THEN set_mp*]
dest: *cov_fv*[*OF _ _ qps_in*] *cov_fv*[*OF _ _ eqs_in*])

Remapping the formalization statements to the lemma's from the paper:

lemmas *icdt22_lemma_1* = *gen_fv gen_sat gen_cp_erase*
lemmas *icdt22_definition_2* = *sub.simps nongens_def rrb_def sr_def*
lemmas *icdt22_lemma_3* = *ex_cov cov_sat_erase*
lemmas *icdt22_lemma_4* = *cov_fv cov_equiv*[*OF _ refl*]
lemmas *icdt22_lemma_5* = *icdt22.cov_Exists_equiv*
lemmas *icdt22_example_6* = *ex_rb_Q_susp_user*[*unfolded Q_susp_user_def Q_susp_user_rb_def*]
lemmas *icdt22_lemma_7* = *cov_eval_fin cov_eval_inf*
lemmas *icdt22_lemma_8* = *inres_SPEC*[*OF _ icdt22.rb_correct*[*unfolded icdt22.rb_spec_def*, *simplified*], *of Q Q'* **for** *Q Q'*]
lemmas *icdt22_lemma_9* = *inres_SPEC*[*OF _ icdt22.split_correct*[*unfolded icdt22.split_spec_def FV_def EVAL_def*, *simplified*],
of Q (*Qfin*, *Qinf*) **for** *Q Qfin Qinf*, *simplified*]
lemmas *icdt22_example_10* = *ex_split_Q_disj*[*unfolded Q_disj_def Q_disj_split_fin_def Q_disj_split_inf_def*]
lemmas *icdt22_example_11* = *ex_split_Q_eq*[*unfolded Q_eq_def Q_eq_split_fin_def Q_eq_split_inf_def*]
lemmas *icdt22_example_12* = *ex_split_Q_susp_user*[*unfolded Q_susp_user_def Q_susp_user_split_fin_def Q_susp_user_split_inf_def*]

Additionally, here are the correctness statements for the algorithm variants with intermediate

constant propagation (which are used in the examples):

lemmas *icdt22_lemma_8′ = inres_SPEC[OF _ extra_cp.RB_correct[unfolded extra_cp.rb_spec_def], simplified, of Q Q′* **for** *Q Q′]*
lemmas *icdt22_lemma_9′ = inres_SPEC[OF _ extra_cp.SPLIT_correct[unfolded extra_cp.split_spec_def FV_def EVAL_def, simplified],*
of Q (Qfin, Qinf) **for** *Q Qfin Qinf, simplified]*

Now, we summarize the formally verified results from our ICDT'22 paper [2]:

icdt22_lemma_1: $[\![gen\ x\ Q\ G;\ Qqp \in G]\!] \Longrightarrow x \in fv\ Qqp \wedge fv\ Qqp \subseteq fv\ Q$

$[\![gen\ x\ Q\ G;\ sat\ Q\ I\ \sigma]\!] \Longrightarrow \exists\, Qqp{\in}G.\ sat\ Qqp\ I\ \sigma$

$[\![gen\ x\ Q\ G;\ Qqp \in G]\!] \Longrightarrow cp\ (Qqp \perp x) = Bool\ False$

icdt22_definition_2: $sub\ (Bool\ t) = \{Bool\ t\}$

$sub\ (Pred\ p\ ts) = \{Pred\ p\ ts\}$

$sub\ (fmla.Eq\ x\ t) = \{fmla.Eq\ x\ t\}$

$sub\ (Neg\ Q) = insert\ (Neg\ Q)\ (sub\ Q)$

$sub\ (Conj\ Q1\ Q2) = insert\ (Conj\ Q1\ Q2)\ (sub\ Q1 \cup sub\ Q2)$

$sub\ (Disj\ Q1\ Q2) = insert\ (Disj\ Q1\ Q2)\ (sub\ Q1 \cup sub\ Q2)$

$sub\ (Exists\ z\ Q) = insert\ (Exists\ z\ Q)\ (sub\ Q)$

$nongens\ Q = \{x \in fv\ Q.\ \neg\ Gen\ x\ Q\}$

$rrb\ Q = (\forall\, y\ Qy.\ Exists\ y\ Qy \in sub\ Q \longrightarrow Gen\ y\ Qy)$

$sr\ Q = (rrf\ Q \wedge rrb\ Q)$

icdt22_lemma_3: $[\![rrb\ Q;\ x \in fv\ Q]\!] \Longrightarrow \exists\, G.\ cov\ x\ Q\ G$

$[\![cov\ x\ Q\ G;\ sat\ (Neg\ (Disj\ (DISJ\ (qps\ G))\ (DISJ\ ((\approx)\ x\ `\ eqs\ x\ G))))\ I\ \sigma]\!] \Longrightarrow sat\ Q\ I\ \sigma = sat\ (cp\ (Q \perp x))\ I\ \sigma$

icdt22_lemma_4: $[\![cov\ x\ Q\ G;\ x \in fv\ Q;\ Qqp \in G]\!] \Longrightarrow x \in fv\ Qqp \wedge fv\ Qqp \subseteq fv\ Q$

$cov\ x\ Q\ G \Longrightarrow Q \triangleq Disj\ (Conj\ Q\ (DISJ\ (qps\ G)))\ (Disj\ (DISJ\ ((\lambda y.\ Conj\ (cp\ (Q[x \rightarrow y]))\ (x \approx y))\ `\ eqs\ x\ G))\ (Conj\ (Q \perp x)\ (Neg\ (Disj\ (DISJ\ (qps\ G))\ (DISJ\ ((\approx)\ x\ `\ eqs\ x\ G))))))$

icdt22_lemma_5: $[\![cov\ x\ Q\ G;\ x \in fv\ Q]\!] \Longrightarrow Exists\ x\ Q \triangleq Disj\ (Exists\ x\ (Conj\ Q\ (DISJ\ (qps\ G))))\ (Disj\ (DISJ\ ((\lambda y.\ cp\ (Q[x \rightarrow y]))\ `\ eqs\ x\ G))\ (cp\ (Q \perp x)))$

icdt22_example_6: $the\_res\ (RB\ (Conj\ (Pred\ ''B''\ [Var\ 0])\ (Exists\ (Suc\ 0)\ (Forall\ (Suc\ (Suc\ 0))\ (Impl\ (Pred\ ''P''\ [Var\ 0,\ Var\ (Suc\ (Suc\ 0))])\ (Pred\ ''S''\ [Var\ (Suc\ (Suc\ 0)),\ Var\ (Suc\ (Suc\ (Suc\ 0))),\ Var\ (Suc\ 0)])))))) = Conj\ (Pred\ ''B''\ [Var\ 0])\ (Disj\ (Exists\ (Suc\ 0)\ (Conj\ (Forall\ (Suc\ (Suc\ 0))\ (Impl\ (Pred\ ''P''\ [Var\ 0,\ Var\ (Suc\ (Suc\ 0))])\ (Pred\ ''S''\ [Var\ (Suc\ (Suc\ 0)),\ Var\ (Suc\ (Suc\ (Suc\ 0))),\ Var\ (Suc\ 0)])))\ (Exists\ (Suc\ (Suc\ 0))\ (Pred\ ''S''\ [Var\ (Suc\ (Suc\ 0)),\ Var\ (Suc\ (Suc\ (Suc\ 0))),\ Var\ (Suc\ 0)]))))\ (Forall\ (Suc\ (Suc\ 0))\ (Neg\ (Pred\ ''P''\ [Var\ 0,\ Var\ (Suc\ (Suc\ 0))])))))$

icdt22_lemma_7: $[\![cov\ x\ Q\ G;\ x \in fv\ Q;\ finite\ (adom\ I);\ \bigwedge\sigma.\ \neg\ sat\ (Q \perp x)\ I\ \sigma]\!] \Longrightarrow eval\ Q\ I = eval\ (Disj\ (Conj\ Q\ (DISJ\ (qps\ G)))\ (DISJ\ ((\lambda y.\ Conj\ (cp\ (Q[x \rightarrow y]))\ (x \approx y))\ `\ eqs\ x\ G)))\ I$

$[\![cov\ x\ Q\ G;\ x \in fv\ Q;\ finite\ (adom\ I);\ sat\ (Q \perp x)\ I\ \sigma]\!] \Longrightarrow infinite\ (eval\ Q\ I)$

icdt22_lemma_8: $inres\ (icdt22.rb\ Q)\ Q' \Longrightarrow rrb\ Q' \wedge Q \triangleq Q' \wedge fv\ Q' \subseteq fv\ Q$

icdt22_lemma_9: *inres* (*icdt22.split Q*) (*Qfin*, *Qinf*) $\Longrightarrow$ *sr Qfin* $\wedge$ *sr Qinf* $\wedge$ (*fv Qfin* = *fv Q* $\vee$ *Qfin* = *Bool False*) $\wedge$ *fv Qinf* = {} $\wedge$ ($\forall$ *I*. *finite* (*adom I*) $\longrightarrow$ (*if eval Qinf I* = {} *then eval Qfin I* = *eval Q I else infinite* (*eval Q I*)))

icdt22_lemma_8': *inres* (*nres_of* (*RB Q*)) *Q'* $\Longrightarrow$ *rrb Q'* $\wedge$ *cpropagated Q'* $\wedge$ *Q* $\triangleq$ *Q'* $\wedge$ *fv Q'* $\subseteq$ *fv Q*

icdt22_lemma_9': *inres* (*nres_of* (*SPLIT Q*)) (*Qfin*, *Qinf*) $\Longrightarrow$ *sr Qfin* $\wedge$ *sr Qinf* $\wedge$ (*fv Qfin* = *fv Q* $\vee$ *Qfin* = *Bool False*) $\wedge$ *fv Qinf* = {} $\wedge$ ($\forall$ *I*. *finite* (*adom I*) $\longrightarrow$ (*if eval Qinf I* = {} *then eval Qfin I* = *eval Q I else infinite* (*eval Q I*))) $\wedge$ *cpropagated Qfin* $\wedge$ *cpropagated Qinf*

icdt22_example_10: *the_res* (*SPLIT* (*Disj* (*Pred* ''*B*'' [*Var 0*]) (*Pred* ''*P*'' [*Var 0*, *Var 1*]))) = (*Conj* (*Disj* (*Pred* ''*B*'' [*Var 0*]) (*Pred* ''*P*'' [*Var 0*, *Var 1*])) (*Pred* ''*P*'' [*Var 0*, *Var 1*]), *Exists 0* (*Pred* ''*B*'' [*Var 0*]))

icdt22_example_11: *the_res* (*SPLIT* (*Conj* (*Pred* ''*B*'' [*Var 0*]) (*1* $\approx$ *2*))) = (*Bool False*, *Exists 0* (*Pred* ''*B*'' [*Var 0*]))

icdt22_example_12: *the_res* (*SPLIT* (*Conj* (*Pred* ''*B*'' [*Var 0*]) (*Exists* (*Suc 0*) (*Forall* (*Suc* (*Suc 0*)) (*Impl* (*Pred* ''*P*'' [*Var 0*, *Var* (*Suc* (*Suc 0*))]) (*Pred* ''*S*'' [*Var* (*Suc* (*Suc 0*)), *Var* (*Suc* (*Suc* (*Suc 0*))), *Var* (*Suc 0*)])))))) = (*Conj Q_susp_user_rb* (*Exists* (*Suc 0*) (*Exists* (*Suc* (*Suc 0*)) (*Pred* ''*S*'' [*Var* (*Suc* (*Suc 0*)), *Var* (*Suc* (*Suc* (*Suc 0*))), *Var* (*Suc 0*)]))), *Exists 0* (*Conj* (*Pred* ''*B*'' [*Var 0*]) (*Forall* (*Suc* (*Suc 0*)) (*Neg* (*Pred* ''*P*'' [*Var 0*, *Var* (*Suc* (*Suc 0*))])))))

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Raszyk, D. A. Basin, S. Krstic, and D. Traytel. Practical relational calculus query evaluation. In D. Olteanu and N. Vortmeier, editors, *ICDT 2022*, volume 220 of *LIPIcs*, pages 11:1–11:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.